# 12. 고급 pandas

## 12.1 Categorical 데이터

이 절에서는 pandas의 Catergorical 형을 활용하여 pandas 메모리 사용량을 줄이고 성능을 개선할 수 있는 방법을 소개한다. 통계와 머신러닝에서 범주형 데이터를 활용하기 위한 도구들도 함께 소개하겠다.

### 12.1.1 개발 배경과 동기

하나의 컬럼 내에 특정 값이 반복되어 존재하는 경우는 흔하다. 우리는 이미 배열 내에서 유일한 ㄱ밧을 추출하거나 특정 값이 얼마나 많이 존재하는지 확인할 수 있는 unique와 value_counts 같은 메서드를 공부했다.

```
In [48]: import numpy as np; import pandas as pd

In [49]: values = pd.Series(['apple', 'orange', 'apple', 'apple']*2)

In [50]: values
Out[50]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
dtype: object

In [51]: pd.unique(values)
Out[51]: array(['apple', 'orange'], dtype=object)

In [53]: pd.value_counts(values)
Out[53]:
apple     6
orange    2
dtype: int64
```

데이터웨어하우스, 분석 컴퓨팅 외 여러 다양한 데이터 시스템은 중복되는 데이터를 얼마나 효율적으로 저장하고 계산할 수 있는가를 중점으로 개발되었다. 데이터웨어하우스의 경우 구별되는 값을 담고 있는 **차원 테이블**과 그 테이블을 참조하는 정수키를 사용하는 것이 일반적이다.

```
In [54]: values = pd.Series([0,1,0,0]*2)

In [55]: dim = pd.Series(['apple', 'orange'])

In [56]: values
Out[56]:
0    0
1    1
2    0
3    0
4    0
```

```
5    1
6    0
7    0
dtype: int64

In [57]: dim
Out[57]:
0     apple
1    orange
dtype: object
```

take메서드를 사용하면 Series 내에 저장된 원래 문자열을 구할 수 있다.

```
In [66]: dim.take(values)
Out[66]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

여기서 정수로 표현된 값은 **범주형**또는 **사전형 표기법**이라고도 불린다. 별개의 값을 담고 있는 배열은 **범주, 사전, 단계 데이터** 라고 불린다. 이 책에서는 이러한 범주형 데이터를 가리키는 정수값은 **범주코드** 혹은 **코드**라고 하겠다.

범주형 표기법을 사용하면 분석 작업에 있어서, 엄청난 성능 향상을 얻을 수 있다. 범주 코드를 변경하지 않은 채로 범주형 데이터를 변형하는 것도 가증하다. 비교적 작은 연산으로 수행할 수 있는 변형의 예는 다음과 같다.

- **범주형 데이터의 이름 변경하기**
- **기존 범주형 데이터의 순서를 바꾸지 않고 새로운 범주 추가하기**

## 12.1.2 pandas의 Categorical

pandas에는 정수 기반의 범주형 데이터를 표현 할 수 있는 Categorical 형이라고 하는 특수한 데이터형이 존재한다.

```
In [67]: fruits = ['apple', 'orange', 'apple', 'apple']*2

In [68]: N = len(fruits)

In [70]: df = pd.DataFrame({'fruit':fruits,
    ...:                    'basket_id':np.arange(N),
    ...:                    'count':np.random.randint(3,15, size=N),
    ...:                    'weight':np.random.uniform(0,4,size = N)},
    ...:                    columns = ['basket_id','fruit','count','wei
    ...: ght'])

In [71]: df
Out[71]:
   basket_id   fruit  count    weight
0          0   apple     12  1.730742
```

```
1          1  orange     12  0.214986
2          2   apple     10  3.568365
3          3   apple     14  0.097853
4          4   apple     11  1.499447
5          5  orange      8  3.958023
6          6   apple      7  0.375225
7          7   apple      6  0.609766
```

아래 예제에서 df['fruit']는 파이썬 문자열 객체의 배열로, 아래 방법으로 쉽게 범주형 데이터로 변경할 수 있다.

