Chapter

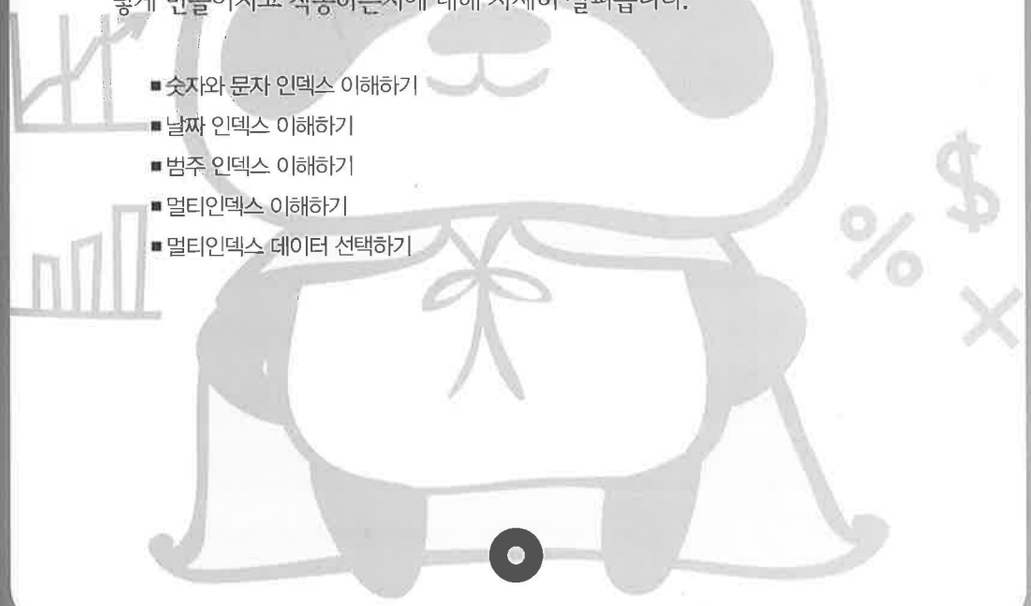
4

판다스 Index 클래스 이해하기

시리즈와 판다스에서 검색하거나 연산할 때 정수나 문자열 레이블로 내부의 원소에 접근했습니다. 정수나 문자열의 레이블은 판다스 안의 Index 클래스와 같이 인덱스를 만드는 클래스에 의해 만들어지는 것입니다.

이 장에서는 인덱스를 검색하기 위한 다양한 방법들을 알아봅니다. 시리즈나 데이터 프레임 안의 행과 열의 정보를 관리하는 레이블을 만드는 방법, 숫자, 날짜, 문자열, 범주 등으로 구성된 단일 레이블 체계와 복수 레이블 체계가 어떻게 만들어지고 작동하는지에 대해 자세히 살펴봅니다.

- 숫자와 문자 인덱스 이해하기
- 날짜 인덱스 이해하기
- 범주 인덱스 이해하기
- 멀티인덱스 이해하기
- 멀티인덱스 데이터 선택하기



4.1 숫자와 문자 인덱스 처리

판다스에서 시리즈나 데이터 프레임을 만들 때 레이블의 구성을 지정하지 않아도 내부에서 행과 열에 기본으로 숫자형 인덱스가 생깁니다. 그러나 문자열 자료형으로 레이블을 만들 때는 레이블에 대한 정보를 명기해야 합니다.

이 장에서는 판다스의 시리즈와 데이터 프레임을 만들 때 .index, .columns 속성이 어떤 클래스의 정보를 관리하는지를 알아봅니다. 그중 가장 기본인 숫자와 문자 인덱스를 생성하는 방법을 배웁니다.

4.1.1 숫자와 문자 인덱스

숫자와 문자에 대한 기본 인덱스도 클래스에 의해 생성된 객체들입니다. 숫자와 문자에 대한 인덱스 클래스의 속성과 메소드들을 알아봅니다.

■ Index 클래스 확인

인덱스는 행과 열에 대한 레이블을 구성하는 객체를 만듭니다. 데이터를 관리하는 시리즈와 데이터 프레임 클래스와 구조가 다릅니다. 인덱스 클래스의 기준부터 예제를 통해 알아봅니다.

[예제 4-1] 숫자와 문자 인덱스 생성하기

판다스의 pd.Index를 조회하면 Index 클래스라는 것을 알 수 있습니다. Index 생성자에 정수를 원소로 갖는 리스트를 넣고 객체를 생성합니다. 이 생성된 객체를 type 클래스를 통해 어떤 클래스로 만들어졌는지를 확인하면 Int64Index 클래스라 보여줍니다.

```
In :  pd.Index

Out:  pandas.core.indexes.base.Index

In :  idx1 = pd.Index([1, 2, 3, 4])

In :  type(idx1)

Out:  pandas.core.indexes.numeric.Int64Index
```

이번에는 Index 클래스의 객체를 만들기 위해 파이썬 range 함수를 사용해서 만들면 그 객체의 클래스는 RangeIndex라는 것을 알 수 있습니다. 판다스에서 자동으로 만들어지는 정수 인덱스가 RangeIndex인 이유는 실제 정수 인덱스보다 적은 메모리 양을 사용하기 때문입니다.

```
In :  idx2 = pd.Index(range(1,4))
In :  type(idx2)
Out:  pandas.core.indexes.range.RangeIndex
```

두 클래스로 만들어진 객체 안의 숫자 레이블이 관리되는데 이 레이블에 대한 데이터 자료형을 .dtype 속성으로 확인하면 int64라는 것을 알 수 있습니다.

```
In :  idx1.dtype, idx2.dtype
Out:  (dtype('int64'), dtype('int64'))
```

문자열을 원소로 갖는 리스트를 인자로 받고 Index 생성자로 인덱스 객체를 만듭니다. Index 클래스임을 알 수 있습니다. 판다스는 Index 클래스를 가지고 Int64Inex, RangeIndex, Index의 객체를 생성합니다.

또한 문자열 레이블을 가지는 Index 객체의 자료형을 .dtype 속성으로 확인하면 파이썬 문자열이 들어왔으므로 판다스의 object 자료형이라는 것을 알 수 있습니다.

```
In :  idx_s = pd.Index(['a', 'b','c'])
In :  idx_s
Out:  Index(['a', 'b', 'c'], dtype='object')
```

Index 클래스에 의해 만들어진 객체도 시리즈나 데이터 프레임처럼 데이터를 .values 속성에 관리하고 이 인덱스의 차원과 개수 정보는 .shape 속성으로 관리합니다.

```
In :  idx_s.values
Out:  array(['a', 'b', 'c'], dtype=object)
```



```
In :  idx_s.shape
Out:  (3,)
```

숫자 인덱스에는 정수형 이외의 실수형 인덱스도 있습니다. Index 클래스 생성자에서 매개변수 dtype='float'으로 지정하여 생성하면 만들어집니다.

```
In :  idx_f = pd.Index([1, 2, 3, 4],dtype='float')
In :  idx_f
Out:  Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')
```

4.1.2 숫자와 문자 인덱스 주요 특징

Index 클래스로 숫자와 문자 인덱스 객체인 배열을 만들었습니다. 이제 Index 내부에서 관리하는 속성과 메소드들을 확인하여 Index 클래스의 주요한 특징이 배열의 특징과 큰 차이가 없음을 확인하려고 합니다.

■숫자와 문자 인덱스 클래스 주요 특징

Index 클래스에서는 레이블의 정보 등을 검색하고 내부의 레이블도 변경할 수 있는 다양한 메소드를 갖고 있습니다. 예제를 통해 메소드가 어떻게 실행되는지 알아봅니다.

[예제 4-2] 숫자와 문자 인덱스 특징 파악하기

[예제 4-1]에서 만들어진 숫자 인덱스의 첫 번째 레이블 정보를 알아보려면 시리즈나 데이터 프레임의 원소를 검색하는 방식과 같이 인덱스를 검색할 때 쓰는 대괄호를 사용하면 됩니다.

