5. pandas 시작하기

pandas는 앞으로 가장 자주 살펴볼 라이브러리이다. 고수준의 자료구조와 파이썬에서 빠르고 쉽게 사용할 수 있는 데이터 분석 도구를 포함하고 있다. pandas는 다른 산술 계산 도구인 NumPy, Scipy, 분석 라이브러리인 statsmodels와 scikit - learn, 시각화 도구인 matplotlib 과 함께 사용하는 경우가 흔하다. pandas는 for문을 사용하지 않고 데이터를 처리한다거나 배열 기반의 함수를 제공하는 등 Numpy의 배열 기반 계산 스타일을 많이 차용했다.

pandas가 numpy의 스타일을 많이 차용했지만 가장 큰 차이점은 pandas는 표 형식의 데이터 혹은 다양한 형태의 데이터를 다루는 데 초점을 맞춰 설계했다는 점이다. numpy는 단일 산술 배열 데이터를 다루는 데 특화되어 있다. 2010년에 오픈소스로 공개한 이후 pandas는 다양한 현실의 요구 사항을 모두 수용할 수 있는 매우 큰 라이브러리가 됐다. Series와 Dataframe은 로컬 네임스페이스로 임포트하는 것이 훨씬 편하므로 그렇게 사용하도록 하겠다.

```
In [140]: import pandas as pd

In [143]: from pandas import Series, DataFrame
```

5.1 pandas 자료구조 소개

pandas에 대해 알아보려면 Series와 DataFrame, 이 두가지 자료구조에 익숙해질 필요가 있다. 이 두가지 자료구조로 모든 문제를 해결할 수 는 없지만 대부분의 어플에서 사용하기 쉬우며 탄탄한 기반을 제공한다.

5.1.1 Series

Series는 일련의 객체를 담을 수 있는 1차원 배열 같은 자료구조다(어떤 NumPy 자료형이라고 할 지라도 담을 수 있다). 그리고 **색인**이라고 하는 배열의 데이터와 연관된 이름을 가지고 있다.

가장 간단한 Series 객체는 배열 데이터로부터 생성할 수 있다.

```
In [144]: obj = pd.Series([4,7,-5,3])

In [145]: obj
Out[145]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series 객체의 문자열 표현은 왼쪽에 색인을 보여주고 오른쪽에 해당 색인의 값을 보여준다. 위 예제에서는 데이터의 색인을 지정하지 않았으니 기본 색인인 정수 0에서 N-1(N은 데이터의 길이)까지의 숫자가 표시된다. Series의 배열과 색인 객체는 각각 values와 index 속성을 통해 얻을 수 있다.

```
In [146]: obj.values
Out[146]: array([ 4,  7, -5,  3], dtype=int64)

In [147]: obj.index
Out[147]: RangeIndex(start=0, stop=4, step=1)
```

각각의 데이터를 지칭하는 색인을 지정하여 Series 객체를 생성해야 할 때는 다음처럼 한다.

```
In [148]: obj2 = pd.Series([4,7,-5,3], index=['d','b','a','c'])

In [149]: obj2
Out[149]:
d    4
b    7
a   -5
c    3
dtype: int64

In [150]: obj2.index
Out[150]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

NumPy 배열과 비교하자면, 단일 값을 선택하거나 여러 값을 선택할 때 색인으로 라벨을 사용할 수 있다.

```
In [151]: obj2['a']
Out[151]: -5

In [152]: obj2['d']=4

In [153]: obj2[['c','a','d']]
Out[153]:
c    3
a   -5
d    4
dtype: int64
```

여기서 ['c','a','d'] 는 (정수가 아니라 문자열이 포함되어 있지만) 색인의 배열로 해석된다.

불리언 배열을 사용해서 값을 걸러 내거나 산술 곱셈을 수행하거나 또는 수학 함수를 적용하는 등 NumPy 배열 연산을 수행해도 색인-값 연결이 유지된다.

```
In [154]: obj2[obj2>0]
Out[154]:
d    4
b    7
c    3
dtype: int64

In [155]: obj2 *2
Out[155]:
d     8
b    14
a   -10
c     6
dtype: int64

In [156]: np.exp(obj2)
Out[156]:
d      54.598150
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Seriese를 이해하는 다른 방법은 고정 길이의 정렬된 사전형이라고 생각하는 것이다. Series는 색인값에 데이터값을 매핑하고 있으므로 파이썬의 사전형과 비슷하다. Series 객체는 파이썬의 사전형을 인자로 받아야 하는 많은 함수에서 사전형을 대체하여 사용할 수 있다.

```
In [157]: 'b' in obj2
Out[157]: True

In [158]: 'e' in obj2
Out[158]: False
```

파이썬 사전형에 데이털르 저장해야 한다면 파이썬 사전 객체로부터 Series 객체를 생성할 수도 있다.

```
In [159]: sdata = {'Ohio':35000, 'Taxas':71000, 'Oregon':16000, 'Utah':5000}

In [160]: obj3 = pd.Series(sdata)

In [161]: obj3
Out[161]:
Ohio      35000
Taxas     71000
Oregon    16000
Utah       5000
dtype: int64
```

사전 객체만 가지고 Series 객체를 생성하면 생성된 Series 객체의 색인에는 사전의 키값이 순서대로 들어간다. 색인을 지정하고 싶다면 원하는 순서대로 색인을 넘겨 줄 수 있다.

```
In [162]: states = ['california','Ohio', 'Oregon', 'Texas']

In [163]: obj4 = pd.Series(sdata, index = states)

In [164]: obj4
Out[164]:
california        NaN
Ohio          35000.0
Oregon        16000.0
Texas             NaN
dtype: float64
```

이 예제를 보면 sdata에 있는 값 중 3개만 확인할 수 있는데, 'California' 에 대한 값은 찾을 수 ㅇㅄ기 때문이다. 이 값은 NaN(Not a Number)로 표시되고, pandas 에서는 누락된 값, 혹은 NA값으로 취급된다. 'Utah' 는 states에 포함되어 있지 않으므로 실행 결과에서 빠지게 된다.

나는 앞으로 '누락된' 또는 'NA'를 누락된 데이터를 지칭하는 데 사용하도록 하겠다. pandas의 innull과 notnull 함수는 누락된 데이터를 찾을 때 사용된다.

```
In [165]: pd.isnull(obj4)
Out[165]:
california     True
Ohio          False
Oregon        False
Texas          True
dtype: bool

In [166]: pd.notnull(obj4)
```

```
Out[166]:
california    False
Ohio           True
Oregon         True
Texas         False
dtype: bool
```

Series의 유용한 기능은 산술 연산에서 색인과 라벨로 자동 정렬하는 것이다.

```
In [167]: obj3
Out[167]:
Ohio      35000
Taxas     71000
Oregon    16000
Utah       5000
dtype: int64

In [168]: obj4
Out[168]:
california        NaN
Ohio          35000.0
Oregon        16000.0
Texas             NaN
dtype: float64

In [169]: obj3+obj4
Out[169]:
Ohio          70000.0
Oregon        32000.0
Taxas             NaN
Texas             NaN
Utah              NaN
california        NaN
dtype: float64
```

데이터 정렬에 대한 내용은 차후에 더 있다. Series 객체와 Series의 색인은 모두 name 속성이 있는데 이 속성은 pandas의 핵심 기능과 밀접한 관련이 있다.

```
In [171]: obj4.name = 'population'

In [172]: obj4.index.name = 'state'

In [173]: obj4
Out[173]:
state
california        NaN
Ohio          35000.0
Oregon        16000.0
Texas             NaN
Name: population, dtype: float64
```

Series의 색인은 대입하여 변경할 수 있다.

```
In [175]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [176]: obj
Out[176]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

5.1.2 DataFrame

DataFrame은 표 같은 스프레드시트 형식의 자료구조이고 여러 개의 컬럼이 있는데 각 칼럼은 서로 다른 종류의 값(숫자, 문자열, 불리언 등)을 담을 수 있다. DataFrame은 로우와 컬럼에 대한 색인을 가지고 있는데 색인의 모양이 같은 Series 객체를 담고 있는 파이썬 사전으로 생각하면 편하다.

내부적으로 데이터는 리스트나 사전 또는 1차원 배열을 담고 있는 다른 컬렉션이 아니라 하나 이상의 2차원 배열에 저장된다. 구체적인 DataFrame의 내부 구조는 이 책에서 다루는 내용과 한참 다르므로 생략하겠다.