```
In [74]: fruit_cat = df['fruit'].astype('category')

In [75]: fruit_cat
Out[75]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

fruit_cat의 값은 NumPy 배열이 아닌 pandas.Categorical의 인스턴스다.

```
In [77]: c = fruit_cat.values

In [78]: type(c)
Out[78]: pandas.core.arrays.categorical.Categorical
```

Categorical 객체는 categories와 codes 속성을 가진다.

```
In [79]: c.categories
Out[79]: Index(['apple', 'orange'], dtype='object')

In [80]: c.codes
Out[80]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

변경 완료된 값을 대입함으로써 DataFrame의 컬럼을 범주형으로 변경할 수 있다.

```
In [81]: df['fruit'] = df['fruit'].astype('category')

In [82]: df.fruit
Out[82]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
```

```
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

파이썬 열거형에서 pandas.Catergorical 형을 직접 생성하는 것도 가능하다.

```
In [83]: my_categories = pd.Categorical(['foo','bar','baz','foo','bar']
    ...: )

In [84]: my_categories
Out[84]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

기존의 정의된 범주와 범주 코드가 있다면 from_codes 함수를 이용해서 범주형 데이터를 생성하는 것도 가능하다.

```
In [85]: categories = ['foo','bar', 'baz']

In [86]: codes = [0,1,2,0,0,1]

In [87]: my_cats_2 = pd.Categorical.from_codes(codes,categories)

In [88]: my_cats_2
Out[88]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

범주형으로 변경하는 경우 명시적으로 지정하지 않는 한 특정 순서를 보장하지 않는다. 따라서 categories 배열은 입력 데이터의 순서에 따라 다른 순서로 나타날 수 있다. from_codes를 사용하거나 다른 범주형 데이터 생성자를 이용하는 경우 순서를 지정할 수 있다.

```
In [89]: ordered_cat = pd.Categorical.from_codes(codes, categories, ord
    ...: ered=True)

In [90]: ordered_cat
Out[90]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

여기서 [foo < bar < baz]는 foo, bar, baz 순서를 가진다는 의미다. 순서가 없는 범주형 인스턴스는 as_ordered 메서드를 이용해 순서를 가지도록 만들 수 있다.

```
In [91]: my_cats_2.as_ordered()
Out[91]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

여기서는 문자열만 예로 들었지만 범주형 데이터는 꼭 문자열일 필요는 없다. 범주형 배열은 변경이 불가능한 값이라면 어떤 자료형이라도 포함할 수 있다.

### 12.1.3 Categorical 연산

pandas에서 Categorical 은 문자열 배열처럼 인코딩되지 않은 자료형을 사용하는 방식과 거의 유사하게 사용할 수 있다. groupby 같은 일부 pandas함수는 범주형 데이터에 사용할 때 더 나은 성능을 보여준다. ordered 플래그를 활용하는 함수들도 마찬가지다.

임의의 숫자 데이터를 pandas.qcut 함수로 구분해보자. 그렇게 하면 pandas.Categorical 객체를 반환한다.

```
In [98]: np.random.seed(12345)

In [99]: draws = np.random.randn(1000)

In [105]: draws[:5]
Out[105]: array([-0.20470766,  0.47894334, -0.51943872,
                 -0.5557303 ,  1.96578057])
```

이 데이터를 사분위로 나누고 통계를 내보자.

```
In [106]: bins = pd.qcut(draws, 4)

In [107]: bins
Out[107]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101],
(-0.684, -0.0101], (0.63, 3.928], ..., (-0.0101, 0.63],
(-0.684, -0.0101], (-2.9499999999999997, -0.684],
(-0.0101, 0.63], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.9499999999999997, -0.684]
< (-0.684, -0.0101] < (-0.0101, 0.63] <(0.63, 3.928]]
```

사분위 이름을 실제 데이터로 지정하는 것은 별로 유용하지 않다. qcut 함수의 labels 인자로 직접 이름을 지정하자.

```
In [108]: bins = pd.qcut(draws, 4, labels= ['Q1','Q2','Q3','Q4'])

In [109]: bins
Out[109]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [110]: bins.codes[:10]
Out[110]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

bins에 이름을 붙이고 나면 데이터의 시작값과 끝값에 대한 정보를 포함하지 않으므로 groupby를 이용해서 요약 통계를 내보자.