인덱스 클래스도 배열이기 때문에 부분집합을 검색하는 슬라이스 처리도 가능합니다. 슬라이스 처리를 할 때에는 사본을 만드는 것이 아니라 기존에 있는 특정 부분을 검색해서 처리하는 것을 알 수 있습니다.

```
In :  idx1[0]
Out:  1
```

```
In :  idx_s[:]

Out:  Index(['a', 'b', 'c'], dtype='object')
```

판다스의 검색이 특징인 팬시 검색과 마스킹 검색도 배열의 특징이 있으므로 바로 적용할 수 있습니다.

팬시 검색은 인자로 리스트를 받으면 결과도 현재 자료형과 똑같은 사본을 만듭니다. 마스킹 검색은 비교 연산의 결과를 논리값으로 보관한 시리즈이고 이를 검색으로 사용합니다.

```
In :  idx1[[0]]

Out:  Int64Index([1], dtype='int64')

In :  idx1[idx1 < 3 ]

Out:  Int64Index([1, 2], dtype='int64')
```

시리즈나 데이터 프레임처럼 행 단위로 검색하기 위한 속성인 인덱서가 없습니다. .loc로 검색하면 인덱서가 없다고 예외를 발생시킵니다.

```
In :  try :
          idx1.loc[0]
      except Exception as e :
          print(e)

Out:  'Int64Index' object has no attribute 'loc'
```

판다스의 인덱스 클래스는 불변(immutable) 객체만 만듭니다. 불변이라는 것은 내부의 원소를 변경할 수 없다는 의미입니다. 대신 같은 형태의 인덱스 클래스를 전체를 대체하는 것은 가능합니다.

새로운 Index 클래스로 정수 인덱스를 만들어서 특정 원소를 변경하면 예외가 발생합니다. 또한 다른 인덱스를 만들어서 같은 변수에 할당하면 나중에 할당된 객체의 정보만 관리하는 것을 볼 수 있습니다.


```
In :  idx4 = pd.Index([4,5,6,7])

In :  try :
          idx4[0] = 100
      except Exception as e :
          print(e)

Out:  Index does not support mutable operations

In :  idx4 = pd.Index([1,2,3,4])

In :  idx4

Out:  Int64Index([1, 2, 3, 4], dtype='int64')
```

[예제 4-3] 숫자와 문자 인덱스 메소드 처리하기

두 개의 정수 Index 클래스의 객체를 만듭니다. 하나의 객체에는 포함되지만 다른 객체에 없는 특정 레이블을 찾기 위해 집합연산을 제공합니다.

집합연산을 통해 나온 결과도 새로운 인덱스 객체로 반환합니다. 판다스의 메소드 처리 기준은 새로운 객체를 만들어서 반환하는 방식인데, 이 방식이 거의 같습니다.

```
In :  idx1

Out:  Int64Index([1, 2, 3, 4], dtype='int64')

In :  idx2

Out:  RangeIndex(start=1, stop=4, step=1)

In :  idx1.difference(idx2)

Out:  Int64Index([4], dtype='int64')
```

정수 인덱스에 덧셈 연산자로 3을 더하면 인덱스 객체의 원소들 값이 모두 3씩 증가한 것을 볼 수 있습니다. 인덱스 클래스의 객체는 불변이라 새로운 객체가 반환됩니다.

시리즈나 데이터 프레임처럼 덧셈 연산자와 매칭되는 add 메소드가 있는지를 확인하면 속성이 없다고 표시합니다.

```
In :  idx1 + 3

Out:  Int64Index([4, 5, 6, 7], dtype='int64')

In :  try :
          idx1.add(3)

      except Exception as e :
          print(e)

Out:  'Int64Index' object has no attribute 'add'
```

4.1.3 숫자와 문자 인덱스의 암묵적 처리

시리즈나 데이터 프레임의 원소에 대한 연산이나 메소드를 사용할 때는 레이블을 이용하여 원소에 접근합니다. 하나가 아닌 여러 개의 같은 레이블을 가질 경우 이 레이블들을 곱한 만큼의 개수가 생깁니다. 레이블이 생겼으므로 해당되는 값들도 들어갑니다.

또한 두 개의 시리즈가 연산 처리될 때 레이블이 한 시리즈만 있으면 다른 시리즈의 레이블이 생기지만, 값은 NaN으로 채워지므로 연산의 결과도 NaN으로 할당됩니다. 이런 암묵적 처리를 예제로 알아봅니다.

■숫자와 문자 인덱스 클래스의 암묵적 처리

숫자와 문자 인덱스를 가지고 연산 및 메소드 처리할 때 인덱스에 대한 정보를 가지고 암묵적인 처리 기준을 확인합니다.

[예제 4-4] 암묵적 인덱스 변경 및 암묵적 NaN 처리하기

먼저, 문자열 레이블을 갖는 두 개의 시리즈를 만듭니다. 두 개의 시리즈는 레이블을 구성하는 문자의 개수는 같지만 순서는 다릅니다.

```
In :  import numpy as np
```



```
In :  s1 = pd.Series(index=list('aaab'), data=np.arange(4))

In :  s1

Out:  a   0
      a   1
      a   2
      b   3
      dtype: int32

In :  s2 = pd.Series(index=list('baaa'), data=np.arange(4))

In :  s2

Out:  b   0
      a   1
      a   2
      a   3
      dtype: int32
```

두 개의 시리즈를 더하면 두 개의 레이블이 달라 시리즈 안의 레이블 개수끼리 곱한 만큼 카테시언 프로덕트(Cartesian product)를 처리한 레이블이 생깁니다.

```
In :  s1 + s2

Out:  a   1
      a   2
      a   3
      a   2
      a   3
      a   4
      a   3
      a   4
      a   5
      b   3
      dtype: int32
```

시리즈의 행 인덱스를 .sort_index 메소드로 정렬(alignment)해 레이블의 위치를 똑같이 만든 후에 .add 메소드를 사용하면 앞의 결과와 다르게 같은 레이블에 더한 결과를 볼 수 있습니다.

```
In :  s1.sort_index().add(s2.sort_index())
```

```
Out:  a   1
      a   3
      a   5
      b   3
      dtype: int32
```

첫 번째 만든 시리즈와 같은 행의 레이블을 가진 또 하나의 시리즈를 만들어서 덧셈 연산자로 더하면 레이블이 같으므로 레이블 기준으로 원소를 더합니다.

```
In :  s3 = pd.Series(index=list('aaab'), data=np.arange(4))

In :  s1 + s3

Out:  a   0
      a   2
      a   4
      b   6
      dtype: int32
```

이번에는 첫 번째 만든 시리즈와 4개의 레이블은 순서가 같지만 레이블 하나가 추가된 새로운 시리즈를 만듭니다. 이를 가지고 첫 번째 시리즈에 더할 때 레이블이 여러 개가 있으므로 매치되지 않아 카티션 프로덕트를 한 만큼 레이블이 생기고 값들도 들어가 있습니다.

특히 레이블 c는 첫 번째 시리즈에 존재하지 않으므로 결과 값이 NaN으로 처리됩니다.

```
In :  s4 = pd.Series(index=list('aaabc'), data=np.arange(5))

In :  s1 + s4

Out:  a   0.0
      a   1.0
      a   2.0
      a   1.0
      a   2.0
      a   3.0
      a   2.0
      a   3.0
      a   4.0
      b   6.0
      c   NaN
      dtype: float64
```


4.2 날짜 및 범주형 인덱스 처리

날짜 정보와 같은 인덱스의 레이블을 만든 데이터를 시계열(time series) 데이터라고 합니다. 실제 데이터를 분석하려면 많은 데이터가 특정 날짜와 시간으로 관리되어 처리하는 경우가 많은데, 특정 값들로 한정되어 사용되는 경우를 범주형 데이터라고 합니다. 이런 데이터를 인덱스로 사용도 가능합니다. 이런 연속적인 날짜 및 범주의 정보를 인덱스로 생성하고 메소드로 다루는 방법을 알아봅니다.