Note_ 물리적으로 DataFrame 은 2차언이지만 계층적 색인을 이용해서 좀 더 고차원의 대아토룰 표현할 수 있으며 이를 포함하여 pandas에서 데이터를 다루는 고급 기법은 이후에 설명하겠다.

DataFrame는 다양한 방법으로 생성할 수 있지만 가장 흔하게 사용되는 방법은 같은 길이의 리스트에 담긴 사전을 이용하거나 NumPy 배열을 이용한 것이다.

```
In [184]: data = {'state' : ['ohio', 'ohio', 'ohio', 'nevada', 'nevada',
'nevada'],
     ...:         'year' : [2000,2001,2002,2001,2002,2003],
     ...:         'pop': [1.5,1.7,3.6,2.4,2.9,3.2]}

In [185]: frame = pd.DataFrame(data)
```

만들어진 DataFrame의 색인은 Series와 같은 방식으로 자동으로 대입되며 컬럼은 정렬되어 저장된다.

```
In [186]: frame
Out[186]:
    state  year  pop
0    ohio  2000  1.5
1    ohio  2001  1.7
2    ohio  2002  3.6
3  nevada  2001  2.4
4  nevada  2002  2.9
5  nevada  2003  3.2
```

주피터 노트북을 사용한다면 DataFrame 객체는 브라우저에서 좀 더 보기 편하도록 HTML표 형식으로 출력될 것이다.

큰 DataFrame을 다룰 때는 head 매서드를 이용해서 처음 5개 로우만 출력할 수도 있다.

```
In [187]: frame.head()
Out[187]:
    state  year  pop
0    ohio  2000  1.5
1    ohio  2001  1.7
2    ohio  2002  3.6
3  nevada  2001  2.4
4  nevada  2002  2.9
```

원하는 순서대로 columns를 지정하면 원하는 순서를 가진 DataFrame 객체가 생성된다,

```
In [188]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[188]:
   year   state  pop
0  2000    ohio  1.5
1  2001    ohio  1.7
2  2002    ohio  3.6
3  2001  nevada  2.4
4  2002  nevada  2.9
5  2003  nevada  3.2
```

Series와 마찬가지로 사전에 없는 값을 넘기면 결측치로 저장된다.

```
In [191]: frame2 = pd.DataFrame(data, columns = ['year', 'state', 'pop', 'debt'],
     ...:                                index = ['one', 'two', 'three', 'four',
'five', 'six']
     ...: )
     ...:
In [192]: frame2
Out[192]:
       year   state  pop debt
one    2000    ohio  1.5  NaN
two    2001    ohio  1.7  NaN
three  2002    ohio  3.6  NaN
four   2001  nevada  2.4  NaN
five   2002  nevada  2.9  NaN
six    2003  nevada  3.2  NaN

In [193]: frame2.columns
Out[193]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame의 칼럼은 Series 처럼 사전 형식의 표기법으로 접근하거나 속성 형식으로 접근할 수 있다.

```
In [194]: frame2['state']
Out[194]:
one        ohio
two        ohio
three      ohio
four     nevada
five     nevada
```

```
six      nevada
Name: state, dtype: object

In [195]: frame.year
Out[195]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

Note 편의를 위해 IPython 에서는 frame2.year 처럼 속성에 접근하듯 사용하거나 탭을 이용한 자동완성 기능을 제공한다. frame2[column] 형태로 사용하는 것은 어떤 칼럼이든 가능하지만 frame2[column] 형태로 사용하는 것은 어떤 컬럼이든 가능하지만 frame2.column 형태로 사용하는 것은 파이썬에서 사용가능한 변수 이름 형식일 때만 작동한다.

반환된 Series 객체가 DataFrame과 같은 색인을 가지면 알맞은 값으로 name 속성이 채워진다.

row는 위치나 loc 속성을 이용해서 이름을 통해 접근할 수 있다.

```
In [197]: frame2.loc['three']
Out[197]:
year     2002
state    ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

컬럼은 대입이 가능하다. 예를 들어 현재 비어 있는 'debt' 칼럼에 스칼라값이나 배열의 값을 대입할 수 있다.

```
In [198]: frame2['debt'] = 16.5

In [199]: frame2
Out[199]:
       year   state  pop  debt
one    2000    ohio  1.5  16.5
two    2001    ohio  1.7  16.5
three  2002    ohio  3.6  16.5
four   2001  nevada  2.4  16.5
five   2002  nevada  2.9  16.5
six    2003  nevada  3.2  16.5

In [200]: frame2['debt'] = np.arange(6)

In [201]: frame2
Out[201]:
       year   state  pop  debt
one    2000    ohio  1.5     0
two    2001    ohio  1.7     1
three  2002    ohio  3.6     2
four   2001  nevada  2.4     3
five   2002  nevada  2.9     4
six    2003  nevada  3.2     5
```

리스트나 배열을 컬럼에 대입할 때는 대입하려는 값의 길이가 DataFrame의 크기와 동일해야 한다.

Series를 대입하면 Dataframe의 색인에 따라 값이 대입되며 존재하지 않는 색인에는 결측치가 대입된다.

```
In [202]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [203]: frame2['debt'] = val

In [204]: frame2
Out[204]:
       year   state  pop  debt
one    2000    ohio  1.5   NaN
two    2001    ohio  1.7  -1.2
three  2002    ohio  3.6   NaN
four   2001  nevada  2.4  -1.5
five   2002  nevada  2.9  -1.7
six    2003  nevada  3.2   NaN
```

존재하지 않는 칼럼을 대입하면 새로운 칼럼을 생성한다. 파이썬 사전형에서처럼 del 예약어를 사용해서 칼럼을 삭제할 수 있다.

다음은 del 예약어에 대한 예제로, state 칼럼의 값이 'Ohio' 인지 아닌지에 대한 불리언값을 담고 있는 새로운 칼럼을 생성해보자.

```
In [205]: frame2['eastern']=frame2.state == 'ohio'

In [206]: frame2
Out[206]:
       year   state  pop  debt  eastern
one    2000    ohio  1.5   NaN     True
two    2001    ohio  1.7  -1.2     True
three  2002    ohio  3.6   NaN     True
four   2001  nevada  2.4  -1.5    False
five   2002  nevada  2.9  -1.7    False
six    2003  nevada  3.2   NaN    False
```

CAUTION_ 이 새로운 칼럼은 frame2.eastern 형태의 문법으로는 생성되지 않는다.

del 예약어를 이용해서 이 컬럼을 삭제할 수 있다.

```
In [207]: del frame2['eastern']

In [210]: frame2.columns
Out[210]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

In [211]: frame2
Out[211]:
       year   state  pop  debt
one    2000    ohio  1.5   NaN
two    2001    ohio  1.7  -1.2
three  2002    ohio  3.6   NaN
four   2001  nevada  2.4  -1.5
five   2002  nevada  2.9  -1.7
six    2003  nevada  3.2   NaN
```

*CAUTION DataFrame의 색인을 이용해서 얻은 컬럼은 내부 데이터에 대한 **뷰(View)** 이며 복사가 이뤄지지 않는다. 따라서 이렇게 얻은 Series 객체에 대한 변경은 실제 DataFrame에 반영된다. 복사본이 필요할 때는 Series 와 copy 메서드를 이용하자.*

중첩된 사전을 이용해서 데이터를 생성할 수 있다. 다음과 같은 중첩된 사전이 있다고 하자.

```
In [212]: pop = {'Nevada': {2001:2.4,2002:2.9},
     ...:        'Ohio'  : {2000:1.5,2001:1.7,2002:3.6} }
```

이 중첩된 사전을 DataFrame에 넘기면 바깥에 있는 사전의 키는 칼럼이 되고 안에 있는 키는 로우가 된다.

```
In [213]: frame3 = pd.DataFrame(pop)

In [214]: frame3
Out[214]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
```

NumPy 배열과 유사한 문법으로 데이터를 전치(컬럼과 로우를 뒤집음) 할 수 있다.

```
In [215]: frame3.T
Out[215]:
        2001  2002  2000
Nevada   2.4   2.9   NaN
Ohio     1.7   3.6   1.5
```

중복된 사전을 이용해서 DataFrame을 생성할 때 안쪽에 있는 사전값은 키값별로 조합되어 결과의 색인이 되지만 색인을 직접 지정하면 지정된 색인으로 DataFrame을 생성한다.

```
In [216]: pd.DataFrame(pop, index = [2001,2002,2003])
Out[216]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Series 객체를 담고 있는 사전 데이터도 같은 방식으로 취급된다.

```
In [220]: pdata = {'Ohio': frame3['Ohio'][:-1],
     ...:          'Nevada' : frame3['Nevada'][:2] }

In [221]: pd.DataFrame(pdata)
Out[221]:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
```

DataFrame 생성자에 넘길 수 있는 자료형의 목록은 표로 정리해뒀다.