```
In [111]: bins = pd.Series(bins, name='quantile')


In [113]: results = (pd.Series(draws)
     ...:            .groupby(bins)
     ...:            .agg(['count','min','max'])
     ...:            .reset_index())

In [114]: results
```

```
Out[114]:
  quantile  count       min       max
0       Q1    250 -2.949343 -0.685484
1       Q2    250 -0.683066 -0.010115
2       Q3    250 -0.010032  0.628894
3       Q4    250  0.634238  3.927528
```

결과에서 quantile 컬럼은 bins의 순서를 포함한 원래 범주 정보를 유지하고 있다.

```
In [115]: results['quantile']
Out[115]:
0    Q1
1    Q2
2    Q3
3    Q4
Name: quantile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

### categorical을 이용한 성능 개선

특정 데이터셋에 대해 다양한 분석을 하는 경우 범주형으로 변환하는 것만으로도 전체 성능을 개선할 수 있다. 범주형으로 변환한 DataFrame의 컬럼은 메모리도 훨씬 적게 사용한다. 소수의 독립적인 카테고리로 분류되는 천만 개의 값을 포함하는 Series를 살펴보자.

```
In [116]: N = 10000000

In [117]: draws = pd.Series(np.random.randn(N))

In [118]: labels = pd.Series(['foo', 'bar', 'baz', 'qux']*(N//4))
```

labels를 categorical로 변환하자.

```
In [119]: categories = labels.astype('category')
```

categories 가 labels에 비해 훨씬 더 적은 메모리를 사용하는 것을 확인할 수 있다.

```
In [120]: labels.memory_usage()
Out[120]: 80000128

In [121]: categories.memory_usage()
Out[121]: 10000320
```

범주형으로 변환하는 과정이 그냥 이뤄지는 것은 아니지만 이는 한 번만 변환하면 되는 일회성 비용이다.

```
In [123]: %time _ = labels.astype('category')
Wall time: 413 ms
```

범주형에 대한 그룹 연산은 문자열 배열을 사용하는 대신 정수 기반의 코드 배열을 사용하는 알고리즘으로 동작하므로, 훨씬 빠르게 동작한다.

## 12.1.4 Categorical 메서드

범주형 데이터를 담고 있는 Series는 특화된 문자열 메서드인 Series.str과 유사한 몇가지 특수 메서드를 제공한다. 이를 통해 categories와 codes에 쉽게 접근할 수 있다. 다음 Series를 살펴보자.

```
In [124]: s = pd.Series(['a','b','c','d']*2)

In [125]: cat_s = s.astype('category')

In [126]: cat_s
Out[126]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

특별한 속성인 cat을 통해 categorical 메서드에 접근할 수 있다.

```
In [127]: cat_s.cat.codes
Out[127]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [128]: cat_s.cat.categories
Out[128]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

이 데이터의 실제 카테고리가 데이터에서 관측되는 4종류를 넘는 것을 이미 알고 있다고 가정하자. 이 경우 set_categories 메서드를 이용해서 변경하는 것이 가능하다.

```
In [129]: actual_categories = ['a','b','c','d','e']

In [130]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [131]: cat_s2
Out[131]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

데이터에는 변함이 없지만 위에서 변경한 대로 새로운 카테고리가 추가되었다. 예를 들어, value_counts를 호출해보면 변경된 카테고리를 반영하고 있다.

```
In [132]: cat_s.value_counts()
Out[132]:
d    2
c    2
b    2
a    2
dtype: int64

In [133]: cat_s2.value_counts()
Out[133]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

큰 데이터셋을 다룰 경우 categorical을 이용하면 메모리를 아끼고 성능도 개선할 수 있다. 분석 과정에서 큰 DataFrame이나 Series를  한 번 걸러내고 나면 실제로 데이터에 존재하지 않은 카테고리가 남아 있을 수 있다. 이 경우 remove_unused_categories 메서드를 이용해서 관측되지 않는 카테고리를 제거할 수 있다.

```
In [134]: cat_s3 = cat_s[cat_s.isin(['a','b'])]

In [135]: cat_s3
Out[135]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]

In [136]: cat_s3.cat.remove_unused_categories()
Out[136]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]
```

