

Pandas_Advanced

류영표

Pandas Advanced

기술 통계 계산과 요약

정렬

함수의 적용과 맵핑 데이터

정제 및 준비

기술 통계 계산과 요약

기술 통계 계산과 요약

- 기술 통계 계산과 요약
- head, tail
- describe
- max, min, sum
- unique, value_counts

기술 통계 계산과 요약

메서드	설명
head, tail	series나 dataframe의 몇 개 데이터만 보여줌
describe	series나 dataframe의 각 컬럼에 대한 요약 통계
count	na 값을 제외한 값의 개수를 반환
min, max	최소값, 최대값
argmin, argmax	최소값, 최대값을 가진 색인의 위치(정수)를 반환
idxmin, idxmax	최소값, 최대값을 가진 색인의 값을 반환
sum	합
mean, median	평균, 중앙값
var, std	분산, 표준편차

기술 통계 계산과 요약

- head / tail

- head : 위에서부터 일정한 개수만큼의 데이터만 보여줌(default = 5)
- tail : 아래에서부터 일정한 개수만큼의 데이터만 보여줌(default = 5)

df.head(개수)

df.tail(개수)



```
stock.head()
```



	Date	kospi	kosdaq	gold_fut_132030	Bond_273130
0	2020. 1. 2 오후 3:30:00	2175.17	674.02	10845	108215
1	2020. 1. 3 오후 3:30:00	2176.46	669.93	11000	108565
2	2020. 1. 6 오후 3:30:00	2155.07	655.31	11245	108745
3	2020. 1. 7 오후 3:30:00	2175.54	663.44	11180	108400
4	2020. 1. 8 오후 3:30:00	2151.31	640.94	11360	108270

기술 통계 계산과 요약

- describe

df.describe()

[13] frame



	c1	c2	c3
i1	10.0	4.0	a
i2	10.0	2.5	b
i3	NaN	10.0	a



frame.describe()



	c1	c2
count	2.0	3.000000
mean	10.0	5.500000
std	0.0	3.968627
min	10.0	2.500000
25%	10.0	3.250000
50%	10.0	4.000000
75%	10.0	7.000000
max	10.0	10.000000

→ 데이터 타입이 섞여있다면
수치형 데이터로 이루어진
컬럼만 요약 제공

기술 통계 계산과 요약

- describe

```
[13] frame
```



	c1	c2	c3
i1	10.0	4.0	a
i2	10.0	2.5	b
i3	NaN	10.0	a

```
[8] frame.describe(include='object')
```



	c3
count	3
unique	2
top	a
freq	2

→ 인자로 include='object' 를 주면
object에 대한 요약 통계를 보여줌

기술 통계 계산과 요약

- describe

```
[13] frame
```



	c1	c2	c3
i1	10.0	4.0	a
i2	10.0	2.5	b
i3	NaN	10.0	a



```
frame.describe(include='all')
```



	c1	c2	c3
count	2.0	3.000000	3
unique	NaN	NaN	2
top	NaN	NaN	a
freq	NaN	NaN	2
mean	10.0	5.500000	NaN
std	0.0	3.968627	NaN
min	10.0	2.500000	NaN
25%	10.0	3.250000	NaN
50%	10.0	4.000000	NaN
75%	10.0	7.000000	NaN
max	10.0	10.000000	NaN

→ 인자로 include='all'을 주면 전체에 대한 요약 통계를 보여줌.

기술 통계 계산과 요약

- max/min/sum

```
[13] frame
```

```
↳
```

	c1	c2	c3
i1	10.0	4.0	a
i2	10.0	2.5	b
i3	NaN	10.0	a

```
[9] frame.max()
```

```
↳ c1    10  
   c2    10  
   c3     b  
   dtype: object
```

→ 축을 지정하지 않으면
기본적으로 row을 기준으로
연산을 함

```
[10] frame.min(axis=1)
```

```
↳ i1     4.0  
   i2     2.5  
   i3    10.0  
   dtype: float64
```

→ 컬럼을 기준으로 연산을 하
고 싶으면 axis='columns'
또는 axis='1' 을
인자로 넘겨주 면 됨

기술 통계 계산과 요약

- unique
 - 중복되는 값을 제거하고 유일값만 담고 있는 **Series**를 반환

series.unique()

[1] obj

```
0    2.0
1    1.0
2    3.0
3    3.0
4    1.0
5    5.0
6    NaN
7    1.0
8    2.0
dtype: float64
```