DataFrame 생성을 위한 입력 데이터의 종류

행	설명
2차원 ndarray	데이터를 담고 있는 행렬. 선택적으로 행(row)과 열(column)의 이름을 전달 할 수 있다.
배열, 리스트, 튜플의 사전	사전의 모든 항목은 같은 길이를 가져야 하며, 각 항목의 내용이 DataFrame의 컬럼이 된다.
NumPy의 구조화 배열	배열의 사전과 같은 방식으로 취급된다.
Series의 사전	Series의 각 값이 칼럼이 된다. 명시적으로 색인을 넘겨주지 않으면 각 Series의 색인이 하나로 합쳐져서 로우의 색인이 된다.
사전의 사전	내부에 있는 사전이 칼럼이 된다. 키값은 'Series의 사전'과 마찬가지로 합쳐져서 로우의 색인이 된다.
사전이나 Series의 리스트	리스트의 각 항목이 DataFrame의 로우가 된다. 합쳐진 사전의 키값이나 Series의 색인이 DataFrame의 컬럼 이름이 된다.
리스트나 튜플의 리스트	'2차원 ndarray'의 경우와 같은 방식으로 취급된다.
다른 DataFrame	색인을 따로 지정하지 않으면 DataFrame의 색인이 그대로 사용된다.
NumPy MaskedArray	'2차원 ndarray'의 경우와 같은 방식으로 취급되지만 마스크값은 반환되는 DataFrame에서 NA값이 된다.

만일 데이터프레임의 색인(index) 과 컬럼(columns)에 name 속성을 지정했다면 이 역시 함께 출력된다.

```
In [225]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [226]: frame3
Out[226]:
state  Nevada  Ohio
year
2001      2.4   1.7
2002      2.9   3.6
2000      NaN   1.5
```

Series와 유사하게 values 속성은 DataFrame에 저장된 데이터를 2차원 배열로 반환한다.

```
In [227]: frame3.values
Out[227]:
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

DataFrame의 컬럼이 서로 다른 dtype을 가지고 있다면 모든 컬럼을 수용하기 위해 그 컬럼의 배열 dtype이 선택된다.

```
In [227]: frame3.values
Out[227]:
```

```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])

In [228]: frame2.values
Out[228]:
array([[2000, 'ohio', 1.5, nan],
       [2001, 'ohio', 1.7, -1.2],
       [2002, 'ohio', 3.6, nan],
       [2001, 'nevada', 2.4, -1.5],
       [2002, 'nevada', 2.9, -1.7],
       [2003, 'nevada', 3.2, nan]], dtype=object)
```

5.1.3 색인 객체

pandas의 색인 객체는 표 형식의 데이터에서 각 로우와 컬럼에 대한 이름과 다른 메타데이터(축의 이름 등)를 저장하는 객체이다. Series나 DataFrame 객체를 생성할 때 사용되는 배열이나 다른 순차적인 이름은 내부적으로 색인으로 변환된다.

```
In [229]: obj = pd.Series(range(3), index=['a','b', 'c'])

In [230]: index = obj.index

In [231]: index
Out[231]: Index(['a', 'b', 'c'], dtype='object')

In [232]: index[1:]
Out[232]: Index(['b', 'c'], dtype='object')
```

색인 객체는 변경이 불가능하다.

```
In [235]: index[1] = 'd'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-235-a452e55ce13b> in <module>
----> 1 index[1] = 'd'

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in
__setitem__(self, key, value)
   3908
   3909     def __setitem__(self, key, value):
-> 3910         raise TypeError("Index does not support mutable operations")
   3911
   3912     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

그러므로 자료구조 사이에서 안전하게 공유될 수 있다.

```
In [244]: labels = pd.Index(np.arange(3))

In [245]: labels
Out[245]: Int64Index([0, 1, 2], dtype='int64')

In [247]: obj2 = pd.Series([1.5,-2.5,0], index= labels)
```

```
In [248]: obj2
Out[248]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [249]: obj2.index is labels
Out[249]: True
```

CAUTION_ 몇몇 독자는 색인이 제공하는 기능을 유용하게 사용할 일이 별로 없을 수 있지만 일부 연산의 경우 색인을 반환하기도 하므로 어떻게 동작하는지 이해하는 것은 매우 중요하다.

또한 배열과 유사하게 Index 객체도 고정 크기로 동작한다.

```
In [250]: frame3
Out[250]:
state  Nevada  Ohio
year
2001      2.4   1.7
2002      2.9   3.6
2000      NaN   1.5

In [251]: frame3.columns
Out[251]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [252]: 'Ohio' in frame3.columns
Out[252]: True

In [253]: 2003 in frame.index
Out[253]: False
```

파이썬의 집합과는 달리 pandas의 인덱스는 중복되는 값을 허용한다.

```
In [255]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [256]: dup_labels
Out[256]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

중복되는 값으로 선택을 하면 해당 값을 가진 모든 항목이 선택된다.

각각의 색인은 자신이 담고 있는 데이터에 대한 정보를 취급하기 위한 여러 가지 메서드와 속성을 가지고 있는데 이는 다음 표를 참고하자.

색인 메서드와 속성

메서드	설명
append	추가적인 색인 객체를 덧붙여 새로운 색인을 반환한다.
difference	색인의 차집합을 반환한다.
intersection	색인의 교집합을 반환한다.
union	색인의 합집합을 반환한다.
isin	색인이 넘겨받은 색인에 존재하는지 알려주는 불리언 배열을 반환한다.
delete	i 위치의 색인이 삭제된 새로운 색인을 반환한다.
drop	넘겨받은 값이 삭제된 새로운 색인을 반환한다.
insert	i 위치에 색인이 추가된 새로운 색인을 반환한다.
is_monotomic	색인이 단조성을 가진다면 True를 반환한다.
is_unique	중복되는 색인이 없다면 True를 반환한다.
unique	색인에서 중복되는 요소를 제거하고 유일한 값만 반환한다.

5.2 핵심기능

이 절에서는 Series나 DataFrame에 저장된 데이터를 다루는 기본적인 방법을 설명하겠다. 다음 몇몇 장에서는 pandas를 이용한 데이터 분석과 조작을 더 자세히 살펴볼 것이다. 이 책은 pandas 라이브러리의 완전한 설명은 포함하지 않고 중요한 기능에만 초점을 맞추고 있다. 잘 쓰이지 않는 내용에 대한 학습은 알아서 해라.

5.2.1 재색인

pandas 객체의 중요한 기능 중 하나는 **reindex**인데, 새로운 색인에 **맞도록** 객체를 새로 생성한다. 아래 간단한 예제를 살펴보자.

```
In [257]: obj = pd.Series([4.5, 7.2,-5.3,3.6],index = ['d','b','a','c'])

In [258]: obj
Out[258]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

이 Series 객체에 대해 reindex를 호출하면 데이터를 새로운 색인에 맞게 재배열하고, 존재하지 않는 색인값이 있다면 NaN을 추가한다.

```
In [259]: obj2 = obj.reindex(['a','b','c','d','e'])

In [260]: obj2
Out[260]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

시계열 같은 순차적인 데이터를 재색인할 때 값을 보강하거나 채워 넣어야 할 경우가 있다. method 옵션을 이용해서 이를 해결할 수 있으며, ffill 같은 메서드를 이용해서 누락된 값을 직전의 값으로 채워 넣을 수 있다.

```
In [261]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index = [0,2,4])

In [262]: obj3
Out[262]:
0      blue
2    purple
4    yellow
dtype: object

In [263]: obj3.reindex(range(6), method = 'ffill')
Out[263]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

DataFrame에 대한 reindex는 로우(색인), 컬럼 또는 둘 다 변경 가능하다. 그냥 순서만 전달하면 로우가 재색인된다.

```
In [264]: frame = pd.DataFrame(np.arange(9).reshape(3,3),
     ...:                      index = ['a', 'c', 'd'],
     ...:                      columns = ['Ohio', 'Texas', 'California']
     ...: )

In [265]: frame
Out[265]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [266]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [267]: frame2
Out[267]:
   Ohio  Texas  California
a   0.0    1.0         2.0
```

```
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

칼럼은 columns 예약어를 사용해서 재색인 할 수 있다.

```
In [268]: states = ['Texas', 'Utah', 'California']

In [269]: frame.reindex(columns = states )
Out[269]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

reindex의 인자는 다음표를 참조하자.

