집계와 분류

대용량 데이터 분석의 기본은 효율적으로 요약하는 데 있다. 이는 하나의 값으로 대용량 데이터세트의 기본 특성에 대한 통찰력을 제공하는 sum(), mean(), median(), min(), max()와 같은 집계 연산을 수행하는 것이다. NumPy 배열에서 본 것과 유사한 간단한 연산부터 groupby개념을 기반으로 하는 좀 더 복잡한 연산까지 Pandas에서 제공하는 집계 연산을 살펴본다.

행성 데이터

이번에는 Seaborn패키지를 통해 사용할 수 있는 행성 데이터세트를 사용하겠다. 이데이터는 천문하가자가 다른 별(외계행성 또는 외행성이라고 함) 주변에서 발견한 행성에 대한 정보를 제공한다. 이 데이터는 간단한 Seaborn명령어로 다운로드할 수 있다.

```
In [1]: import pandas as pd
        import numpy as np
In [2]:import seaborn as sns
        planets = sns.load_dataset('planets')
        planets.shape

Out[2]:(1035, 6)
In [3]:planets.head()
Out[3]:
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

이 데이터는 2014년까지 발견된 1,000개 이상의 외계 행성에 대한 세부 정보를 담고 있다.

Pandas의 간단한 집계 연산

1차원 NumPy배열에서와 마찬가지로 Pandas Series에 대해 이 집계 연산들은 하나의 값을 반환한다.

```
In [4]:rng = np.random.RandomState(42)
        ser = pd.Series(rng.rand(5))
        ser
Out[4]:0    0.374540
       1    0.950714
       2    0.731994
       3    0.598658
       4    0.156019
       dtype: float64
```

```
In [5]:ser.sum()
Out[5]:2.8119254917081569
In [6]:ser.mean()
Out[6]:0.56238509834163142
```

DataFrame의 경우, 집계 함수는 기본적으로 각 열 내의 결과를 반환한다.

```
In [7]:df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
        df
Out[7]:
```

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

```
In [8]:df.mean()
Out[8]:A    0.477888
       B    0.443420
       dtype: float64
```

axis 인수를 지정하면 각 행에 대해 집계할 수 있다.

```
In [9]:df.mean(axis='columns')
Out[9]:0    0.088290
       1    0.513997
       2    0.849309
       3    0.406727
       4    0.444949
       dtype: float64
```

Pandas Series와 DataFrame은 "집계: 최소값 , 최대값, 그리고 그 사이의 모든 것"에서 언급했던 일반적인 집계 연산을 모두 포함하고 있으며, 그 밖에도 각 열에 대한 여러 일반적인 집계를 계산하고 그 결과를 반환하는 편리한 describe()메소드가 있다. 행성 데이터에 이 메서드를 사용해 누락된 값이 있는 행을 삭제한다.