#### 모델링을 위한 더미값 생성하기

통계나 머신러닝 도구를 사용하다보면 범주형 데이터를 **더미값**으로 변환 해야 하는 경우가 생긴다. 이를 위해 각각의 구별되는 카테고리를 컬럼으로 가지는 DataFrame을 생성하는데, 각 컬럼에는 해당 카테고리 여부에 따라 0과 1의 값을 가진다.

```
In [137]: cat_s = pd.Series(['a','b','c','d']*2,dtype='category')

In [138]: pd.get_dummies(cat_s)
Out[138]:
```

```
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1

In [139]: cat_s
Out[139]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

## 12.2 고급 GroupBy 사용

### 12.2.1 그룹 변환과 GroupBy 객체 풀어내기

transform이라는 내장 메서드를 이용하면 apply 매서드와 유사하게 동작하면서도 사용할 수 있는 함수의 종류에 대해 좀 더 많은 제한을 포함할 수 있다.

- **그룹형태로 브로드캐스트할 수 있는 스칼라값을 생성해야 한다.**
- **입력 그룹과 같은 형태의 객체를 반환해야 한다.**
- **입력을 변경하지 않아야 한다.**

```
In [141]: df = pd.DataFrame({'key':['a','b','c']*4,
     ...:                    'value':np.arange(12.)})

In [142]: df
Out[142]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

key에 따른 그룹의 평균을 구해보자.

```
In [150]: g = df.groupby('key').value

In [152]: g.mean()
Out[152]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

df['value']와 같은 형태의 Series를 원하는 것 아닌 'key'에 따른 그룹의 평균값으로 값을 변경하기 원했다고 가정한다면 transform에 람다 함수 lambda x:x.mean()을 넘기면 된다.

```
In [153]: g.transform(lambda x:x.mean())
Out[153]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

내장 요약함수에 대해서는 agg 메서드에서처럼 문자열 그룹 연산 이름을 넘기면 된다.

```
In [155]: g.transform('mean')
Out[155]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

apply와 마찬가지로 transform은 Series를 반환하는 함수만 사용할 수 있지만 결과는 입력과 똑같은 크기여야 한다. 예를 들어 람다 함수를 이용해서 각 그룹에 모두 2를 곱할 수 있다.

```
In [156]: g.transform(lambda x:x*2)
Out[156]:
0     0.0
1     2.0
2     4.0
```

```
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64
```

좀 더 복잡한 예제로 각 그룹에 대해 내림차순으로 순위를 계산할 수도 있다.

```
In [157]: g.transform(lambda x:x.rank(ascending = False))
Out[157]:
0     4.0
1     4.0
2     4.0
3     3.0
4     3.0
5     3.0
6     2.0
7     2.0
8     2.0
9     1.0
10    1.0
11    1.0
Name: value, dtype: float64
```

간단한 요약을 통해 그룹 변환을 수행하는 함수를 살펴보자.

```
In [1]: def normalize(x):
   ...:     return (x-x.mean()) / x.std()
   ...:
```

이 경우에는 transform이나 apply를 이용해서 같은 결과를 얻을 수 있다.

```
In [7]: g.transform(normalize)
Out[7]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64

In [8]: g.apply(normalize)
Out[8]:
0    -1.161895
```

```
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

mean 이나 sum 같은 내장 요약함수는 일반적인 apply 함수보다 더 빠르게 동작한다. 또한 이 함수들을 transform 과 함께 사용하면 뒤로 되돌릴 수 있는데 이를 통해 그룹 연산을 풀어낼 수 있다.

```
In [9]: g.transform('mean')
Out[9]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64

In [11]: normalized = (df['value'] -g.transform('mean'))/g.transform('std')

In [12]: normalized
Out[12]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

그룹 연산을 풀어내면 수차례의 그룹 연산을 수행하지만 전체 벡터 연산의 장점이 더 크다.

### 12.2.2 시계열 그룹 리샘플링

시계열 데이터에서 resample 메서드는 의미적으로 시간 간격에 기반한 그룹 연산이다.