[19] obj.unique()

```
array([ 2.,  1.,  3.,  5., nan])
```

→ 순서를 정렬해서 반환하지 않음

기술 통계 계산과 요약

- value_counts

- 값을 인덱스(라벨)로 하고 그 값의 개수를 담고 있는 **Series**를 반환

```
[ ] obj
```

```
[ ] 0    2.0  
    1    1.0  
    2    3.0  
    3    3.0  
    4    1.0  
    5    5.0  
    6    NaN  
    7    1.0  
    8    2.0  
dtype: float64
```

```
[20] obj.value_counts()
```

```
[ ] 1.0    3  
    3.0    2  
    2.0    2  
    5.0    1  
dtype: int64
```

→ 값에 대해 내림차순으로 정렬하여 반환



```
# normalize  
obj.value_counts(normalize=True)
```

```
[ ] 1.0    0.375  
    3.0    0.250  
    2.0    0.250  
    5.0    0.125  
dtype: float64
```

정렬

정렬

- Series
- DataFrame

Series

- Series.sort_index
 - index를 기준으로 정렬

`series.sort_index(axis=0, ascending=True, na_position='last')`

- **axis** : 정렬 기준 축, 이전과 마찬가지로 0('index')과 1('column')가 있음
- **ascending** : True이면 오름차순, False이면 내림차순으로 정렬
- **na_position** : 정렬시 NaN값의 위치 지정, {'first', 'last'}

Series

- Series.sort_index
 - index를 기준으로 정렬

[5] obj

```
[>] a    1  
     d    2  
     e    3  
     b   -1  
     c   -2  
     dtype: int64
```

[6] obj.sort_index()

```
[>] a    1  
     b   -1  
     c   -2  
     d    2  
     e    3  
     dtype: int64
```


Series

- Series.sort_index
- index를 기준으로 정렬

```
[5] obj
```

```
[>] a    1  
     d    2  
     e    3  
     b   -1  
     c   -2  
     dtype: int64
```

```
[7] obj.sort_index(ascending=False)
```

```
[>] e    3  
     d    2  
     c   -2  
     b   -1  
     a    1  
     dtype: int64
```

Series

- Series.sort_values
 - 값을 기준으로 정렬

`series.sort_values(axis=0, ascending=True, na_position='last')`

- `axis` : 정렬 기준 축, 이전과 마찬가지로 0('index')과 1('column')가 있음
- `ascending` : True이면 오름차순, False이면 내림차순으로 정렬
- `na_position` : 정렬시 NaN값의 위치 지정, {'first', 'last'}

Series

- Series.sort_values
 - 값을 기준으로 정렬

```
[13] obj
```

```
↳ 0    10.0  
   1     NaN  
   2    20.0  
   3     0.0  
   4     NaN  
   dtype: float64
```

```
[10] obj.sort_values()
```

```
↳ 3     0.0  
   0    10.0  
   2    20.0  
   1     NaN  
   4     NaN  
   dtype: float64
```

Series

- Series.sort_values
 - 값을 기준으로 정렬

```
[13] obj
```

```
0    10.0  
1     NaN  
2    20.0  
3     0.0  
4     NaN  
dtype: float64
```

```
[11] obj.sort_values(ascending=False)
```

```
2    20.0  
0    10.0  
3     0.0  
1     NaN  
4     NaN  
dtype: float64
```

Series

- Series.sort_values
 - 값을 기준으로 정렬

```
[13] obj
```

```
0    10.0  
1     NaN  
2    20.0  
3     0.0  
4     NaN  
dtype: float64
```

```
[▶] obj.sort_values(na_position='first')
```

```
1     NaN  
4     NaN  
3     0.0  
0    10.0  
2    20.0  
dtype: float64
```

DataFrame

- df.sort_index
 - index를 기준으로 정렬

**df.sort_index(*by*, *axis=0*, *ascending=True*,
na_position='last')**

- *by* : 여러개의 컬럼이나 인덱스를 정렬할 때 유용한 인자로, 정렬 기준
- *axis* : 정렬 기준 축, 이전과 마찬가지로 0('index')과 1('column')가 있음
- *ascending* : True이면 오름차순, False이면 내림차순으로 정렬
- *na_position* : 정렬시 NaN값의 위치 지정, {'first', 'last'}