4.2.1 날짜 인덱스 처리

날짜를 연속적으로 만들어서 하나의 인덱스로 만들 수 있습니다. 이런 특정 날짜들이 연속된 레이블이 인덱스로 구성된 데이터를 시계열 데이터라고 앞서 잠깐 언급했는데, 실제 금융기관이나 대기업 등에서는 시계열 데이터들을 많이 보유하며 이를 분석해서 다양한 용도로 사용하고 있습니다. 여기서는 날짜 인덱스를 만들고 이를 이용하는 방법을 알아봅니다.

■ 날짜 인덱스

먼저 날짜에 대한 인덱스를 생성해보고 이에 대한 속성을 알아봅니다.

[예제 4-5] 날짜 인덱스 생성 및 속성 보기

Index 클래스로 날짜 인덱스를 생성할 수 있습니다. 이때 pd.date_range 함수를 이용해서 날짜 정보를 만들고 인자로 전달합니다.

만들어진 날짜 인덱스 객체의 .dtype 속성을 보면 datetime64라는 것을 알 수 있습니다. 이 데이터가 만들어질 때 특정 일자와 기간을 지정해서 내부적으로는 3일 간의 날짜가 만들어진 것을 볼 수 있는데, 이 기준이 날짜가 된 것을 .freq=D 속성으로 확인합니다. 알파벳 D는 일 단위 주기를 표시합니다.

```
In :  idx_d = pd.Index(pd.date_range('20130101',periods=3))

In :  idx_d

Out:  DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03'], dtype='datetime64[ns]', freq='D')

In :  idx_d.values
```

```
Out:  array(['2013-01-01T00:00:00.000000000', '2013-01-02T00:00:00.000000000',
             '2013-01-03T00:00:00.000000000'], dtype='datetime64[ns]')
```

연속된 날짜를 만들었는데 빈도가 일 단위인지의 이유를 확인하기 위해 .freq 속성을 조회할 수도 있습니다.

```
In :  idx_d.freq

Out:  <Day>
```

Pd.DatetimeIndex 클래스 생성자를 이용해서 직접 날짜 인덱스를 만들어봅니다.

시리즈의 객체를 생성하는 날짜를 원소로 하는 리스트를 pd.DatetimeIndex에 넣어 날짜를 관리하는 객체를 생성합니다.

시리즈 생성자에 숫자 리스트를 넣고 index 매개변수에 날짜 인덱스를 만든 것을 넣습니다. 생성된 결과를 조회하면 .index 속성의 값으로 날짜 인덱스 정보가 들어간 것을 볼 수 있습니다.

```
In :  index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                '2015-07-04', '2015-08-04'])

In :  data = pd.Series([0, 1, 2, 3], index=index)

In :  data

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

변수 index를 조회하면 날짜 정보가 들어가 있지만 이 날짜가 연속된 것이 아니므로 .freq 속성에 값이 없습니다. 배열로 구성되므로 모양과 자료형을 확인할 수도 있습니다.

```
In :  index

Out:  DatetimeIndex(['2014-07-04', '2014-08-04', '2015-07-04', '2015-08-04'], dtype='datetime64[ns]', freq=None)
```



```
In :  index.shape

Out:  (4,)

In :  index.dtype

Out:  dtype('<M8[ns]')
```

[예제 4-6] 타임에 대한 빈도 이해하기

이번에는 특정 일자가 생성될 때 특정 주기를 가지고 만들어지는 객체에 대해 알아보겠습니다. 일단 data_range 함수를 이용해서 일자를 두 개 지정하면, 두 일자 사이의 차이에 따른 날짜들이 원소를 이루는 DatetimeIndex에서 하나의 객체가 만들어집니다.

생성된 객체를 보면 8개의 일자가 있고 그 일자들은 하루 간격으로 해서 만들어진 것을 알 수 있습니다.

```
In :  dr1 = pd.date_range('2018-07-03', '2018-07-10')

In :  dr1

Out:  DatetimeIndex(['2018-07-03', '2018-07-04', '2018-07-05', '2018-07-06',
                     '2018-07-07', '2018-07-08', '2018-07-09', '2018-07-10'],
                    dtype='datetime64[ns]', freq='D')
```

하나의 일자를 지정하고 periods 매개변수에 8를 넣고 실행해도 위해서 생성된 것과 같은 객체가 만들어집니다.

특정 주기에 대한 처리 방식이 일자일 때는 두 가지 경우가 같은 방식입니다.

```
In :  dr2 = pd.date_range('2018-07-03', periods=8)

In :  dr2

Out:  DatetimeIndex(['2018-07-03', '2018-07-04', '2018-07-05', '2018-07-06',
                     '2018-07-07', '2018-07-08', '2018-07-09', '2018-07-10'],
                    dtype='datetime64[ns]', freq='D')
```

이번에는 특정 주기를 배정해서 날짜, 시간 등 다양한 방식으로 날짜에 관한 객체를 생성하는 방법을 알아보겠습니다.

기호	시간 기준	기호	시간 기준
D	달력상 일	B	영업일
W	주	M	월말
BM :	영업기준 월말	Q	분기말
BQ :	영업기준 분기말	A	연말
BA	영업기준 연말	H	시간
BH	영업시간	T	분
S	초	L	밀리초
N	나노초		

[표 4-1] 시간 주기를 나타내는 항목과 해당 기호

함수 data_range 안의 매개변수 freq에 [표 4-1]의 주기를 나타내는 항목을 넣어서 객체를 생성하면 그 기준에 따라 만들어집니다.

이번에는 특정 시간을 기준으로 만들어봅니다. 특정 일자를 넣고 periods에 24를 넣은 후 freq 속성에 H를 넣으면 이 특정 일자를 기준으로 24시간의 범위로 생성됩니다.

```
In :  dr3 = pd.date_range('2018-08-03', periods=24, freq='H')

In :  dr3

Out:  DatetimeIndex(['2018-08-03 00:00:00', '2018-08-03 01:00:00',
                     '2018-08-03 02:00:00', '2018-08-03 03:00:00',
                     '2018-08-03 04:00:00', '2018-08-03 05:00:00',
                     '2018-08-03 06:00:00', '2018-08-03 07:00:00',
                     '2018-08-03 08:00:00', '2018-08-03 09:00:00',
                     '2018-08-03 10:00:00', '2018-08-03 11:00:00',
                     '2018-08-03 12:00:00', '2018-08-03 13:00:00',
                     '2018-08-03 14:00:00', '2018-08-03 15:00:00',
                     '2018-08-03 16:00:00', '2018-08-03 17:00:00',
                     '2018-08-03 18:00:00', '2018-08-03 19:00:00',
                     '2018-08-03 20:00:00', '2018-08-03 21:00:00',
                     '2018-08-03 22:00:00', '2018-08-03 23:00:00'],
                    dtype='datetime64[ns]', freq='H')
```

데이터에 대해 만들어진 객체를 주기를 나타내는 인덱스로 바꾸기 위해 to_period 메소드를 이용해서 일자 기준으로 전환합니다. 그러면 현재 시간을 기준으로 만들어진 객체를 일자로

변형했으므로 같은 일자만 나오는 것을 볼 수 있습니다.

또한 주기를 나타내는 인덱스 클래스가 PeriodIndex로 변경되었습니다.

```
In :  dr3.to_period('D')

Out:  PeriodIndex(['2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03',
                   '2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03',
                   '2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03',
                   '2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03',
                   '2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03',
                   '2018-08-03', '2018-08-03', '2018-08-03', '2018-08-03'],
                  dtype='period[D]', freq='D')
```

주기를 만드는 함수인 period_range에 특정 달을 넣고 periods 매개변수에 13, freq 매개변수에 M을 넣었습니다. 달을 기준으로 주기를 가지고 계산되어서 2019년 1월까지 인덱스를 만듭니다. 이 함수로 만들어진 객체의 클래스는 주기를 나타내는 PeriodIndex라는 것을 알 수 있습니다.