```
In [14]: times = pd.date_range('2017-05-20 00:00', freq = '1min', perio
    ...: ds = N)

In [15]: df = pd.DataFrame({'time' : times,
    ...:                    'value' : np.arange(N)})

In [16]: df
Out[16]:
                 time  value
0  2017-05-20 00:00:00      0
1  2017-05-20 00:01:00      1
2  2017-05-20 00:02:00      2
3  2017-05-20 00:03:00      3
4  2017-05-20 00:04:00      4
5  2017-05-20 00:05:00      5
6  2017-05-20 00:06:00      6
7  2017-05-20 00:07:00      7
8  2017-05-20 00:08:00      8
9  2017-05-20 00:09:00      9
10 2017-05-20 00:10:00     10
11 2017-05-20 00:11:00     11
12 2017-05-20 00:12:00     12
13 2017-05-20 00:13:00     13
14 2017-05-20 00:14:00     14
```

여기서 time으로 색인한 후 리샘플해보자.

```
In [17]: df.set_index('time').resample('5min').count()
Out[17]:
                     value
time
2017-05-20 00:00:00      5
2017-05-20 00:05:00      5
2017-05-20 00:10:00      5
```

key컬럼으로 구분되는 여러 시계열 데이터를 담고 있는 DataFrame을 생각해보자.

```
In [25]: df2 = pd.DataFrame({'time': times.repeat(3),
    ...:                     'key':np.tile(['a','b','c'],N),
    ...:                     'value':np.arange(N*3.)})

In [56]: df2[:7]
Out[56]:
                 time key  value
0 2017-05-20 00:00:00   a    0.0
1 2017-05-20 00:00:00   b    1.0
2 2017-05-20 00:00:00   c    2.0
3 2017-05-20 00:01:00   a    3.0
4 2017-05-20 00:01:00   b    4.0
5 2017-05-20 00:01:00   c    5.0
6 2017-05-20 00:02:00   a    6.0
```

'key'의 각 값에 대해 같은 리샘플을 수행하기 위해서는 pandas.TimeGrouper 객체를 이용한다.

```
In [37]: time_key = pd.Grouper(freq = '5min')
```

그리고 time을 색인으로 한 다음 'key'와 time_key로 그룹지어 합을 구해보자.

```
In [59]: resampled = (df2.set_index('time')
    ...:               .groupby(['key',time_key])
    ...:               .sum())

In [60]: resampled
Out[60]:
                        value
key time
a   2017-05-20 00:00:00   30.0
    2017-05-20 00:05:00  105.0
    2017-05-20 00:10:00  180.0
b   2017-05-20 00:00:00   35.0
    2017-05-20 00:05:00  110.0
    2017-05-20 00:10:00  185.0
c   2017-05-20 00:00:00   40.0
    2017-05-20 00:05:00  115.0
    2017-05-20 00:10:00  190.0

In [61]: resampled.reset_index()
Out[61]:
  key                time  value
0   a 2017-05-20 00:00:00   30.0
1   a 2017-05-20 00:05:00  105.0
2   a 2017-05-20 00:10:00  180.0
3   b 2017-05-20 00:00:00   35.0
4   b 2017-05-20 00:05:00  110.0
5   b 2017-05-20 00:10:00  185.0
6   c 2017-05-20 00:00:00   40.0
7   c 2017-05-20 00:05:00  115.0
8   c 2017-05-20 00:10:00  190.0
```

pd.Grouper를 쓸때는 시간값이 Series 혹은 DataFrame 이어야 한다.

---

## 12.3 메서드 연결 기법

데이터셋을 여러 차례 변형해야 하는 경우 분석에는 전혀 필요 없는 임시 변수를 계속 생성하는 상황이 발생한다. 다음 예제를 살펴보자.

```
In [62]: df = load_data()
    ...: df2 = df[df['col2']<0]
    ...: df2['col_demeaned'] = df2['col1'] - df2['col1'].mean()
    ...: result = df2.groupby('key').col1_demeaned.std()
```

여기서는 실제 데이터를 사용하지는 않지만 새로운 메서드 몇 가지를 만나게 되는데, 그 중 하나는 df[k] = v 처럼 컬럼에 값을 대입하는 함수형 DataFrame.assign 메서드다. 객체를 변경하는 대신 값 대입이 완료된 새로운 DataFrame을 반환한다. 아래 두 코드는 동일하다 .

```
          #실용적이지 않는 방법
In [63]: df2 = df.copy()
    ...: df2['k'] = v
    ...:
        #실용적인 방법
    ...: df2 = df.assign(k=v)
```

값을 직접 대입하는 것이 assign을 사용하는 것보다 빠르게 수행되지만 assign을 이용하면 메서드를 연결해서 사용할 수 있다.