DataFrame

- `df.sort_index`
 - index를 기준으로 정렬

[16] frame



	e	d	f
a	0	1	2
c	3	4	5
b	6	7	8

[17] frame.sort_index()



	e	d	f
a	0	1	2
b	6	7	8
c	3	4	5

DataFrame

- `df.sort_index`
 - index를 기준으로 정렬

[16] frame



	e	d	f
a	0	1	2
c	3	4	5
b	6	7	8



`frame.sort_index(axis=1)`



	d	e	f
a	1	0	2
c	4	3	5
b	7	6	8

DataFrame

- df.sort_index
 - index를 기준으로 정렬

[21] frame



	a	b
0	5	2
1	4	0
2	10	3
3	10	1
4	8	4



frame.sort_values(by='a', ascending = False)



	a	b
2	10	3
3	10	1
4	8	4
0	5	2
1	4	0

→ by에서 정해진 값을 기준으로 정렬

DataFrame

- df.sort_index
 - index를 기준으로 정렬

[21] frame



	a	b
0	5	2
1	4	0
2	10	3
3	10	1
4	8	4



```
frame.sort_values(by=['a', 'b'], ascending = [False, True])
```



	a	b
3	10	1
2	10	3
4	8	4
0	5	2
1	4	0

→ by에 여러 개의 기준을 주고 싶으면 리스트로 인자를 넘김
리스트에 있는 순서가 우선순위가 되며, 이에 상응하여 ascending인자도 multi-value를 줄 수 있음

함수 적용과 맵핑

함수 적용과 맵핑

- Series
- DataFrame

Series

- map

map은 **series**의 **각각의 element**들을 다른 어떤 값으로 대체하는 역할

series.map(arg, na_action=None)

- **arg** : 대체할 값으로 function, dictionary, series 등
- **na_action** : NaN값이 존재할 경우 어떻게 작동할지에 대한 인자
{None, 'ignore'} None이라면 NaN에도 작동, 'ignore'은 NaN은 무시

Series

- map

`series.map(arg, na_action=None)`

```
[28] series
```

```
0    100
1    200
2    300
dtype: int64
```

```
series.map({100:'C', 200:'B', 300:'A'})
```

```
0    C
1    B
2    A
dtype: object
```

→ `arg`가 `dict`인 경우, 원본 `series`에 있는 값을 `key`로 바꿔 줄 값을 `value`로 설정
만일, 원본 데이터에 있는 값이 `dict`에 빠졌을 경우에는, `NaN`으로 대체

Series

- map

`series.map(arg, na_action=None)`

```
[28] series
```

```
0    100  
1    200  
2    300  
dtype: int64
```

→ arg에 함수도 가능

```
[25] series.map('${}'.format)
```

```
0    $100  
1    $200  
2    $300  
dtype: object
```

```
[26] series.map('{}달러'.format)
```

```
0    100달러  
1    200달러  
2    300달러  
dtype: object
```

Series

- map

`series.map(arg, na_action=None)`

```
[28] series
```

```
0    100  
1    200  
2    300  
dtype: int64
```

```
# lambda  
f = lambda x: np.add(x, 3)  
series.map(f)
```

```
0    103  
1    203  
2    303  
dtype: int64
```


Series

- apply

어떠한 함수를 **series**의 각각의 **element**에 적용시켜주는 함수
map함수 보다 적용할 수 있는 함수의 범위가 넓음

series.apply(func, args, **kwargs)

- **func** : series에 적용될 함수
- **args** : series를 제외한 함수에 들어갈 다른 매개변수
- ****kwargs** : 함수에 넘겨줄 키워드 인자들

Series

- apply

`series.apply(func, args, **kwds)`

```
[9] s
```

```
↳ London      20  
   New York    21  
   Helsinki    12  
   dtype: int64
```

```
[11] def sub_custom_value(x, val) :  
      return x - val
```

```
[13] s.apply(sub_custom_value, args=(10,))
```

```
↳ London      10  
   New York    11  
   Helsinki     2  
   dtype: int64
```

Series

- apply

`series.apply(func, args, **kwds)`

```
[9] s
```

```
↳ London      20  
   New York    21  
   Helsinki    12  
   dtype: int64
```

```
[14] def add_custom_values(x, **kwargs):  
      for month in kwargs:  
          x += kwargs[month]  
      return x
```

```
[▶] s.apply(add_custom_values, june=30 , july=20, august=25)
```

```
↳ London      95  
   New York    96  
   Helsinki    87  
   dtype: int64
```