```
In :  dr_m = pd.period_range('2018-01', periods=13, freq='M')

In :  dr_m

Out:  PeriodIndex(['2018-01', '2018-02', '2018-03', '2018-04', '2018-05', '2018-06',
                   '2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-12',
                   '2019-01'],
                  dtype='period[M]', freq='M')
```

특정 시간의 간격을 만들기 위해 이번에는 timedelta_range 함수로 특정 시간대부터 주기를 periods=10으로 주고 freq=H를 지정해서 시간별로 만들도록 했습니다.

만들어진 객체를 확인하면 00시부터 09시까지의 총 10개의 데이터가 만들어집니다. 시간을 기준으로 만들어, 특정 시간을 기준으로 timedelta64 자료형이 됩니다.

이 인덱스에 대한 클래스는 TimedeltaIndex의 객체가 만들어진 것을 알 수 있습니다.

```
In :  tm_1 = pd.timedelta_range(0, periods=10, freq='H')
```

```
In :  tm_1
```

```
Out:  TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                      '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
                     dtype='timedelta64[ns]', freq='H')
```

시간과 시간 사이의 계산을 하면 연산이 됩니다. 일단 한 시간을 빼면 배열이므로 벡터화(vectorizing) 연산이 되어 전체가 다 바뀌는 것을 확인할 수 있습니다.

timedelta64 자료형은 기본으로 시간에 대한 연산이 발생하면 그 차이를 관리하는 자료형인 것을 알 수 있습니다.

```
In :  tm_1 - tm_1[1]
```

```
Out:  TimedeltaIndex(['-1 days +23:00:00',          '00:00:00',          '01:00:00',
                      '02:00:00',          '03:00:00',          '04:00:00',
                      '05:00:00',          '06:00:00',          '07:00:00',
                      '08:00:00'],
                     dtype='timedelta64[ns]', freq='H')
```

또한 날짜로 생성된 DatetimeIndex 객체에 대해 수학 연산을 하면 그 결과에 대한 차이를 관리하는 TimedeltaIndex가 생성됩니다.

```
In :  dr2
```

```
Out:  DatetimeIndex(['2018-07-03', '2018-07-04', '2018-07-05', '2018-07-06',
                     '2018-07-07', '2018-07-08', '2018-07-09', '2018-07-10'],
                    dtype='datetime64[ns]', freq='D')
```

```
In :  dr2 - dr2[0]
```

```
Out:  TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days', '5 days',
                      '6 days', '7 days'],
                     dtype='timedelta64[ns]', freq=None)
```

시간에 대한 인덱스를 분 단위로 지정할 수도 있습니다. 이때는 분 단위로 생성하고 이를 시리즈 행의 레이블로 지정하면 됩니다. 날짜 인덱스는 date_range 메소드 안의 freq=T를 넣어서 실행하여 만들어지는데, 이를 인덱스로 넣고 시리즈를 생성합니다.


```
In :  index = pd.date_range('1/1/2018', periods=9, freq='T')
```

```
In :  series = pd.Series(range(9), index=index)
```

위에서 만들어진 series를 확인하면 행의 레이블이 분 단위로 만들어져 있는 것을 볼 수 있습니다.

```
In :  series
```

```
Out:  2018-01-01 00:00:00    0
      2018-01-01 00:01:00    1
      2018-01-01 00:02:00    2
      2018-01-01 00:03:00    3
      2018-01-01 00:04:00    4
      2018-01-01 00:05:00    5
      2018-01-01 00:06:00    6
      2018-01-01 00:07:00    7
      2018-01-01 00:08:00    8
      Freq: T, dtype: int64
```

1분 단위로 생성을 한 것을 그룹화(groupby)해서 시리즈 안에 만들어진 것을 특정 시간별로 묶어서 처리도 가능합니다.

일단 분 단위로 만들어진 시리즈를 3분 단위로 처리하기 위해 resample 메소드로 다른 시리즈로 만들었습니다.

다시 만들어진 객체를 조회하면 DateTimeIndexResampler 클래스의 객체가 만들어진 것을 볼 수 있습니다. 현재 객체만 만들어진 것이지 실제 실행한 결과를 갖는 것은 아닙니다.

```
In :  s_3T = series.resample('3T')
```

```
In :  s_3T
```

```
Out:  DatetimeIndexResampler [freq=<3 * Minutes>, axis=0, closed=left, label=left, convention=start, base=0]
```

위의 객체를 가지고 실제 합산을 하기 위해 .sum 메소드를 사용하면 계산된 결과는 3분 단위의 합산한 결과를 보여줍니다.

```
In :  s_3T.sum()
```

```
Out:  2018-01-01 00:00:00     3
      2018-01-01 00:03:00    12
      2018-01-01 00:06:00    21
      Freq: 3T, dtype: int64
```

4.2.2 날짜 인덱스 주요 처리 방법

판다스의 클래스에 행의 레이블로 날짜을 붙이는 것은 관측된 결과가 시계열(time series)로 처리가 되는 곳에 매우 유용합니다. 보통 파일로 데이터를 읽어오면 날짜로 구성된 것처럼 보이지만 일반 문자열인 경우가 많습니다.

이런 문자열이 인덱스 처리가 되는 경우 날짜 인덱스로 변환해서 처리해야 합니다.

■ 날짜 인덱스 활용

시계열(time series) 데이터 등을 사용하기 위해서는 날짜가 인덱스로 처리되어야 합니다. 날짜 인덱스로 다양한 함수나 메소드에서 처리가 가능하므로 기본적인 날짜 처리에 대해 알아봅니다.

[예제 4-7] 날짜 인덱스 활용하기

하나의 csv 파일을 읽을 때 parse_dates를 추가해 날짜를 인덱스로 처리합니다.

```
In :  data = pd.read_csv('../data/hanriver_bridge.csv', index_col='Date', parse_dates=True,encoding='cp949')
```

```
In :  data.head()
```

Out:

Date	한강 좌측 인도	한강 우측 인도
2012-10-03 00:00:00	4.0	9.0
2012-10-03 01:00:00	4.0	6.0
2012-10-03 02:00:00	1.0	1.0
2012-10-03 03:00:00	2.0	3.0
2012-10-03 04:00:00	6.0	1.0

파일에 대한 행과 열의 인덱스를 확인합니다. 행에 인덱스는 날짜로 되어 있는 것을 알 수 있습니다.

```
In :  data.columns
```

```
Out:  Index(['한강 좌측 인도', '한강 우측 인도'], dtype='object')
```

```
In :  data.index
```

```
Out:  DatetimeIndex(['2012-10-03 00:00:00', '2012-10-03 01:00:00',
                     '2012-10-03 02:00:00', '2012-10-03 03:00:00',
                     '2012-10-03 04:00:00', '2012-10-03 05:00:00',
                     '2012-10-03 06:00:00', '2012-10-03 07:00:00',
                     '2012-10-03 08:00:00', '2012-10-03 09:00:00',
                     ...
                     '2018-05-31 14:00:00', '2018-05-31 15:00:00',
                     '2018-05-31 16:00:00', '2018-05-31 17:00:00',
                     '2018-05-31 18:00:00', '2018-05-31 19:00:00',
                     '2018-05-31 20:00:00', '2018-05-31 21:00:00',
                     '2018-05-31 22:00:00', '2018-05-31 23:00:00'],
                    dtype='datetime64[ns]', name='Date', length=49608, freq=None)
```

열의 이름에 빈 공간이 있으므로 단순하게 열 이름을 변경합니다. 실제 연산을 문자열로 만들어서 eval 메소드로 실행하면 새로운 합산 열에 추가됩니다.