인자	설명
index	색인으로 사용할 새로운 순서. index 인스턴스나 다른 순차적인 자료구조 사용가능하다. index는 복사가 이뤄지지 않고 그대로 사용된다.
method	채움 메서드. ffill은 직전 값을 채워 넣고 bfill은 다음 값을 채워 넣는다.
fill_value	재색인 과중 중에 새롭게 나타나는 비어 있는 데이터를 채우기 위한 값
limit	전/후 보간 시에 사용할 최대 갭 크기(채워넣을 원소의 수)
tolerance	전/후 보간 시에 사용할 최대 갭 크기(값의 차이)
level	MultiIndex의 단계(level) 에 단순 색인을 맞춘다. 그렇지 않으면 MultiIndex의 하위 집합에 맞춘다.
copy	True인 경우 새로운 색인이 이전 색인과 동일하더라도 데이터를 복사한다. False인 경우 새로운 색인이 이전 색인과 동일한 경우 복사하지 않는다.

```
In [270]: obj3.reindex(range(6), method = 'bfill')
Out[270]:
0      blue
1    purple
2    purple
3    yellow
4    yellow
5       NaN
dtype: object
```

5.2.2. 하나의 로우나 컬럼 삭제하기

색인 배열, 또는 삭제하려는 로우나 칼럼이 제외된 리스트를 이미 가지고 있다면 로우나 컬럼을 쉽게 삭제할 수 있는데 이 방법은 데이터의 모양을 변경하는 작업이 필요하다. drop 메서드를 사용하면 선택한 값들이 삭제된 새로운 객체를 얻을 수 있다.

```
In [273]: obj = pd.Series(np.arange(5.), index = ['a', 'b', 'c', 'd', 'e'])

In [274]: obj
```

```
Out[274]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [275]: new_obj = obj.drop('c')

In [276]: new_obj
Out[276]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [277]: obj.drop(['c', 'd'])
Out[277]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

DataFrame 에서는 로우와 컬럼 모두에서 값을 삭제 할 수 있다. 다음 예제를 살펴보자.

```
In [290]: In [264]: data = pd.DataFrame(np.arange(16).reshape(4,4),^M
     ...:      ...:                       index = ['Ohio', 'Colorado', 'Utah',
'New York'],^M
     ...:      ...:                       columns = ['one', 'two', 'three',
'four']^M
     ...:      ...: )

In [291]: data
Out[291]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

drop 함수에 인자로 로우 이름을 넘기면 해당 로우(axis 0)의 값을 모두 삭제 한다.

```
In [292]: data.drop(['Colorado', 'Ohio'])
Out[292]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

컬럼의 값을 삭제할 때는 axis =1 또는 axis = 'columns'를 인자로 넘겨주면 된다.

```
In [293]: data.drop('two', axis = 1)
Out[293]:
          one  three  four
Ohio        0      2     3
```

```
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [294]: data.drop(['two', 'four'], axis = 'columns')
Out[294]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

drop 함수처럼 Series 나 DataFrame의 크기 또는 형태를 변경하는 함수는 새로운 객체를 반환하는 대신 원본 객체를 변경한다.

```
In [295]: obj.drop('c', inplace = True)

In [296]: obj
Out[296]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

inplace 옵션을 사용하는 경우 버려지는 값을 모두 삭제하므로 주의해서 사용하자.

5.2.3 색인하기, 선택하기, 거르기

Series의 색인 (obj[...]) 은 NumPy 배열의 색인과 유사하게 동작하지만 정수가 아니어도 된다는 점이 다르다. 몇 가지 예제를 살펴보자.

```
In [297]: obj = pd.Series(np.arange(4.), index = ['a', 'b', 'c', 'd'])

In [299]: obj
Out[299]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [300]: obj['b']
Out[300]: 1.0

In [301]: obj[1]
Out[301]: 1.0

In [302]: obj[2:4]
Out[302]:
c    2.0
d    3.0
dtype: float64
```

```
In [303]: obj[['b', 'a', 'd']]
Out[303]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [304]: obj[[1,3]]
Out[304]:
b    1.0
d    3.0
dtype: float64

In [305]: obj[obj<2]
Out[305]:
a    0.0
b    1.0
dtype: float64
```

라벨이름으로 슬라이싱하면 시작점과 끝점을 포함한다는 것이 일반 파이썬에서의 슬라이싱과 다른 점이다.

```
In [306]: obj['b':'c']
Out[306]:
b    1.0
c    2.0
dtype: float64
```

슬라이싱 문법으로 선택된 영역에 값을 대입하는 것은 생각하는 대로 동작한다.

```
In [307]: obj['b':'c'] = 5

In [308]: obj
Out[308]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

색인으로 DataFrame에서 하나 이상의 컬럼 값을 가져올 수 있다.

```
In [309]: data['two']
Out[309]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32

In [310]: data[['three', 'one']]
Out[310]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
```

```
New York     14   12
```

슬라이싱으로 로우를 선택하거나 불리언 배열로 로우를 선택할 수 있다.

```
In [311]: data [:2]
Out[311]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [313]: data[data['three']>5]
Out[313]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

data[:2] 형식의 문법으로 편리하게 로우를 선택할 수 있다. [] 연산자에 단일 값을 넘기거나 리스트를 넘겨서 여러 컬럼을 선택할 수 있다. 또 다른 방법으로는 스칼라 비교를 이용해 생성된 불리언 Dat aFrame을 사용해서 값을 선택하는 것이다.

```
In [316]: data < 5
Out[316]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [317]: data[data<5]= 0

In [318]: data
Out[318]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

위 예제는 DataFrame을 2차원 ndarray와 문법적으로 비슷하게 보이도록 의도한 것이다.

loc과 iloc으로 선택하기

DataFrame의 로우에 대해 라벨로 색인하는 방법은 특수한 색인 필드인 loc와 iloc을 소개한다. 이 방법을 이용하면 NumPy와 비슷한 방식에 추가적으로 축의 라벨을 사용하여 DataFrame의 로우와 컬럼을 선택할 수 있다. 축의 이름을 선택할 때는 loc을, 정수 색인으로 선택할 때는 iloc를 사용한다.

앞선 예제에서 축의 라벨로 하나의 로우와 여러 컬럼을 선택해보자.

```
In [319]: data.loc['Colorado', ['two', 'three']]
Out[319]:
two      5
three    6
Name: Colorado, dtype: int32
```

iloc을 이용하면 정수 색인으로도 위와 비슷하게 선택할 수 있다.

```
In [320]: data.iloc[2,[3,0,1]]
Out[320]:
four    11
one      8
two      9
Name: Utah, dtype: int32

In [321]: data.iloc[2]
Out[321]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32

In [322]: data.iloc[[1,2],[3,0,1]]
Out[322]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

이 두 함수는 슬라이스도 지원할 뿐더러 단일 라벨이나 라벨 리스트도 지원한다.

```
In [323]: data.loc[:'Utah', 'two']
Out[323]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32

In [324]: data.iloc[:,:3][data.three>5]
Out[324]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

지금까지 살펴봤듯이 pandas 객체에서 데이터를 선택하고 재배열하는 방법은 여러 가지가 있다.

다음 표에 다양한 방법을 요약해두었다. 나중에 살펴볼 계층적 색인을 이용하면 좀 더 다양한 방법을 사용할 수 있다.

NOTE_ pandas를 설계할 때 나는 칼럼을 선택하는 작업을 매우 빈번하게 사용하지만, 칼럼을 선택하기 위해 매번 frame[:, col]이라고 입력해야 하는 것은 너무 과하다고 생각했다. (입력할 내용이 많으면 에러가 발생할 확률도 높아진다). 그래서 라벨과 정수 색인을 이용해서 값을 선택할 수 있는 기능을 모두 ix 연산자에 밀어 넣기로 결정했다. 하지만 실사용성 측면에서 정수축 라벨을 사용할 경우 모호한 상황이 발생하는 빈도가 높아졌고 pandas 개발팀은 라벨과 정수 색인을 명시적으로 구분해서 사용하게끔 loc와 iloc를 추가했다.