DataFrame

- apply
축별로 함수를 적용

`df.apply(func, axis)`

- `func` : df에 적용될 함수
- `axis` : 함수가 적용될 기준 축

DataFrame

- apply

`df.apply(func, axis)`

```
[18] frame
```

```
↗
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
[22] frame.apply(lambda x: x.max()-x.min())
```

```
↗
```

a	8
b	8
c	8
d	8

dtype: int64

DataFrame

- apply

`df.apply(func, axis)`

```
[18] frame
```

```
↗
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
[23] frame.apply(lambda x: x.max()-x.min(), axis=1)
```

```
↗
```

0	3
1	3
2	3

dtype: int64

DataFrame

- applymap

모든 원소에 원소별로 함수를 적용

`df.applymap(func)`

```
[18] frame
```

```
↗
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
[24] frame.applymap(lambda x: x**2)
```

```
↗
```

	a	b	c	d
0	0	1	4	9
1	16	25	36	49
2	64	81	100	121

데이터 정제 및 준비

데이터 정제 및 준비

- 데이터 삭제
- 데이터 병합
- missing data
- 데이터 변형
- groupby

데이터 삭제 - drop

- drop

row나 column에서 특정한 label을 삭제하는 함수

[31] frame



	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

[32] frame.drop('r1')



	c1	c2	c3	c4
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15


→ 기본 axis = 0

데이터 삭제 - drop

- drop
row나 column에서 특정한 label을 삭제하는 함수

[31] frame

	c1	c2	c3	c4
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

 frame.drop('c2', axis = 1)

	c1	c3	c4
r1	0	2	3
r2	4	6	7
r3	8	10	11
r4	12	14	15

→ axis를 명시해서 column을 삭제할 수도 있음

데이터 삭제 - drop

- drop
row나 column에서 특정한 label을 삭제하는 함수

```
[31] frame
```

	c1	c3	c4
r1	0	2	3
r2	4	6	7
r3	8	10	11
r4	12	14	15

```
[34] frame.drop(columns=['c3', 'c4'])
```

	c1	c2
r1	0	1
r2	4	5
r3	8	9
r4	12	13

→ columns라는 인자에 리스트로 label값을 넘기면 여러개의 label을 한번에 지울 수 있음
(axis 설정이 따로 필요없음)

데이터 삭제 - drop

- drop

row나 column에서 특정한 label을 삭제하는 함수

[31] frame



	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

[35] # inplace

```
frame.drop(['r2'], inplace=True)  
frame
```



	c1	c2	c3	c4
r1	0	1	2	3
r3	8	9	10	11
r4	12	13	14	15

→ inplace 인자를 이용할 수 있음

데이터 삭제 - drop

- drop
row나 column에서 특정한 label을 삭제하는 함수

[31] frame



	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15



할당

```
frame = frame.drop(['r3'])  
frame
```



	c1	c2	c3	c4
r1	0	1	2	3
r4	12	13	14	15

→ inplace 인자를 이용하지 않고, 직접 할당

데이터 병합- concat

- concat

기준이 되는 축을 따라 객체를 이어 붙이는 함수

pd.concat(objs, **axis**, **join='outer'**)

- **axis** : 기준이 되는 축
- **join** : 어떠한 방식으로 이어 붙일 지 {'outer', 'inner'}

참고) Outer : 합집합으로 DataFrame 합치기
innter : 교집합으로 DataFrame 합치기

데이터 병합- concat

- concat

pd.concat(objs, **axis**, **join='outer'**)

```
[46] print(s1, s2, s3, sep='\n\n')
```

```
↳ c    100  
   b    200  
   dtype: int64  
  
   c    300  
   d    300  
   e    300  
   dtype: int64  
  
   f    500  
   g    600  
   dtype: int64
```

```
[38] pd.concat([s1, s2, s3])
```

```
↳ c    100  
   b    200  
   c    300  
   d    300  
   e    300  
   f    500  
   g    600  
   dtype: int64
```

→ axis 0 을 따라 쪽 이어붙이게 됨

데이터 병합- concat

- concat

`pd.concat(objs, axis, join='outer')`

```
[46] print(s1, s2, s3, sep='\n\n')
```

```
↳ c    100  
   b    200  
   dtype: int64  
  
   c    300  
   d    300  
   e    300  
   dtype: int64  
  
   f    500  
   g    600  
   dtype: int64
```

```
▶ pd.concat([s1, s2], axis=1)
```

```
↳
```

	0	1
c	100.0	300.0
b	200.0	NaN
d	NaN	300.0
e	NaN	300.0

→ axis 1을 따라 쪽 이어붙이게 됨.
즉, 새로운 컬럼이 생기고 dataframe이 됨

데이터 병합 - concat

- concat

[47] data1

	id	col1	col2
0	01	43	1918
1	02	17	1103
2	03	31	1269
3	04	12	1991
4	05	19	1853
5	06	1	1127