```
In :  data.columns=['좌측','우측']
```

```
In :  data['합산'] = data.eval('좌측 + 우측')
```

```
In :  data.head()
```

Out:

Date	좌측	우측	합산
2012-10-03 00:00:00	4.0	9.0	13.0
2012-10-03 01:00:00	4.0	6.0	10.0
2012-10-03 02:00:00	1.0	1.0	2.0
2012-10-03 03:00:00	2.0	3.0	5.0
2012-10-03 04:00:00	6.0	1.0	7.0

누락 값을 삭제하기 위해 누락 값을 .isnull과 .sum 메소드로 찾으면 8개가 나옵니다. 이를 .dropna 메소드로 누락 값을 일단 삭제합니다.

```
In : data.isnull().sum()
```

```
Out: 좌측    8
     우측    8
     합산    8
     dtype: int64
```

```
In : data.shape
```

```
Out: (49608, 3)
```

```
In : data_dp = data.dropna()
```

```
In : data_dp.shape
```

```
Out: (49600, 3)
```

특정 날짜를 기준으로 다시 샘플링해 .sum 메소드를 하고 처리하면 실제 데이터 프레임 안 행의 레이블이 전부 일자별로 변경되어 처리되는 것을 알 수 있습니다.

```
In : daily = data.resample('D').sum()
```

```
In : daily.head()
```

Out:

Date	좌측	우측	합산
2012-10-03	1760.0	1761.0	3521.0
2012-10-04	1708.0	1767.0	3475.0
2012-10-05	1558.0	1590.0	3148.0
2012-10-06	1080.0	926.0	2006.0
2012-10-07	1191.0	951.0	2142.0

기존 데이터를 다시 주 단위 리샘플링하기 위해 resample 메소드에 W를 넣고 평균을 계산하면 새로운 데이터 프레임이 만들어집니다. 행 인덱스가 7일 단위로 구성되는 것을 알 수 있습니다.


```
In : weekly_r = data.resample('W').mean()
```

```
In : weekly_r.head()
```

Out:

Date	좌측	우측	합산
2012-10-07	60.808333	58.291667	119.100000
2012-10-14	51.660714	48.309524	99.970238
2012-10-21	47.297619	45.017857	92.315476
2012-10-28	41.077381	38.904762	79.982143
2012-11-04	38.142857	34.440476	72.583333

기존 데이터 프레임에서 freq를 W로 바꾸려면 asfreq 메소드를 이용해 처리하면 됩니다. 이때는 특정 날짜에 있는 행의 데이터로 세팅이 되고 나머지 데이터는 사라집니다.

리샘플링 메소드를 이용하면 날짜별로 값을 다양한 기준으로 합산하거나 평균을 내어 처리할 수 있습니다.

```
In : weekly_f = data.asfreq('W')
```

```
In : weekly_f.head()
```

Out:

Date	좌측	우측	합산
2012-10-07	6.0	5.0	11.0
2012-10-14	3.0	3.0	6.0
2012-10-21	5.0	12.0	17.0
2012-10-28	5.0	5.0	10.0
2012-11-04	7.0	11.0	18.0

4.2.3 범주형 인덱스 주요 처리 방법

범주형 인덱스를 만드는 CategoricalIndex 클래스도 별도로 제공하고 있습니다. 인덱스의 레이블을 범주형으로 관리하면 숫자나 문자열일 때와 어떻게 다른지를 알아봅니다.

■범주형 인덱스 생성 및 변경

범주형 인덱스를 만들어서 시리즈에 적용하고 레이블이 추가적으로 필요할 때 처리 방법을 알아봅니다.

[예제 4-8] 범주형 인덱스 활용하기

범주형 인덱스는 CategoricalIndex 클래스를 이용해서 정수 리스트를 인자로 받아 인덱스 내의 레이블이 한정된 정수값을 갖습니다.

```
In :  inx_i = pd.CategoricalIndex([1,2,3,4])
```

하나의 시리즈를 만들 때 범주형 인덱스를 넣어서 생성합니다. 정수의 값이므로 실제 정수 인덱스와 같아 보입니다.

.index 속성을 확인해보면 범주형 인덱스의 정보를 명확히 구분할 수 있습니다.

```
In :  s = pd.Series([1,2,3,4],index=inx_i)

In :  s
```

```
Out:  1    1
      2    2
      3    3
      4    4
      dtype: int64
```

```
In :  s.index
```

```
Out:  CategoricalIndex([1, 2, 3, 4], categories=[1, 2, 3, 4], ordered=False, dtype='category')
```

시리즈 내의 특정 원소를 갱신하기 위해 레이블을 지정해서 처리하면 범주형 레이블이 아닌 포지션 레이블이 알려주는 값이 갱신된 것을 알 수 있습니다.

```
In :  s[3] = 100

In :  s
```



```
Out:  1      1
      2      2
      3      3
      4    100
      dtype: int64
```

시리즈에서 레이블의 범위를 벗어나는 객체를 추가하려면 예외가 발생합니다.

```
In :  try :
          s[5] = 100
      except Exception as e :
          print(e)
```

```
Out:  index 5 is out of bounds for axis 0 with size 4
```

범주형 인덱스를 추가한 후에 인덱스 레이블을 추가해도 예외가 발생합니다. 범주형 인덱스로 처리하면 처음 생성된 기준을 유지하므로 실제 레이블 값이 변경되지 않습니다.

```
In :  s.index = s.index.add_categories(5)

In :  s.index
```

```
Out:  CategoricalIndex([1, 2, 3, 4], categories=[1, 2, 3, 4, 5], ordered=False, dtype='category')
```

```
In :  try :
          s.index = s.index.insert(4,5)
      except Exception as e :
          print(e)
```

```
Out:  Length mismatch: Expected axis has 4 elements, new values have 5 elements
```

시리즈의 레이블을 추가하기 위해 범주형 인덱스를 리스트로 변환하고 이를 시리즈에 .index 속성에 할당해서 정수형 인덱스로 변환합니다.

정수형일 경우는 인덱스의 레이블이 추가되므로 인덱스 검색에 따른 값을 할당합니다.

```
In :  stl = s.index.tolist()
```

```
In :  stl

Out:  [1, 2, 3, 4]

In :  s.index = stl

In :  s.index

Out:  Int64Index([1, 2, 3, 4], dtype='int64')

In :  s[5] = 100

In :  s

Out:  1      1
      2      2
      3      3
      4    100
      5    100
      dtype: int64
```

시리즈의 행을 추가하고 다시 .index 속성을 .astype 메소드로 자료형을 변환하면 범주형 인덱스가 됩니다.

```
In :  s.index = s.index.astype('category')

In :  s

Out:  1      1
      2      2
      3      3
      4    100
      5    100
      dtype: int64

In :  s.index

Out:  CategoricalIndex([1, 2, 3, 4, 5], categories=[1, 2, 3, 4, 5], ordered=False, dtype='category')
```


4.3 멀티인덱스 처리

시리즈와 데이터 프레임을 사용하면 1차원과 2차원 처리만 가능하지만 다차원 처리가 필요한 경우 시리즈와 데이터 프레임을 어떻게 처리해야 할까요? 그 답은 계층적인 레이블을 만들어서 시리즈나 데이터 프레임의 차원을 확대하는 것입니다.

판다스에서는 계층적인 레이블을 만드는 MultiIndex 클래스를 제공합니다. 이 클래스를 이용해서 인덱스를 생성하는 방법을 먼저 알아봅니다. 또한 만들어진 계층적 레이블로 시리즈나 데이터 프레임의 행과 열에 접근하는 방법을 살펴봅니다.

4.3.1 멀티인덱스 생성

멀티인덱스(MultiIndex) 클래스에서 해당 객체를 만들기 위해 다양한 데이터 구조의 정보를 가져와서 처리하는 함수들을 알아봅니다.

■ 멀티인덱스 생성

멀티인덱스를 만들기 전에 인덱스를 튜플로 만들어서 처리해보면 멀티인덱스가 왜 필요한지를 이해할 수 있습니다. 멀티인덱스를 만들어서 레이블이 어떻게 붙어서 작용하는지를 알아봅니다.

[예제 4-9] 멀티인덱스 생성하기

2개를 원소 쌍으로 하는 튜플을 두 개 만들고 리스트에 넣습니다. 시리즈를 만들 때 매개변수 index에 이 리스트를 지정했습니다.

만들어진 시리즈 객체를 보면 튜플로 된 인덱스가 만들어진 것을 볼 수 있습니다.

.index 속성을 확인하면 계층적 인덱스가 아닌 단일 인덱스인 Index 클래스의 객체가 조회됩니다.