DataFrame의 값 선택하기

방식	설명
df[val]	DataFrame에서 하나의 컬럼 또는 여러 컬럼을 선택한다. 편의를 위해 불리언 배열, 슬라이스, 불리언 DataFrame(어떤 기준에 근거해서 값을 대입해야 할 때)을 사용할 수 있다.
df.loc[val]	DataFrame에서 라벨값으로 로우의 부분집합을 선택한다.
df.loc[:,val]	DataFrame에서 라벨값으로 칼럼의 부분집합을 선택한다.
df.loc[val1,val2]	DataFrame에서 라벨값으로 로우와 칼럼의 부분집합을 선택한다.
df.iloc[where]	DataFrame에서 정수 색인으로 로우의 부분집합을 선택한다.
df.iloc[:,where]	DataFrame에서 정수 색인으로 칼럼의 부분집합을 선택한다.
df.iloc[where_i,where_j]	DataFrame에서 정수 색인으로 로우와 칼럼의 부분집합을 선택한다.
df.at[label_i, label_j]	로우와 컬럼의 정수 색인으로 단일 값을 선택한다.
df.iat[i,j]	로우와 컬럼의 정수 색인으로 단일 값을 선택한다.
reindex 메서드	하나 이상의 축을 새로운 색인으로 맞춘다.
get_value, set_value 메서드	로우와 컬럼 이름으로 DataFrame의 값을 선택한다.

5.2.4 정수 색인

정수 색인으로 pandas 객체를 다루다보면 리스트나 튜플 같은 파이썬 내장 자료구조에서 색인을 다루는 방법과의 차이점 때문에 실수하는 경우가 있다. 예를 들어 다음 코드가 에러를 발생할 것이라고 생각하지 않을 것이다.

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

이 경우 pandas는 라벨 색인을 찾는데 실패하므로 정수 색인으로 값을 찾는다. 하지만 이를 어떤 경우에도 버그 없이 잘 작동하도록 구현하기란 쉽지 않다. 라벨 색인이 0,1,2를 포함하는 경우 사용자가 라벨 색인으로 선택하려는 것인지 정수 색인으로 선택하려는 것인지 추측하기 쉽지 않다.

```
In [327]: ser
Out[327]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

반면 정수 기반의 색인을 사용하지 않는 경우 이런 모호함은 사라진다.

```
In [329]: ser2 = pd.Series(np.arange(3.),index=['a','b','c'])

In [330]: ser2[-1]
Out[330]: 2.0

In [331]: ser2[-2]
```

```
Out[331]: 1.0

In [332]: ser2[-3]
Out[332]: 0.0

In [333]: ser2
Out[333]:
a    0.0
b    1.0
c    2.0
dtype: float64
```

일관성을 유지하기 위해 정수값을 담고 있는 축 색인이 있다면 우선적으로 라벨을 먼저 찾아보도록 구현되어 있다. 좀 더 세밀하게 사용하고 싶다면 라벨에 대해서는 loc을 사용하고 정수 색인에 대해서는 iloc를 사용하자.

```
In [334]: ser[:1]
Out[334]:
0    0.0
dtype: float64

In [335]: ser.loc[:1]
Out[335]:
0    0.0
1    1.0
dtype: float64

In [336]: ser.iloc[:1]
Out[336]:
0    0.0
dtype: float64
```

5.2.5 산술 연산과 데이터 정렬

pandas에서 가장 중요한 기능 중 하나는 다른 색인을 가지고 있는 객체 간의 산술 연산이다. 객체를 더할 때 짝이 맞지 않는 색인이 있다면 결과에 두 색인이 통합된다. 데이터베이스를 사용해본 경험이 있다면 색인 라벨에 대한 외부 조인과 유사하게 동작한다고 생각할 수 있다. 예제를 보자.

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

In [2]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [4]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f',
'g'])

In [5]: s1
Out[5]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [6]: s2
Out[6]:
a   -2.1
```

```
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

이 두 객체를 더하면 다음과 같은 결과를 얻는다.

```
In [8]: s1 + s2
Out[8]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

서로 겹치는 색인이 없는 경우 데이터는 NA값이 된다. 산술 연산 시 누락된 값은 전파된다.

DataFrame의 경우 정률은 로우와 컬럼 모두에 적용된다.

```
In [14]: df1 = pd.DataFrame(np.arange(9.).reshape((3,3)), columns=list('bcd'),
    ...:                    index=['Ohio', 'Texas', 'Colorado'])

In [15]: df2 = pd.DataFrame(np.arange(12.).reshape((4,3)), columns=list('bde'),
    ...:                    index=['Utah','Ohio', 'Texas', 'Oragon'])

In [17]: df1
Out[17]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [18]: df2
Out[18]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oragon  9.0  10.0  11.0
```

이 두 DataFrame을 더하면 각 DataFrame에 있는 색인과 컬럼이 하나로 합쳐진다.

```
In [19]: df1 + df2
Out[19]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oragon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

'c'와 'e' 컬럼이 양쪽 DataFrame 객체에 존재하지 않으므로 결과에서는 모두 없는 값으로 나타난다. 로우 역시 마찬가지로 양쪽에 다 존재하지 않는 라벨에 대해서는 없는 값으로 나타난다.

공통되는 컬럼 라벨이나 로우 라벨이 없는 DataFrame을 더하면 결과에 아무것도 나타나지 않을 것이다.

```
In [20]: df1 = pd.DataFrame({'A':[1,2]})

In [21]: df2 = pd.DataFrame({'B':[3,4]})

In [22]: df1
Out[22]:
   A
0  1
1  2

In [23]: df2
Out[23]:
   B
0  3
1  4

In [24]: df1-df2
Out[24]:
    A   B
0 NaN NaN
1 NaN NaN
```

산술 연산 메서드에 채워 넣을 값 지정하기

서로 다른 색인을 가지는 객체 간의 산술 연산에서 존재하지 않는 축의 값을 특수한 값(0과 같은) 으로 지정하고 싶을 때는 다음과 같이 할 수 있다.

```
In [25]: df1 = pd.DataFrame(np.arange(12.).reshape((3,4)),
    ...:                    columns=list('abcd') )

In [26]: df2 = pd.DataFrame(np.arange(20.).reshape((4,5)),
    ...:                    columns=list('abcde'))

In [27]: df2
Out[27]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   6.0   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0

In [28]: df2.loc[1,'b'] = np.nan

In [29]: df1
Out[29]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [30]: df2
Out[30]:
```

```
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

위 둘을 더하면 겹치지 않는 부분은 NA 값이 된다.

```
In [31]: df1 + df2
Out[31]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

df1에 add 메서드를 사용하고, df2와 fill_value 값을 인자로 전달한다.

```
In [32]: df1
Out[32]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [33]: df2
Out[33]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0

In [34]: df1.add(df2, fill_value=0)
Out[34]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

다음 표에 Series와 DataFrame의 산술 연산 메서드를 정리했다. 각각의 산술 연산 메서드는 r로 시작하는 계산 인자를 뒤집어 계산하는 짝궁 메서드를 가진다.

```
In [35]: 1/df1
Out[35]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909

In [36]: df1.rdiv(1)
Out[36]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909
```

Series나 DataFrame을 재색인할 때도 fill_value를 지정할 수 있다.

```
In [37]: df1.reindex(columns=df2.columns, fill_value=0)
Out[37]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

산술 연산 메서드

메서드	설명
add, radd	덧셈(+)을 위한 메서드
sub,rsub	뺄셈(-)을 위한 메서드
div, rdiv	나눗셈(/)을 위한 메서드
floordiv, rfloordiv	소수점 내림(//) 연산을 위한 메서드
mul, rmul	곱셈(*)을 위한 메서드
pow, rpow	역승(**)을 위한 메서드

DataFrame과 Series 간의 연산

다른 차원의 NumPy 배열과의 연산처럼 DataFrame과 Series 간의 연산도 잘 정의되어 있다.

먼저 2차원 배열과 그 배열의 한 로우의 차이에 대해 생각할 수 있는 예제를 살펴보자.

```
In [38]: arr = np.arange(12.).reshape((3,4))

In [39]: arr
Out[39]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])

In [40]: arr[0]
Out[40]: array([0., 1., 2., 3.])
```

```
In [41]: arr-arr[0]
Out[41]:
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

arr에서 arr[0]을 빼면 계산은 각 로우에 대해 한 번씩만 수행된다. 이를 **브로드캐스팅**이라고 하는데 더 자세한 내용은 부록A에 있다. DataFrame과 Series 간의 연산은 이와 유사하다.

```
In [58]: frame.loc[:'Texas']
Out[58]:
         b    d    e
Utah   0.0  1.0  2.0
Ohio   3.0  4.0  5.0
Texas  6.0  7.0  8.0

In [59]: series = frame.iloc[0]

In [60]: series
Out[60]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64

In [61]: frame
Out[61]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

기본적으로 DataFrame과 Series 간의 산술 연산은 Series의 색인을 DataFrame의 컬럼에 맞추고 아래 로우로 전파한다.