```
In [10]:planets.dropna().describe()
Out[10]:
```

	number	orbital_period	mass	distance	year
Count	498.00000	498.000000	498.000000	498.000000	498.000000
Mean	1.73494	835.778671	2.509320	52.068213	2007.377510
Std	1.17572	1469.128259	3.636274	46.596041	4.167284

Min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
Max	6.00000	17337.500000	25.000000	354.000000	2014.000000

이것은 데이터세트의 전반적인 속성을 이해하기 시작할 때 유용한 방법이다. 예를 들면 year열을 보면 외계 행성이 1989년 처럼 오래전에는 발견됐지만 알려진 외계 행성 중 절반은 2010년이나 그 이후까지 발견되지 않은 상태였다는 사실을 알 수 있다. 이는 다른 별들 때문에 가려지는 행성을 찾기 위해 특별히 설계된 우주 망원경인 케플러 미션(Kepler mission)덕택이다.

Pandas 에서 기본으로 제공하는 집계 연산을 요약한 내용이다.

집계연산	설명
count()	항목 전체 차수
first(), last()	첫 항목과 마지막 항목
mean(), median()	평균값과 중앙값
min(), max()	최소값과 최대값
std(), var()	표준편차와 분산
mad()	절대 평균값
prod()	전체 항목의 곱
sum()	전체 항목의 합

.

이것들은 모두 DataFrame과 Series 객체에서 제공하는 메서드이다. 하지만 데이터를 더 깊이 살펴보려면 간단한 집계 연산만으로는 충분하지 않은 경우가 많다. 데이터 요약의 다음 단계로 데이터 부분 집합별로 빠르고 효율적으로 집계를 계산할 수 있는 groupby연산이 있다.

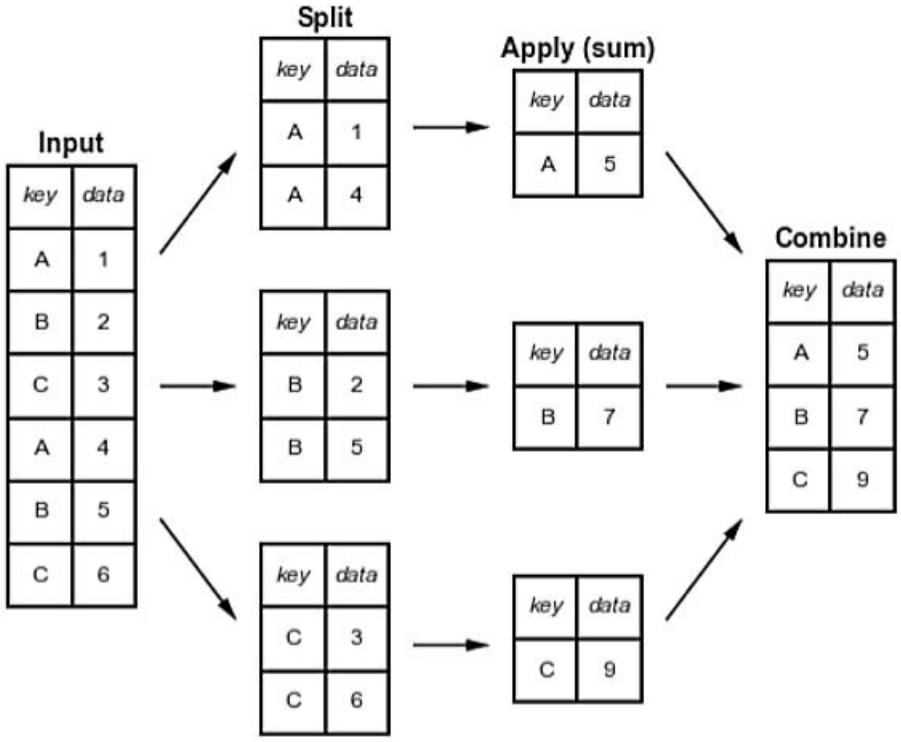
GroupBy: Split(분할), Apply(적용), Combine(결합)

간단한 집계는 데이터세트의 전반적인 특성을 알려주지만, 때에 따라서는 어떤 레이블이나 인덱스를 기준으로 조건부로 집계하고 싶은 경우가 있다. 이러한 작업은 groupby라는 연산으로 구현한다. 'groupby'라는 명칭은 SQL데이터베이스 언어의 명령어에서 유래했지만, R분석 권위자인 해들리 위컴이 최초로 고안한 용어인 분할, 적용, 결합으로 생각하면 더 이해하기 빠를 것이다.

Split(분할), apply(적용), combine(결합)

분할-적용-결합 연산의 고전적인 예를 아래 그림으로 나타냈다. 여기서 '적용'은 요약 집계를 말한다.

- 분할 단계 - 지정된 키 값을 기준으로 DataFrame을 나누고 분류하는 단계
- 적용 단계 - 개별 그룹 내에서 일반적으로 집계, 변환, 필터링 같은 함수를 계산한다.
- 결합 단계 - 이 연산의 결과를 결과 배열에 병합한다.



물론 이 작업을 앞에서 다룬 마스킹, 집계, 병합 명령어의 조합을 사용해 직접 수행할 수 있지만, 중간 단계의 분할은 명시적으로 설명할 필요가 없다는 사실을 깨닫는 것이 중요하다. 오히려 GroupBy는 데이터를 한 번에 처리해 각 그룹의 합계나 평균, 개수, 최소값을 비롯한 다른 집계를 구할 수 있다. GroupBy의 힘은 이 단계들을 추상화한다는 데 있다. 사용자는 이 계산이 내부에서 어떻게 수행되는지에 대해 신경 쓸 필요 없이 전체 차원에서의 연산만 생각하면 된다.