```
In :  import numpy as np

In :  ind = [("서울", 2017), ("경기도",2017)]
```

```
In :  si = pd.Series(np.random.randint(1,10,2),index=ind)

In :  si

Out:  (서울, 2017)      9
      (경기도, 2017)    3
      dtype: int32

In :  si.index

Out:  Index([('서울', 2017), ('경기도', 2017)], dtype='object')
```

Index 클래스의 객체는 단일 레이블이므로 튜플 단위로 검색을 처리해야 합니다. 이를 분리해서 검색에 사용하면 예외를 발생시킵니다.

```
In :  try :
          si['서울']
      except Exception as e :
          print(e)

Out:  '서울'

In :  si['서울',2017]

Out:  9

In :  si[('서울',2017)]

Out:  9
```

위에서 만들어진 ind 변수를 이용해서 이번에는 계층적 레이블을 만들기 위해 멀티인덱스 클래스에 있는 .from_tuples 함수를 이용합니다.

멀티인덱스에 만들어진 index 변수를 확인하면 클래스가 MultiIndex라는 것을 알 수 있습니다.

들어간 레이블의 정보는 .levels와 .labels 두 개의 속성에 나눠 관리됩니다. .labels 속성에 들어간 숫자는 .levels 속성에 있는 레이블 이름의 위치를 가리키는 포지션 정보입니다.


```
In :  index = pd.MultiIndex.from_tuples(ind)

In :  index

Out:  MultiIndex(levels=[['경기도', '서울'], [2017]],
                 labels=[[1, 0], [0, 0]])
```

이 멀티인덱스를 갖는 시리즈 객체를 생성하기 위해 매개변수 index에 위에서 만들어진 index를 지정했습니다.

시리즈가 만들어진 것을 조회하면 행의 인덱스 레이블이 튜플로 들어가지 않고 개별적인 이름으로 들어간 것을 볼 수 있습니다.

```
In :  s = pd.Series(np.random.randint(1,10,2),index=index)

In :  s

Out:  서울   2017      9
      경기도  2017      5
      dtype: int32
```

멀티인덱스로 만들어진 레이블은 인덱스를 검색할 때 계층별로 분리해서 검색에 사용할 수 있습니다. 먼저 첫 번째 계층으로 조회하면 결과는 다음 계층의 레이블과 값으로 구성된 시리즈를 반환합니다.

첫 번째 계층 레이블과 두 번째 계층 레이블을 다 넣고 조회하거나 이를 튜플로 넣어 조회하면 이 레이블에 매칭되는 스칼라 값이 조회됩니다.

```
In :  s['서울']

Out:  2017    9
      dtype: int32

In :  s['서울',2017]

Out:  9

In :  s[('서울',2017)]
```

```
Out:  9
```

또 인덱서의 .loc 속성을 이용해서 검색을 할 때에도 계층별 레이블을 이용하면 됩니다.

시리즈는 행을 중심으로 처리하므로 처리되는 결과가 인덱스 검색과 인덱서 검색이 동일한 것을 알 수 있습니다.

```
In :  s.loc['서울']
```

```
Out:  2017      9
      dtype: int32
```

```
In :  s[:,2017]
```

```
Out:  서울      9
      경기도    5
      dtype: int32
```

```
In :  s.loc[:,2017]
```

```
Out:  서울      9
      경기도    5
      dtype: int32
```

넘파이 배열을 두 개 지정하고 리스트의 원소로 넣어서 리스트를 만듭니다.

```
In :  import numpy as np
```

```
In :  arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
               np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
```

시리즈를 만들 때 매개변수 index에 arrays 변수를 지정했습니다. 만들어진 시리즈를 검색하면 멀티인덱스가 만들어진 것을 볼 수 있습니다.

```
In :  s = pd.Series(np.random.randn(8), index=arrays)
```

```
In :  s
```



```
Out:  bar  one   -0.176555
           two    0.072935
      baz  one    0.747105
           two   -1.226154
      foo  one    1.087123
           two    0.072995
      qux  one   -0.258387
           two    0.720419
      dtype: float64
```

만들어진 시리즈의 .index 속성을 조회하면 .levels와 .lables가 구성된 것을 알 수 있습니다. 리스트에 넣어진 넘파이 배열의 위치에 따라 levels의 포지션을 차지해서 구성되었습니다.

```
In :  s.index
```

```
Out:  MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
                 labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]])
```

파이썬 리스트에 리스트를 넣어서 멀티인덱스를 만들려면 MultiIndex 클래스의 .from_product 함수로 생성하면 됩니다.

이번에는 두 개의 계층적 레이블에 이름도 부여했습니다.

```
In :  iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]
```

```
In :  arrays2 = pd.MultiIndex.from_product(iterables, names=['first', 'second'])
```

생성한 멀티인덱스의 객체를 조회하면 .levels, .labels, .names 속성에 값들이 지정한 대로 들어가 있는 것을 볼 수 있습니다.

```
In :  arrays2
```

```
Out:  MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
                 labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
                 names=['first', 'second'])
```

생성된 속성을 하나하나 조회해보면 인덱스 정보는 바꿀 수 없다고 배웠는데 왜 갱신이 안 되었는지를 깨닫게 됩니다.

.levels, .labels 속성에 FrozenList 클래스의 인스턴스가 할당되어 있어 변경이 불가능한 리스트로 구성된 것을 알 수 있습니다.

내부의 값 중 .values 속성을 조회하면 넘파이 배열의 원소들이 튜플로 구성되어 있습니다. 각 계층별 이름도 바꿀 수 없는 객체에 값이 들어 있습니다.

```
In :  arrays2.levels

Out:  FrozenList([['bar', 'baz', 'foo', 'qux'], ['one', 'two']])

In :  arrays2.labels

Out:  FrozenList([[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]])

In :  arrays2.values

Out:  array([('bar', 'one'), ('bar', 'two'), ('baz', 'one'), ('baz', 'two'),
             ('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')],
            dtype=object)

In :  arrays2.names

Out:  FrozenList(['first', 'second'])
```

계층적 인덱스를 만든 객체를 시리즈의 매개변수 index에 넣어 생성자를 실행하면 계층적 인덱스가 만들어집니다. 이번에는 계층적 인덱스에서 이름이 맨 위에 표시되는 것도 볼 수 있습니다.

```
In :  s2 = pd.Series(np.random.randn(8), index=arrays2)

In :  s2

Out:  first  second
      bar    one       1.399236
             two       0.566254
      baz    one       0.921560
             two       0.822533
      foo    one      -0.974971
             two      -0.647916
      qux    one      -0.270709
             two      -0.382016
      dtype: float64
```


멀티인덱스에서 객체의 값을 보관하는 .values를 보면 실제 튜플로 관리하므로 튜플을 원소로 가진 리스트가 만들어질 때 멀티인덱스의 .from_tuples 함수를 이용합니다. 계층적 인덱스의 특별한 위치를 지정하지 않아서 생성될 때 레이블의 정렬을 먼저 수행해서 처리된 것을 알 수 있습니다.

만들어진 인덱스를 확인하면 튜플의 첫 번째 원소들이 .levels 속성의 첫 번째 level에 들어가고 튜플의 두 번째 원소들이 .levels 속성의 두 번째 level로 들어갑니다.

.labels 속성을 확인하면 첫 번째 리스트는 튜플의 첫 번째 값들의 순서를 나타냅니다.