```
In [62]: frame - series
Out[62]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

만약 색인값을 DataFrame의 컬럼이나 Series의 색인에서 찾을 수 없다면 그 객체는 형식을 맞추기 위해 재색인된다.

```
In [65]: series2 = pd.Series(np.arange(3), index=['b', 'e', 'f'])
    ...:

#e에 2가 아닌 1이 더해짐에 주목해라!!! 재색인 된다.
In [66]: frame + series2
Out[66]:
          b   d     e   f
```

```
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN

In [69]: series2
Out[69]:
b    0
e    1
f    2
dtype: int32

In [68]: frame
Out[68]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

만약 각 로우에 대해 연산을 수행하고 싶다면 산술 연산 메서드를 사용하면 된다. 예를 들어보자.

```
In [72]: series3 = frame['d']

In [73]: frame
Out[73]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [74]: series3
Out[74]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [75]: frame.sub(series3, axis='index')
Out[75]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0

In [76]: frame.sub(series3, axis=0)
Out[76]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

인자로 넘기는 axis 값은 **연산을 적용할 축 번호**이다. axis = 'index' 나 axis = 0은 DataFrame의 로우를 따라 연산을 수행하라는 의미이다.

5.2.6 함수 적용과 매핑

pandas 객체에도 NumPy의 유니버설 함수(배열의 각 원소에 적용되는 메서드) 를 적용할 수 있다.

```
In [79]: frame = pd.DataFrame(np.random.randn(4,3)
    ...:                      ,columns=list('bde')
    ...:                      ,index=['Utah','Ohio','Texas','Orego
    ...: n'])

In [80]: frame
Out[80]:
               b         d         e
Utah   -0.356591  0.956634 -1.165780
Ohio    2.135437 -0.584561  0.785446
Texas  -0.468037  1.135099  1.998468
Oregon  1.253462  1.020565  1.286889

In [81]: abs(frame)
Out[81]:
               b         d         e
Utah    0.356591  0.956634  1.165780
Ohio    2.135437  0.584561  0.785446
Texas   0.468037  1.135099  1.998468
Oregon  1.253462  1.020565  1.286889
```

자주 사용되는 또 다른 연산은 각 컬럼이나 로우의 1차원 배열의 함수를 적용하는 것이다.

DataFrame의 apply 메서드를 이용해 수행할 수 있다.

```
In [82]: f = lambda x: x.max()-x.min()

In [83]: frame.apply(f)
Out[83]:
b    2.603473
d    1.719660
e    3.164248
dtype: float64

In [84]: f
Out[84]: <function __main__.<lambda>(x)>
```

여기서 함수 f는 Series의 최댓값과 최솟값의 차이를 계산하는 함수이다. frame의 각 컬럼에 대해 한 번만 수행되며 결괏값은 계산을 적용한 컬럼을 색인하는 Series를 반환한다.

apply 함수에 axis = 'columns' 인자를 넘기면 각 로우에 대해 한 번씩만 수행된다.

```
In [87]: frame.apply(f, axis = 'columns')
Out[87]:
Utah      2.122414
Ohio      2.719998
Texas     2.466505
Oregon    0.266324
dtype: float64
```

배열에 대한 일반적인 통계(sum이나 mean 같은)는 DataFrame의 메서드로 존재하므로 apply메서드를 사용할 필요 없다.

apply 메서드에 전달된 함수는 스칼라값을 반환할 필요가 없다. 여러 값을 가진 Series를 반환해도 된다.

```
In [92]: def f(x):
    ...:     return pd.Series([x.max(),x.min()],index=['max','min
    ...: '])

In [93]: frame.apply(f)
Out[93]:
            b         d         e
max  2.135437  1.135099  1.998468
min -0.468037 -0.584561 -1.165780
```

배열의 각 원소에 적용되는 파이썬의 함수를 사용할 수도 있다. frame 객체에서 실숫값을 문자열 포맷으로 변환하고 싶다면 applymap을 이용해서 다음과 같이 할 수 있다.

```
In [95]: format = lambda x: '%.2f'% x

In [96]: frame.applymap(format)
Out[96]:
            b      d      e
Utah    -0.36   0.96  -1.17
Ohio     2.14  -0.58   0.79
Texas   -0.47   1.14   2.00
Oregon   1.25   1.02   1.29

In [97]: frame
Out[97]:
               b         d         e
Utah   -0.356591  0.956634 -1.165780
Ohio    2.135437 -0.584561  0.785446
Texas  -0.468037  1.135099  1.998468
Oregon  1.253462  1.020565  1.286889
```

이 메서드의 이름이 applymap인 이유는 Series는 각 원소에 적용할 함수를 지정하기 위한 map 메서드를 가지고 있기 때문이다.

```
In [98]: frame['e'].map(format)
Out[98]:
Utah      -1.17
Ohio       0.79
Texas      2.00
Oregon     1.29
Name: e, dtype: object
```

5.2.7 정렬과 순위

어떤 기준에 근거해서 데이터를 정렬하는 것 역시 중요한 명령이다. 로우나 컬럼의 색인을 알파벳순으로 정렬하려면 정렬된 새로운 객체를 반환하는 sort_index 메서드를 사용하면 된다.

```
In [99]: obj = pd.Series(range(4), index=['d','a','b', 'c'])

In [100]: obj.sort_index()
Out[100]:
a    1
b    2
c    3
d    0
dtype: int64
```

DataFrame은 로우나 칼럼 중 하나의 축을 기준으로 정렬할 수 있다.

```
In [101]: frame = pd.DataFrame(np.arange(8).reshape((2,4)),
     ...:                      index=['three','one'],
     ...:                      columns=['d','a','b','c'] )

In [102]: frame.sort_index()
Out[102]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [103]: frame.sort_index(axis=1)
Out[103]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

데이터는 기본적으로 오름차순으로 정렬되고 내림차순으로 정렬할 수도 있다.

```
In [104]: frame.sort_index(axis=1, ascending=False)
Out[104]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Series 객체를 값에 따라 정렬하고 싶다면 sort_values 메서드를 사용하면 된다.

```
In [105]: obj = pd.Series([4,7,-3,2])

In [106]: obj.sort_values()
Out[106]:
2   -3
3    2
0    4
1    7
dtype: int64
```

정렬할 때 비어 있는 값은 기본적으로 Series 객체에서 가장 마지막에 위치한다.

```
In [107]: obj = pd.Series([4,np.nan,7,np.nan,-3,2])

In [108]: obj.sort_values()
Out[108]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

DataFrame에서 하나 이상의 컬럼에 있는 값으로 정렬을 하는 경우 sort_values 함수의 by 옵션에 하나 이상의 컬럼 이름을 넘기면 된다.

```
In [109]: frame = pd.DataFrame({'b':[4,7,-3,2],'a':[0,1,0,1]})

In [110]: frame
Out[110]:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1

In [111]: frame.sort_values(by='b')
Out[111]:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

여러 개의 컬럼을 정렬하려면 컬럼 이름이 담긴 리스트를 전달하면 된다.

```
In [112]: frame.sort_values(by=['a','b'])
Out[112]:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

순위는 정렬과 거의 흡사한데, 1부터 배열의 유효한 데이터 개수까지 순서를 매긴다. 기본적으로 Series와 DataFrame의 rank 메서드는 동점인 항목에 대해서는 평균 순위를 매긴다.

```
In [113]: obj = pd.Series([7,-5,7,4,2,0,4])

In [114]: obj.rank()
Out[114]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

데이터 상에서 나타나는 순위에 따라 순위를 매길 수도 있다.

```
In [116]: obj.rank(method = 'first')
Out[116]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

여기서 0번째와 2번째 항목에 대해 평균 순위인 6.5를 적용하는 대신 먼저 출련한 순서대로 6과 7을 적용했다.

내림차순으로 순서를 매길 수도 있다.

```
In [117]: obj.rank(ascending=False, method='max')
Out[117]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

다음 표에서 사용 가능한 동률 처리 메서드를 나열해두었다.