```
In [11]:df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])
         df
Out[11]:
```

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

DataFrame의 groupby() 메서드에 원하는 키 열의 이름을 전달해 가장 기본적인 분할 - 적용 - 결합 연산을 계산할 수 있다.

```
In [12]:df.groupby('key')
Out[12]:<pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

여기서 DataFrame의 집합이 아니라 DataFrameGroupBy 객체가 반환된다는 데 주목하라. 이 객체는 그룹을 세부적으로 조사할 만반의 준비는 돼 있으나 집계 로직이 적용되기까지는 사실상 아무 계산도 하지 않는 DataFrame의 특별한 뷰로 생각하면 된다. 이 '게으른 평가(lazyevaluation)'방식은 일반 집계 연산이 사용자에게 거의 투명한 방식으로 매우 효율적으로 구현될 수 있음을 의미한다.:

집계 연산을 이 DataFrameGroupBy객체에 적용하면 적절한 적용/결합 단계를 수행해 예상한 결과를 만들어낼 것이다.

```
In [13]:df.groupby('key').sum()
Out[13]:
```

key	data
A	3
B	5
C	7

sum()메서드는 하나의 예일 뿐이고, 이어지는 내용에서 보듯이 사실상 Pandas나 NumPy의 집계 함수를 비롯해 모든 유효한 DataFrame연산을 적용할 수 있다.

GroupBy 객체

GroupBy객체는 매우 유연한 추상화다. 여러 면에서 이 객체는 단순히 DataFrame컬렉션처럼 취급할 수 있으며 내부적으로 어려운 일들을 처리한다.
GroupBy에서 사용할 수 있는 가장 중용한 연산은 집계, 필터, 변환, 적용일 것이다.

Column 인덱싱

GroupBy객체는 DataFrame과 동일한 방식으로 열 인덱싱을 지원하며 수정된 GroupBy객체를 반환한다:

```
In [14]:planets.groupby('method')
Out[14]:<pandas.core.groupby.DataFrameGroupBy object at 0x1172727b8>
In [15]:planets.groupby('method')['orbital_period']
Out[15]:<pandas.core.groupby.SeriesGroupBy object at 0x117272da0>

In [16]:planets.groupby('method')['orbital_period'].median()
Out[16]:method
        Astrometry                     631.180000
        Eclipse Timing Variations     4343.500000
        Imaging                      27500.000000
        Microlensing                  3300.000000
        Orbital Brightness Modulation    0.342887
```

```
Pulsar Timing                    66.541900
Pulsation Timing Variations    1170.000000
Radial Velocity                 360.200000
Transit                           5.714932
Transit Timing Variations        57.011000
Name: orbital_period, dtype: float64
```

이 결과를 통해 각 방법이 예민하게 감지해내는 궤도 주기(일 단위)가 일반적으로 어느 정도인지 알 수 있다.

Group 내의 반복¶

GroupBy 객체는 그룹을 직접 순회할 수 있도록 지원하며, 각 그룹을 Series나 DataFrame으로 반환한다.

```
In [17]:for (method, group) in planets.groupby('method'):
             print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

이 방법이 특정 작업을 직접 수행할 때는 유용할 수 있지만, 잠시 후에 논의할 내장된 apply 기능을 사용하는 것이 대체로 훨씬 더 빠르다.

Dispatch methods(디스패치 메서드)

GroupBy객체가 명시적으로 구 현하지 않은 메서드는 그것이 DataFrame객체든 Series객체든 상관없이 일부 파이썬 클래스 매직을 통해 그 그룹에 전달되고 호출될 것이다. 예를들면 DataFrame의 describe()메서드를 데이터의 각 그룹을 설명하는 일련의 집계 연산을 수행할 수 있다.