```
In :  index = [('서울',2008),('서울',2010),('부산',2008),('부산',2010), ('인천',2008),('인천',2010)]

In :  mul_index = pd.MultiIndex.from_tuples(index)

In :  mul_index

Out:  MultiIndex(levels=[['부산', '서울', '인천'], [2008, 2010]],
                 labels=[[1, 1, 0, 0, 2, 2], [0, 1, 0, 1, 0, 1]])
```

멀티인덱스의 객체 값을 조회하면 내부에 튜플을 지정한 값이 그대로 들어 있는 것을 확인할 수 있습니다. 이 값들의 빈도를 알아보기 위해 .value_counts를 실행합니다. 모든 레이블의 개수가 1인 것을 알 수 있습니다.

```
In :  mul_index.values

Out:  array([('서울', 2008), ('서울', 2010), ('부산', 2008), ('부산', 2010),
             ('인천', 2008), ('인천', 2010)], dtype=object)

In :  mul_index.value_counts()

Out:  (서울, 2010)    1
      (부산, 2010)    1
      (부산, 2008)    1
      (인천, 2008)    1
      (서울, 2008)    1
      (인천, 2010)    1
      dtype: int64
```

또한 .levels, .labels의 속성을 조회한 결과를 확인합니다.

```
In :  mul_index.levels

Out:  FrozenList([['부산', '서울', '인천'], [2008, 2010]])

In :  mul_index.labels

Out:  FrozenList([[1, 1, 0, 0, 2, 2], [0, 1, 0, 1, 0, 1]])
```

4.3.2 멀티인덱스 활용

단일 인덱스의 레이블로 시리즈의 값을 효율적으로 검색해보았습니다. 계층형 인덱스는 추가적인 차원을 확대하는 것이므로 검색하는 다른 기능도 제공합니다.

실제 시리즈와 데이터 프레임의 행과 열의 레이블에 대한 정보를 변형할 때 기존에 있는 열을 가지고 행의 레이블로 지정할 수 있고 행의 레이블을 열의 레이블로 지정할 수도 있습니다.

이런 활용은 '5장 데이터 재구성하기'에서 상세히 설명되므로 이번에는 멀티인덱스에 지정된 객체를 검색해서 사용하는 방법을 간략히 알아봅니다.

■멀티인덱스로 시리즈 생성

단일 인덱스와 멀티인덱스를 인덱스 검색을 통해 원소 하나 또는 부분집합의 슬라이스로 처리하는 방법을 알아봅니다.

[예제 4-10] 멀티인덱스를 활용한 시리즈 만들기

앞 절에서 만들어진 멀티인덱스 객체인 mul_index를 시리즈 생성자의 매개변수인 index에 지정했고, 실제 값은 하나의 리스트로 만들었습니다. 실제 값은 멀티인덱스의 레이블 개수와 맞게 설정했습니다.

시리즈가 만들어진 후에 조회하면 두 개의 계층을 갖는 시리즈가 만들어집니다.

```
In :  populations = [ 30000,37000, 18970, 19370, 20850, 25140]

In :  pop = pd.Series(populations, index=mul_index)
```



```
In :  pop

Out:  서울  2008    30000
      2010    37000
      부산  2008    18970
      2010    19370
      인천  2008    20850
      2010    25140
      dtype: int64
```

인덱스를 검색할 때 시리즈는 행 단위로만 처리하므로 하나의 값만 넣고 조회합니다. 하지만 멀티인덱스로 레이블이 지정되어 레이블에 접근하려면 레벨 순서로 입력해서 조회가 가능합니다.

첫 번째 레이블로 조회하면 해당되는 두 개의 값이 나오고 두 개의 레이블을 넣으면 시리즈 하나의 행과 매칭되어 하나의 값만 나옵니다.

```
In :  pop['서울']

Out:  2008    30000
      2010    37000
      dtype: int64

In :  pop['서울',2008]

Out:  30000
```

첫 번째 레이블 전체에 슬라이스 처리하고 두 번째 레이블 이름을 선택하면 첫 번째 레이블 기준으로 결과 값이 나옵니다.

```
In :  pop[:, 2010]

Out:  서울    37000
      부산    19370
      인천    25140
      dtype: int64
```

문자로 처리되는 경우는 슬라이스 처리를 하기 위해서 .sort_index 메소드로 인덱스 값의 문자열을 정렬해야 합니다.

```
In :  try :
          pop["서울" : "인천"]
      except Exception as e :
          print(e)

Out:  'Key length (1) was greater than MultiIndex lexsort depth (0)'

In :  pop = pop.sort_index()

In :  pop["서울" : "인천"]

Out:  서울   2008    30000
           2010    37000
      인천   2008    20850
           2010    25140
      dtype: int64
```

멀티인덱스에도 각 레벨에 맞는 이름을 .index.nemes 속성에 리스트로 지정해서 넣으면 레벨별로 이름이 들어갑니다.

```
In :  pop.index.names = ['시','연도']

In :  pop

Out:   시    연도
      부산   2008    18970
           2010    19370
      서울   2008    30000
           2010    37000
      인천   2008    20850
           2010    25140
      dtype: int64
```

■ 멀티인덱스를 활용한 데이터 프레임

멀티인덱스를 갖는 데이터 프레임을 만들어봅니다. 데이터 프레임에는 여러 열이 있으므로 이를 인덱스로 바꿀 때 멀티인덱스 처리도 가능합니다.

이번에는 멀티인덱스에서 각 레벨의 이름을 이용하여 데이터 프레임에 접근하고 검색하는 법을 알아봅니다.

[예제 4-11] 멀티인덱스를 이용한 데이터 프레임 활용하기

멀티인덱스는 두 개의 레벨을 연도와 정수 리스트로 표현했고 이 레벨명 이름을 '연도', '과제점수'라고 명명해서 멀티인덱스의 객체를 만들었습니다.

```
In :  r_inx = pd.MultiIndex.from_product([[2017,2018],[1,2]], names=['연도','과제점수'])

In :  r_inx

Out:  MultiIndex(levels=[[2017, 2018], [1, 2]],
                 labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
                 names=['연도', '과제점수'])
```

다른 멀티인덱스는 .from_product 함수를 이용해서 2개의 레벨을 리스트로 만들었습니다.

```
In :  c_inx = pd.MultiIndex.from_product([['철수','영희','지원'],['컴공','경제']], names=['학생','학과'])

In :  c_inx

Out:  MultiIndex(levels=[['영희', '지원', '철수'], ['경제', '컴공']],
                 labels=[[2, 2, 0, 0, 1, 1], [1, 0, 1, 0, 1, 0]],
                 names=['학생', '학과'])
```

넘파이 모듈을 이용해서 4행 6열짜리 데이터를 만듭니다.

```
In :  import numpy as np

In :  data = np.round(np.abs(np.random.randn(4,6)),1)

In :  data

Out:  array([[0.4, 0.2, 0.1, 0. , 1. , 0.5],
             [0.4, 2.1, 1.3, 1.4, 1.5, 1.4],
             [1.9, 0.3, 0.8, 1.7, 0.6, 0.2],
             [0.7, 1. , 0.7, 1.3, 0.8, 0.2]])
```

위에 만들어진 두 개의 멀티인덱스를 데이터 프레임의 행과 열의 레이블로 지정해서 데이터 프레임을 생성합니다.