```
In [65]: result = (df2.assign(col1_demeaned = df2.col1 - df2.col2.mean(
    ...: )).groupby('key').col1_demeaned.std())
```

여기서는 줄바꿈을 편리하게 하기 위해 위 코드를 괄호로 둘러쌓다. 메서드를 연결해서 사용할 때 주의해야 할 점은 임시 객체를 참조해야 할 경우가 있을 수도 있다는 점이다. 앞서 예제에서 load_data의 반환값을 임시 변수인 df에 잠기 전까지는 그 결과를 참조할 수 없었다. 이런 경우 assign 이나 호출이 가능한 객체 또는 함수를 인자로 받는 pandas의 다른 함수를 이용해서 해결할 수 있다.

호출이 가능한 객체(callable)의 예시를 보기 위해 위 예제의 일부 코드를 다시 살펴보자.

```
In [67]: df = load_data()
    ...: df2 = df[df['col2'] < 0]
```

위 코드는 다음과 같이 고쳐 쓸 수 있다.

```
In [68]: df = (load_data()
    ...:       [lambda x: x['col2']<0])
```

여기서 load_data의 결과를 변수에 저장하지 않았다. 그래서 []에 함수를 전달해서 메서드 연결이 이어지도록 했다.

계속해서 전체 코드를 하나의 메서드 연결 표현으로 작성할 수도 있다.

```
In [69]: result = (load_data()
    ...:           [lambda x: x.col2<0]
    ...:           .assign(col1_demeaned = lambda x:x.col1 - x.col1.mean())
    ...:           .groupby('key')
    ...:           .col1_demeaned.std())
```

### 12.3.1 pipe 메서드

pandas의 내장 함수와 방금 살펴본 메서드 연결을 통해 다양한 일을 할 수 있다. 하지만 직접 작성한 함수나 다른 서드파티 라이브러리의 함수를 사용해야 하는 경우도 생긴다. 이때 pipe 메서드를 사용할 수 있다. 다음과 같은 일련의 함수 호출을 생각해보자.

```
In [70]: a = f(df, arg1 = v1)
    ...: b = g(a,v2,arg3=v3)
    ...: c = h(b,arg4=v4)
```

Series나 DataFrame 객체를 인자로 취하고 반환하는 함수를 사용하는 경우 위 코드를 pipe를 이용해서 아래처럼 고쳐쓸 수 있다.

```
In [71]: result = (df.pipe(f,arg1 = v1)
    ...:           .pipe(g,v2,arg3 = v3)
    ...:           .pipe(h,arg4 = v4))
```

f(df)와 df.pipe(f)는 동일하다. 하지만 pipe는 메서드 연결을 좀 더 쉽게 쓸 수 있게 해준다.

pipe를 이용한 유용한 패턴 중 하나는 일련의 연산을 재사용 가능한 함수로 일반화하는 것이다.

컬럼에서 그룹 평균을 빼는 과정을 생각해보자.

f(df) 와 df.pipe(f) 는 동일하다. 하지만 pipe는 메서드 연결을 좀 더 쉽게 쓸 수 있도록 해준다.

pipe를 이용한 유용한 패턴 중 하나는 일련의 연산을 재사용 가능한 함수로 일반화 하는 것이다. 컬럼에서 그룹 평균을 빼는 과정을 생각해보자.

```
In [72]: g = df.groupby(['key1','key2'])
    ...: df['col1'] = df['col1'] - g.transform('mean')
```

한 컬럼이 아니라 여러 컬럼에 대해 그룹 평균을 뺄 수 있고 그룹의 키를 쉽게 변경할 수 있기바란다면, 또 이 작업을 메서드 연결로도 수행할 수 있기 바란다면 아래 구현 예제를 보자.

```
In [73]: def group_demean(df,by,cols):
    ...:     result = df.copy()
    ...:     g = df.groupby(by)
    ...:     for c in cols:
    ...:         result[c] = df[c] - g[c].transform('mean')
    ...:     return result
```

이제 group_demean 함수를 사용해서 아래처럼 작성할 수 있다.

```
In [74]: result = (df[df.cols < 0]
    ...:           .pipe(group_demean, ['key1','key2'],['col1']))
```