```
In [18]:planets.groupby('method')['year'].describe().unstack()
```

Out[18]:

	count	mean	std	min	25%	50%	75%
Method							
Astrometry	2.0	2011.500000	2.121320	2010.0	2010.75	2011.5	2012.
Eclipse Timing Variations	9.0	2010.000000	1.414214	2008.0	2009.00	2010.0	2011.
Imaging	38.0	2009.131579	2.781901	2004.0	2008.00	2009.0	2011.
Microlensing	23.0	2009.782609	2.859697	2004.0	2008.00	2010.0	2012.
Orbital Brightness Modulation	3.0	2011.666667	1.154701	2011.0	2011.00	2011.0	2012.
Pulsar Timing	5.0	1998.400000	8.384510	1992.0	1992.00	1994.0	2003.
Pulsation Timing Variations	1.0	2007.000000	NaN	2007.0	2007.00	2007.0	2007.

Radial Velocity	553.0	2007.518987	4.249052	1989.0	2005.00	2009.0	2011.
Transit	397.0	2011.236776	2.077867	2002.0	2010.00	2012.0	2013.
Transit Timing Variations	4.0	2012.500000	1.290994	2011.0	2011.75	2012.5	2013.

이 표를 보면 데이터를 더 잘 이해할 수 있다. 예를 들어 대다수의 행성이 시선 속도법과 통과법에 의해 벌견됐지만, 후자의 방법은 정확성이 향상된 망원경 덕분에 10년전쯤에야 보편화됐다. 최신 방법은 2011년까지 새로운 행성을 발견하는 데 사용된 적이 없는 통과 시점 변화(Transit Timing Variations) 및 궤도 밝기 변조(OrBital Brightness Modulation)방법일 것이다.

이 예제는 디스패치 메서드 사용법에 대한 하나의 예시일 뿐이다. 디스패치 메서드는 각 개별 그룹에 적용되고 그 결과는 GroupBy 내에서 결합돼 반환된다는 사실을 알아두자. 모든 유효한 DataFrame/Series메서드는 상응하는 GroupBy 객체에 사용되어 몇가지 매우 유연하고 강력한 연산을 수행할 수 있다.

Aggregate(집계), filter(필터), transform(변환), apply(적용)¶

GroupBy 객체에는 그룹 데이터를 결합하기 전에 여러 유용한 연산을 효율적으로 구현하는 aggregate(), filter(), transform(),apply()메서드가 있다.:

In [19]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                  columns = ['key', 'data1', 'data2'])
df
```

Out[19]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation(집계)

sum(), median()등을 사용하는 GroupBy집계 연산에 익숙해졌지만 aggregate()메서드가 훨씬 더 많은 유연성을 제공한다. 이 메서드는 문자열, 함수, 리스트 등을 취해 한 번에 모든 집계를 계산할 수 있다

In [20]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

Out[20]:

	data1			data2		
	min	**median**	**max**	**min**	**median**	**max**

key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

다른 유용한 패턴은 열 이름을 해당 열에 적용될 연산에 매핑하는 딕셔너리를 전달하는 것이다.:

```
In [21]:df.groupby('key').aggregate({'data1': 'min',
                                     'data2': 'max'})
```

Out[21]:

key	data1	data2
A	0	5
B	1	7
C	2	9

Filtering(필터링)

필터링 연산을 사용하려면 그룹 속성을 기준으로 데이터를 걸러낼 수 있다. 예를 들어, 표준 편차가 어떤 임계 값보다 큰 그룹을 모두 유지할 수 있다.

```
In [22]:def filter_func(x):
            return x['data2'].std() > 4
        print(df); print(df.groupby('key').std()); print(df.groupby('key').filter(filter_func))
```

Out[22]:df

	Key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').std()
```

key	data1	data2
A	2.12132	1.414214
B	2.12132	4.949747

C	2.12132	4.242641

```
df.groupby('key').filter(filter_func)
```

	Key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

Filter()함수는 그룹이 필터링을 통과하는지 아닌지를 지정하는 부울 값을 변환한다. 여기서는 그룹A의 표준편차가 4보다 작으므로 결과에서 그 그룹이 제거된다.

Transformation(변환)

집계는 데이터의 축소 버전을 반환해야 하지만, 재결합을 위해 데이터의 변환된 버전을 반환할 수 있다. 그러한 변환의 결과는 입력과 같은 형상을 가진다. 일반적인 예로 데이터에서 그룹별 평균을 빼서 데이터를 중앙에 정렬하는 것을 들 수 있다.