DataFrame에서는 로우나 컬럼에 대해 순위를 정할 수 있다.

```
In [119]: frame = pd.DataFrame({'b':[4.3,7,-3,2],'a':[0,1,0,1],
     ...:                       'c':[-2,5,8,-2.5]})

In [120]: frame
Out[120]:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5
```

```
In [121]: frame.rank(axis=1)
Out[121]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

순위의 동률을 처리하는 메서드

메서드	설명
'average'	기본값. 같은 값을 가지는 행목들의 평균값을 순위로 삼는다.
'min'	같은 값을 가지는 그룹을 낮은 순위로 매긴다.
'max'	같은 값을 가지는 그룹을 높은 순위로 매긴다.
'first'	데이터 내의 위치에 따라 순위를 매긴다.
'dense'	method = 'min'과 같지만 같은 그룹 내에서 모두 같은 순위를 적용하지 않고 1씩 증가시킨다.

5.2.8 중복 색인

지금까지 살펴본 모든 예제는 축 이름(색인값)이 유일한 경우 밖에 없었다. pandas의 많은 함수(reindex 같은)에서 색인값은 유일해야 하지만 의무적이지는 않다. 중복된 색인값을 가지는 Series객체를 살펴보

```
In [122]: obj = pd.Series(range(5),index = ['a','a','b','b','c'])
     ...:

In [123]: obj
Out[123]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

색인의 is_unique 속성은 해당 값이 유일한지 아닌지 알려준다.

```
In [124]: obj.index.is_unique
Out[124]: False
```

중복되는 색인값이 있다면 색인을 이용해서 데이터에 접근했을 때 다르게 동작한다. 중복되는 색인값이 없을 때는 색인을 이용해서 데이터에 접근하면 스칼라값을 반환하지만 중복되는 색인값이 있을 때는 하나의 Series 객체를 반환한다.

```
In [125]: obj['a']
Out[125]:
a    0
a    1
dtype: int64

In [126]: obj['c']
Out[126]: 4
```

이는 라벨이 반복되는지 여부에 따라 색인을 이용해서 선택한 결과가 다를 수 있기 때문에 코드를 좀 더 복잡하게 만들 수 있다.

DataFrame에서 로우를 선택하는 것도 동일하다.

```
In [127]: df = pd.DataFrame(np.random.randn(4,3)
     ...:                   ,index=['a','a','b','b'])

In [128]: df
Out[128]:
          0         1         2
a  0.444392  0.172952  1.377211
a  0.843067 -0.125133 -1.075130
b  0.825088  0.476811 -1.498679
b  0.425003 -0.631890 -0.481425

In [129]: df.loc['b']
Out[129]:
          0         1         2
b  0.825088  0.476811 -1.498679
b  0.425003 -0.631890 -0.481425
```

5.3 기술 통계 계산과 요약

pandas 객체는 일반적인 수학 메서드와 통계 메서드를 가지고 있다. 이 메서드의 대부분은 하나의 Series나 DataFrame의 로우나 칼럼에서 단일 값(합이나 평균같은) 을 구하는 **축소** 혹은 **요약 통계** 범주에 속한다.

순수 NumPy 배열에서 제공하는 동일한 메서드와 비교하여 pandas의 메서드는 처음부터 누락된 데이터를 제외하도록 설계되었다. 다음과 같은 DataFrame을 생각해보자.

```
In [130]: df = pd.DataFrame([[1.4,np.nan],[7.1,-4.5],
     ...:                   [np.nan,np.nan],[0.75,-1.3]],
     ...:                   index=['a','b','c','d'],
     ...:                   columns=['one','two'])

In [131]: df
Out[131]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

DataFrame의 sum 메서드를 호출하면 각 컬럼의 합을 담은 Series를 반환한다.

```
In [132]: df.sum()
Out[132]:
one    9.25
two   -5.80
dtype: float64
```

axis = 'columns' 또는 axis = 1 옵션을 넘기면 각 컬럼의 합을 변환한다.

```
In [133]: df.sum(axis = 'columns')
Out[133]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

전체 로우나 컬럼의 값이 NA가 아니라면 NA값은 제외되고 계산된다. 이는 skipna 옵션으로 조정할 수 있다.

```
In [134]: df.mean(axis='columns',skipna=False)
Out[134]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

다음 표에서 공통적으로 사용되는 축소 메서드의 옵션을 확인할 수 있다.

축소 메서드의 옵션

옵션	설명
axis	연산을 수행할 축. DataFrame에서 0은 로우고 1은 칼럼이다.
skipna	누락된 값을 제외할 것인지 정하는 옵션. 기본값은 True이다.
level	계산하려는 축이 계층적 색인(다중 색인) 이라면 레벨에 따라 묶어서 계산한다.

idxmin이나 idxmax 같은 메서드는 최솟값 혹은 최댓값은 가지고 있는 색인값과 같은 간접 통계를 반환한다.

```
In [136]: df.idxmax()
Out[136]:
one    b
two    d
dtype: object
```

또 다른 메서드로 **누산**이 있다.

```
In [137]: df.cumsum()
Out[137]:
    one  two
a  1.40  NaN
```

```
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8

In [138]: df
Out[138]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

축소나 누산이 아닌 다른 종류의 메서드로 descibe가 있는데, 이 메서드는 한 번에 여러개의 통계 결과를 만들어낸다.

```
In [140]: df.describe()
Out[140]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

수치 데이터가 아닐 경우 descirbe는 다른 요약 통계를 생성한다.

```
In [142]: obj = pd.Series(['a','a','b','c']*4)

In [143]: obj.describe()
Out[143]:
count     16
unique     3
top        a
freq       8
dtype: object
```

다음 표에 요약 통계 관련 메서드의 전체 목록을 나열했다.

요약 통계 관련 메서드

메서드	설명
count	NA값을 제외한 값의 수를 반환 한다.
describe	Series나 DataFrame의 각 컬럼에 대한 요약 통계를 계산한다.
min,max	최솟값과 최댓값을 계산한다.
argmin,argmax	각각 최솟값과 최댓값을 담고 있는 색인의 위치(정수)를 반환한다.
idxmin,idxmax	각각 최솟값과 최댓값을 담고 있는 색인의 값을 반환한다.
quantile	0부터 1까지의 분위수를 계산한다.
sum	합을 계산한다.
mean	평균을 계산한다.
median	중간값(50% 분위)을 반환한다.
mad	평균값에서 평균절대편차를 계산한다.
prod	모든 값의 곱
var	표본분산의 값을 계산한다.
std	표본표준편차의 값을 계산한다.
skew	표본비대칭도(3차 적률)의 값을 계산한다.
kurt	표본첨도(4차 적률)의 값을 계산한다.
cumsum	누적값을 계산한다.
cummin, cummax	각각 누적 최솟값과 누적 최댓값을 계산한다.
cumprod	누적곱을 계산한다.
diff	1차 산술치를 계산한다(시계열 데이터 처리 시 유용하다).
pct.change	퍼센트 변화율을 계산한다.

5.3.1 상관관계와 공분산

상관관계나 공분산 같은 요약 통계 계산은 두 쌍의 인자를 필요로 한다. pandas-datareader 패키지를 이용해서 야후! 금융 사이트에서 구한 주식가격과 시가총액을 담고 있는 다음 DataFrame을 생각해보자. 아직 설치하지 않았다면 conda나 pip를 통해 설치할 수 있다.

```
conda install pandas-datareader
```

pandas-datareader 모듈을 이용해서 주가 정보를 다운ㄹ오드 하자.

```
In [153]: import pandas_datareader.data as web
C:\ProgramData\Anaconda3\lib\site-
packages\pandas_datareader\compat\__init__.py:7: FutureWarning:
pandas.util.testing is deprecated. Use the functions in the public API at
pandas.testing instead.
  from pandas.util.testing import assert_frame_equal

In [158]: all_data = {ticker: web.get_data_yahoo(ticker)
     ...:             for ticker in ['AAPL','IBM','MSFT','GOOG']}

In [165]: price = pd.DataFrame({ticker: data['Adj Close']
     ...:                       for ticker,data in all_data.items()})

In [166]: volume = pd.DataFrame({ticker: data['Volume']
     ...:                       for ticker,data in all_data.items()})
```

```
In [167]: price
Out[167]:
                 AAPL         IBM        MSFT         GOOG
Date
2015-04-16  116.109131  131.007416   38.015030   532.338440
2015-04-17  114.802368  129.031784   37.528114   522.615173
2015-04-20  117.425117  133.440796   38.691284   533.914124
2015-04-21  116.790138  131.914948   38.447842   532.507996
2015-04-22  118.363777  132.798309   38.763428   537.888245
...                ...         ...         ...          ...
2020-04-07  259.429993  114.940002  163.490005  1186.510010
2020-04-08  266.070007  119.290001  165.130005  1210.280029
2020-04-09  267.989990  121.500000  165.139999  1211.449951
2020-04-13  273.250000  121.150002  165.509995  1217.560059
2020-04-14  287.049988  123.910004  173.699997  1269.229980

[1258 rows x 4 columns]

In [168]: volume
Out[168]:
                 AAPL        IBM        MSFT       GOOG
Date
2015-04-16  28369000.0  3136900.0  22509700.0  1299800.0
2015-04-17  51957000.0  4314400.0  42387600.0  2151800.0
2015-04-20  47054300.0  9609200.0  46057700.0  1679200.0
2015-04-21  32435100.0  9683600.0  26013800.0  1844700.0
2015-04-22  37654500.0  4024800.0  25064300.0  1593500.0
...                ...        ...         ...        ...
2020-04-07  50721800.0  5595300.0  62769000.0  2387300.0
2020-04-08  42223800.0  5158600.0  48318200.0  1975100.0
2020-04-09  40529100.0  5576800.0  51431800.0  2175400.0
2020-04-13  32755700.0  5119700.0  41905300.0  1739800.0
2020-04-14  48612900.0  5087500.0  52776100.0  2468700.0

[1258 rows x 4 columns]
```

CAUTION_ 2017년에 버라이즌이 야후!를 인수했기에 이 책을 읽고 있는 시점에 야후! 금융 서비스가 더 이상 작동하지 않을 가능성도 있다. 최신 기능은 온라인에서 pandas-datareader 문서로 확인 할 수 있다.

이제 각 주식의 퍼센트 변화율을 계산해보자. 시계열을 다루는 법은 11장에서 자세히 설명하겠다.