```
In :  study_data = pd.DataFrame(data, index=r_inx, columns=c_inx)
```

```
In :  study_data
```

Out:

	학생	철수		영희		지원	
	학과	컴공	경제	컴공	경제	컴공	경제
연도	과제점수						
2017	1	0.4	0.2	0.1	0.0	1.0	0.5
	2	0.4	2.1	1.3	1.4	1.5	1.4
2018	1	1.9	0.3	0.8	1.7	0.6	0.2
	2	0.7	1.0	0.7	1.3	0.8	0.2

데이터 프레임 안의 .index와 .columns 속성을 확인해보면 멀티인덱스 객체로 들어가 있는 것을 알 수 있습니다.

```
In :  study_data.index
```

```
Out:  MultiIndex(levels=[[2017, 2018], [1, 2]],
                 labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
                 names=['연도', '과제점수'])
```

```
In :  study_data.columns
```

```
Out:  MultiIndex(levels=[['영희', '지원', '철수'], ['경제', '컴공']],
                 labels=[[2, 2, 0, 0, 1, 1], [1, 0, 1, 0, 1, 0]],
                 names=['학생', '학과'])
```

.index, .columns 속성도 배열이므로 레벨별로 조회하면 튜플로 결과를 보여줍니다. 레벨이 두 개여서 .names 속성에도 이름이 두 개 들어 있으므로, 프로그램에서 하나씩 조회하도록 따로 입력했습니다.

```
In :  study_data.index[0]
```

```
Out:  (2017, 1)
```



```
In :  study_data.index[1]
```

```
Out:  (2017, 2)
```

```
In :  study_data.index.names[0]
```

```
Out:  '연도'
```

```
In :  study_data.index.names[1]
```

```
Out:  '과제점수'
```

데이터 프레임 안의 data 보관을 확인하는 .values 속성을 조회합니다.

```
In :  study_data.values
```

```
Out:  array([[0.4, 0.2, 0.1, 0. , 1. , 0.5],
             [0.4, 2.1, 1.3, 1.4, 1.5, 1.4],
             [1.9, 0.3, 0.8, 1.7, 0.6, 0.2],
             [0.7, 1. , 0.7, 1.3, 0.8, 0.2]])
```

먼저 인덱스 검색을 수행하기 위해 열을 기준으로 검색해서 처리합니다. 열의 레이블에서 첫 번째 레벨의 이름을 가져오면 그 아래 레벨의 데이터 프레임이 검색되는 것을 알 수 있습니다.

```
In :  study_data['지원']
```

Out:

	학과	컴공	경제
연도	과제점수		
2017	1	1.0	0.5
	2	1.5	1.4
2018	1	0.6	0.2
	2	0.8	0.2

열의 레이블에서 두 개의 레벨을 차례로 지정하면 한 열의 정보를 가져와 시리즈로 반환합니다.

```
In :  study_data['지원','컴공']
```

```
Out:  연도    과제점수
      2017  1      1.0
            2      1.5
      2018  1      0.6
            2      0.8
      Name: (지원, 컴공), dtype: float64
```

행을 기준으로 하는 인덱서 검색에 대해 이번에는 멀티인덱스 기준으로 처리하는 법을 예제를 통해 배워봅니다.

먼저 모든 행을 전부 포함하고 열의 정보를 레벨별로 해서 튜플로 묶었습니다. 모든 행에서 특정 열을 처리한 결과이므로 인덱스 검색에서 열 단위로 처리한 것과 같은 결과가 나옵니다.

```
In :  study_data.loc[:, ('지원','컴공')]
```

```
Out:  연도    과제점수
      2017  1      1.0
            2      1.5
      2018  1      0.6
            2      0.8
      Name: (지원, 컴공), dtype: float64
```

인덱서 검색이라도 문자열을 넣고 슬라이스 검색을 할 때 문자열 레이블이 정렬되어 있지 않으면 예외가 발생합니다. 문자열 레이블일 경우에도 슬라이스 검색은 순서를 지정해서 처리됩니다. 한글일 경우에도 자음 순서대로 정렬이 필요합니다.

```
In :  try :
          study_data.loc[:2018, '철수':'영희']
      except Exception as e :
          print(e)
```

```
Out:  'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

데이터 프레임의 열의 레이블을 정렬하기 위해 .sort_index 메소드를 실행합니다. .sort_index 메소드에서는 열과 행을 전부 사용하는데, 이 경우에서는 기준으로 행을 사용하므로 axis 매개변수에 1을 넣어 열 단위 처리하는 것을 명기해야 합니다.


```
In :  study_data = health_data.sort_index(axis=1)
```

```
In :  study_data
```

Out:

		학생	철수		영희		지원	
		학과	컴공	경제	컴공	경제	컴공	경제
연도	과제점수							
2017	1	0.4	0.2	0.1	0.0	1.0	0.5	
	2	0.4	2.1	1.3	1.4	1.5	1.4	
2018	1	1.9	0.3	0.8	1.7	0.6	0.2	
	2	0.7	1.0	0.7	1.3	0.8	0.2	

인덱서를 이용해서 행은 레벨(level) 1 기준으로 열은 레벨(level) 0 기준으로 슬라이싱을 처리해서 정보를 조회합니다.

```
In :  study_data.sort_index(axis=1).loc[:2018, '영희':'지원']
```

Out:

		학생	영희		지원	
		학과	경제	컴공	경제	컴공
연도	과제점수					
2017	1	0.0	0.1	0.5	1.0	
	2	1.4	1.3	1.4	1.5	
2018	1	1.7	0.8	0.2	0.6	
	2	1.3	0.7	0.2	0.8	

또 다른 방식으로 슬라이싱을 처리하려면 IndexSlice 속성을 이용해서 대괄호([ ]) 검색을 넣고 슬라이싱을 처리하면 됩니다.

이때는 차원에 맞는 슬라이싱과 인덱스 검색을 혼용해서 모두 사용할 수 있습니다.

```
In :  study_data.loc[pd.IndexSlice[:,1], pd.IndexSlice[:, '컴공']]
```

Out:

		학생	철수	영희	지원
		학과	컴공	컴공	컴공
연도	과제점수				
2017	1	0.4	0.1	1.0	
2018	1	1.9	0.8	0.6	

멀티인덱스 검색을 위한 .xs 메소드도 있습니다. 이 메소드에는 인덱스 레이블을 넣어서 검색하므로 먼저 인덱스에 대한 자료형을 확인합니다.

```
In :  study_data.index

Out:  MultiIndex(levels=[[2017, 2018], [1, 2]],
                 labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
                 names=['연도', '과제점수'])

In :  study_data.index.levels[0].dtype

Out:  dtype('int64')

In :  study_data.index.levels[1].dtype

Out:  dtype('int64')
```

.xs 메소드에 레벨을 쌍으로 구성해서 처리하면 두 번째 레벨이 맞춰서 처리됩니다. 이 경우는 행을 기준으로 하므로 행의 멀티인덱스를 넣어서 처리했습니다.

```
In :  study_data.xs((2017,1))

Out:  학생  학과
      철수  컴공    0.4
            경제    0.2
      영희  컴공    0.1
            경제    0.0
      지원  컴공    1.0
            경제    0.5
      Name: (2017, 1), dtype: float64

In :  study_data.xs(2017)
```

Out:

학생	철수		영희		지원	
학과	컴공	경제	컴공	경제	컴공	경제
과제점수						
1	0.4	0.2	0.1	0.0	1.0	0.5
2	0.4	2.1	1.3	1.4	1.5	1.4

데이터를 열 단위로 처리하기 위해서는 열의 레이블의 자료형을 확인해서 두 개의 레벨에 맞

춰 쌍을 만들어 작성하고 반드시 축 .axis=1로 지정해서 실행해야 합니다.

```
In :  study_data.columns.levels[0].dtype

Out:  dtype('O')

In :  study_data.columns.levels[1].dtype

Out:  dtype('O')

In :  study_data.xs(('지원','컴공'), axis=1)

Out:  연도    과제점수
      2017  1       1.0
            2       1.5
      2018  1       0.6
            2       0.8
      Name: (지원, 컴공), dtype: float64
```