```
In [23]:df.groupby('key').transform(lambda x: x - x.mean())
```

Out[23]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

apply()메서드

apply() 메서드는 임의의 함수를 그룹 결과에 적용할 때 사용한다. 이 함수는 DataFrame을 취해 Pandas객체(즉, DataFrame, Series)나 스칼라를 반환한다. 결합 연산은 반환된 출력값 유형에 따라 조정된다.

다음 예제에서 사용한 apply()는 첫번째 열을 두번째 열의 합계로 정규화한다.:

```
In [24]:def norm_by_data2(x):
            # x는 그룹 값을 가지는 DataFrame
            x['data1'] /= x['data2'].sum()
            return x
        print(df)
        print(df.groupby('key').apply(norm_by_data2))
```

Out[24]:df

	Key	data1	data2

0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').apply(norm_by_data2)
```

	Key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

GroupBy 내에서 apply()는 상당히 유연하다. 함수가 DataFrame을 취하고 Pandas객체나 스칼라를 반환한다는 것이 유일한 규칙이다. 그 중간에 무엇을 하든지 상관없다.

분할 키 지정하기

분할 키를 제공하는 리스트, 배열, 시리즈, 인덱스 키

DataFrame의 길이와 일치하는 길이의 시리즈나 리스트일 수 있다.

```
In [25]:L = [0, 1, 0, 1, 2, 0]
         print(df); print(df.groupby(L).sum())
Out[25]:df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby(L).sum()
```

	data1	data2
0	7	17
1	4	3
2	4	7

이전의 df.groupby('key')를 구현하는 더 자세한 방법이 있다.

```
In [26]:print(df); print(df.groupby(df['key']).sum())
Out[26]:df
```

	Key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby(df['key']).sum()
```

key	data1	data2
A	3	8
B	5	7
C	7	12

인덱스를 그룹에 매핑한 딕셔너리나 시리즈

딕셔너리나 시리즈 또 다른 방법은 인덱스 값을 그룹 키에 매핑하는 딕셔너리를 제공하는 것이다.

```
In [27]:df2 = df.set_index('key')
        mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
        print(df2);print(df2.groupby(mapping).sum())
Out[27]:df2
```

key	data1	data2
A	0	5
B	1	0

C	2	3
A	3	3
B	4	7
C	5	9

```
df2.groupby(mapping).sum()
```

	data1	data2
consonant	12	19
Vowel	3	8

파이썬 함수

매핑과 유사하게 인덱스 값을 입력해서 그룹을 출력하는 파이썬 함수를 전달하면 된다.:

In [28]:print(df2); print(df2.groupby(str.lower).mean())

Out[28]:df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

```
df2.groupby(str.lower).mean()
```

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

유효한 키의 리스트

아울러 앞에서 다룬 모든 키 선택 방식은 다중 인덱스에서 그룹에 결합할 수 있다.

In [29]:df2.groupby([str.lower, mapping]).mean()

Out[29]:

		data1	data2

a	**vowel**	1.5	4.0
b	**consonant**	2.5	3.5
c	**consonant**	3.5	6.0

분류(Grouping) 예제

이에 대한 예제로 , 파이썬 코드 몇 줄에 이 모든 것을 집어 넣어 방법 및 연대별로 발견된 행성의 개수를 셀 수 있다.

```
In [30]:decade = 10 * (planets['year'] // 10)
         decade = decade.astype(str) + 's'
         decade.name = 'decade'
         planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

Out[30]:

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

이 코드는 현실적인 데이터세트를 살펴볼 때 지금까지 논의했던 다양한 연산을 결합했을 때의 힘을 보여준다. 지난 수십년 동안 언제 어떻게 행성을 발견했는지에 대해 바로 이해할 수 있다.
이 코드 몇 줄을 자세히 살펴보면서 결과가 나오기까지 정확히 어떤 작업을 하는지 이해하기 위해 단계별로 평가해 볼 것을 권한다. 다소 복잡한 예제인 것은 확실하지만, 이 코드를 이해하고 나면 자신의 데이터를 비슷하게 탐색할 수 있는 방법을 알게 될 것이다.