```
In [9]: returns = price.pct_change()

In [10]: returns.tail()
Out[10]:
                AAPL       IBM      MSFT      GOOG
Date
2020-04-13  0.019628 -0.002881  0.002240  0.005044
2020-04-14  0.050503  0.022782  0.049483  0.042437
2020-04-15 -0.009127 -0.042127 -0.010478 -0.005326
2020-04-16  0.007946 -0.024939  0.030021  0.000792
2020-04-17 -0.013569  0.037933  0.008812  0.015655
```

corr 메서드는 NA가 아니며 정렬된 색인에서 연속하는 두 Series에 대해 상관관계를 계산하고 cov 메서드는 공분산을 계산한다.

```
In [11]: returns['MSFT'].corr(returns['IBM'])
Out[11]: 0.5915047729111016

In [12]: returns['MSFT'].cov(returns['IBM'])
Out[12]: 0.00015976960818835873
```

MSFT는 파이썬 속성 이름 규칙에 어긋나지 않으므로 보다 편리한 문법으로 해당 칼럼을 선택할 수 있다.

```
In [13]: returns.MSFT.corr(returns.IBM)
Out[13]: 0.5915047729111016
```

반면에 DataFrame에서 corr과 cov 메서드는 DataFrame 행렬에서 상관관계와 공분산을 계산한다.

```
In [14]: returns.corr()
Out[14]:
          AAPL       IBM      MSFT      GOOG
AAPL  1.000000  0.531470  0.695602  0.639329
IBM   0.531470  1.000000  0.591505  0.528921
MSFT  0.695602  0.591505  1.000000  0.744572
GOOG  0.639329  0.528921  0.744572  1.000000

In [15]: returns.cov()
Out[15]:
          AAPL       IBM      MSFT      GOOG
AAPL  0.000326  0.000150  0.000217  0.000195
IBM   0.000150  0.000243  0.000160  0.000139
MSFT  0.000217  0.000160  0.000300  0.000217
GOOG  0.000195  0.000139  0.000217  0.000284
```

DataFrame의 corrwith 메서드를 사용하면 다른 Series나 DataFrame 과의 상관관계를 계산한다.

Series를 넘기면 각 칼럼에 대해 계산한 상관관계를 담고 있는 Series를 반환한다.

```
In [16]: returns.corrwith(returns.IBM)
Out[16]:
AAPL    0.531470
IBM     1.000000
MSFT    0.591505
GOOG    0.528921
dtype: float64
```

DataFrame을 넘기면 맞아떨어지는 칼럼 이름에 대한 상관관계를 계산한다. 여기서 나는 시가총액의 퍼센트 변화율에 대한 상관관계를 계산해보았다.

```
In [17]: returns.corrwith(volume)
Out[17]:
AAPL   -0.139966
IBM    -0.101548
MSFT   -0.032154
GOOG   -0.039791
dtype: float64
```

axis='columns' 옵션을 넘기면 각 컬럼에 대한 상관관계와 공분산을 계산한다. 모든 경우 데이터는 상관관계를 계산하기 전에 색인의 이름순으로 정렬된다.

5.3.2 유일값, 값 세기, 멤버쉽

또 다른 종류의 메서드로는 1차원 Series에 담긴 값의 정보를 추출하는 메서드가 있다. 이를 설명하기 위해 다음과 같은 예제가 있다고 하자.

```
In [18]: obj = pd.Series(['c', 'a', 'd', 'a','a','b','b','c','c'])
```

unique 메서드는 중복되는 값을 제거하는 유일값만 담고 있는 Series를 반환한다.

```
In [19]: unique = obj.unique()

In [20]: unique
Out[20]: array(['c', 'a', 'd', 'b'], dtype=object)

In [21]: unique.sort()

In [22]: unique
Out[22]: array(['a', 'b', 'c', 'd'], dtype=object)
```

유일값은 정렬된 순서대로 반환되지 않지만 필요하다면 unique.sort()를 이용해서 나중에 정렬할 수 있다. 그리고 value_counts 메서드는 Series에서 도수를 계산하여 반환한다.

```
In [24]: obj.value_counts()
Out[24]:
c    3
a    3
b    2
d    1
dtype: int64
```

value_counts 에서 반환하는 Series는 담고 있는 값을 내림차순으로 정렬한다. value_counts 메서드는 pandas의 최상위 메서드로, 어떤 배열이나 순차 자료구조에서도 사용할 수 있다.

```
In [28]: pd.value_counts(obj.values, sort=False)
Out[28]:
b    2
d    1
a    3
c    3
dtype: int64
```

isin 메서드는 어떤 값이 Series에 존재하는지 나타내는 불리언 벡터를 반환하는데, Series나 DataFrame의 컬럼에서 값을 골라내고 싶을 때 유용하게 사용할 수 있다.

```
In [29]: obj
Out[29]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [30]: mask = obj.isin(['b','c'])

In [31]: mask
Out[31]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [32]: obj[mask]
Out[32]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

isin과 관련이 있는 Index.get_indexer 메서드는 여러 값이 들어 있는 배열에서 유일한 값의 색인 배열을 구할 수 있다.

```
In [33]: to_match = pd.Series(['c','a','b','b','c','a'])
```

```
In [34]: to_match
Out[34]:
0    c
1    a
2    b
3    b
4    c
5    a
dtype: object

In [35]: unique_vals = pd.Series(['c','b','a'])

In [36]: unique_vals
Out[36]:
0    c
1    b
2    a
dtype: object

In [37]: pd.Index(unique_vals).get_indexer(to_match)
Out[37]: array([0, 2, 1, 1, 0, 2], dtype=int64)
```

유일 값, 값 세기, 멤버십 메서드

메서드	설명
isin	Series의 각 원소가 넘겨받은 연속된 값에 속하는지 나타내는 불리언 배열을 반환한다.
match	각 값에 대해 유일한 값을 담고 있는 배열에서의 정수 색인을 계산한다. 데이터 정렬이나 조인 행렬의 연산을 하는 경우에 유용하다.
unique	Series에서 중복되는 값을 제거하고 유일값만 포함하는 배열을 반환한다. 결과는 Series에서 발견된 순서대로 반환된다.
value_counts	Series에서 유일값에 대한 색인과 도수를 계산한다. 도수는 내림차순으로 정렬된다.

DataFrame의 여러 갈럼에 대해 히스토그램을 구해야 하는 경우가 있다. 다음 예제를 보자.

```
In [38]: data = pd.DataFrame({'Qu1':[1,3,4,3,4],
    ...:                      'Qu2':[2,3,1,2,3],
    ...:                      'Qu3':[1,5,2,4,4]})

In [39]: data
Out[39]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

위 DataFrame의 apply 함수에 pandas.value_counts를 넘기면 다음과 같은 결과를 얻을 수 있다.

```
In [40]: result = data.apply(pd.value_counts).fillna(0)

In [41]: result
Out[41]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

여기서 결괏값의 로우 라벨은 전체 컬럼의 유일한 값들을 담고 있다. 각 값은 각 컬럼에서 해당 값이 몇 번 출현했는지 나타낸다.

5.4 마치며

다음 장에서는 pandas를 이용해서 데이터셋을 읽고(또는 로딩) 쓰는 도구를 다루도록 하겠다. 그 다음에는 데이터를 정제하고 분석하고 시각화하는 도구를 더 깊이 살펴본다.