[48] data2

	id	col1
0	04	3876
1	05	2076
2	06	1509
3	07	4533

[49] pd.concat([data1, data2])



	id	col1	col2
0	01	43	1918.0
1	02	17	1103.0
2	03	31	1269.0
3	04	12	1991.0
4	05	19	1853.0
5	06	1	1127.0
0	04	3876	NaN
1	05	2076	NaN
2	06	1509	NaN
3	07	4533	NaN

데이터 병합 - concat

- concat

[47] data1

	id	col1	col2
0	01	43	1918
1	02	17	1103
2	03	31	1269
3	04	12	1991
4	05	19	1853
5	06	1	1127

▶ data2

	id	col1
0	04	3876
1	05	2076
2	06	1509
3	07	4533

▶ pd.concat([data1, data2], axis=1)

	id	col1	col2	id	col1
0	01	43	1918	04	3876.0
1	02	17	1103	05	2076.0
2	03	31	1269	06	1509.0
3	04	12	1991	07	4533.0
4	05	19	1853	NaN	NaN
5	06	1	1127	NaN	NaN

데이터 병합- merge

- merge
key를 이용해 데이터의 row를 기준으로 연결시켜 합침. (sql의 join과 유사)

pd.merge(objs, how='inner', on)

- objs : 병합할 자료
- how : 어떤 방식으로 병합할 것인가 {'inner', 'outer', 'left', 'right'}
- on : 어떤 label을 기준으로 병합할 것인가, 기본적으로 중복되는 컬럼을 기준으로 병합을 함

데이터 병합 - merge

- merge

`pd.merge(objs, how='inner', on)`

[6] data1

	id	col1	col2
0	01	5	1083
1	02	29	1548
2	03	6	1594
3	04	16	1763
4	05	7	1305
5	06	20	1070

data2

	id	col1
0	04	4532
1	05	2599
2	06	4556
3	07	3547

```
#inner join  
pd.merge(data1, data2, on='id')
```

	id	col1_x	col2	col1_y
0	04	16	1763	4532
1	05	7	1305	2599
2	06	20	1070	4556

데이터 병합 - merge


- merge

pd.merge(objs, how='inner', on)

- 중복되는 키가 하나도 없다면? left_on, right_on으로 각각의 자료에서 컬럼의 키로 쓸 이름을 지정해줄 수 있음

[13] data1

	lkey	value
0	a	1
1	b	2
2	c	3
3	d	5

 data2

	rkey	value
0	d	5
1	e	6
2	a	7
3	c	8

`pd.merge(data1, data2, left_on='lkey', right_on='rkey')`

	lkey	value_x	rkey	value_y
0	a	1	a	7
1	c	3	c	8
2	d	5	d	5

Missing data

함수	설명
isnull	누락되거나 NA(not available) 값을 알려주는 불리언 값들이 저장된 객체를 반환
notnull	isnull과 반대되는 메서드
fillna	누락된 데이터에 값을 채우는 메서드. (특정한 값이나 ffill, bfill 같은 보간 메서드 적용)
dropna	누락된 데이터가 있는 축(로우, 컬럼)을 제외시키는 메서드

Missing data

- isnull

isnull()

- 누락되거나 NA인 값을 알려주는 불리언값이 저장된 같은 형의 객체를 반환
- None, np.NaN

```
[17] obj
```

```
↳ 0    apple  
   1    mango  
   2      NaN  
   3     None  
   4    peach  
   dtype: object
```

```
[18] obj.isnull()
```

```
↳ 0    False  
   1    False  
   2     True  
   3     True  
   4    False  
   dtype: bool
```

```
obj.isnull().sum()
```

```
2
```


Missing data

- notnull

notnull()

- missing data가 없다는 것을 파악하여 불리언값이 저장된 같은 형의 객체를 반환
- isnull과 반대되는 메소드

```
[17] obj
```

```
↳ 0    apple  
   1    mango  
   2      NaN  
   3     None  
   4    peach  
   dtype: object
```

```
[19] obj.notnull()
```

```
↳ 0     True  
   1     True  
   2    False  
   3    False  
   4     True  
   dtype: bool
```

Missing data

- drop

dropna(axis=0, how='any', thresh)

- 누락된 데이터가 있는 축(컬럼, 로우)를 제외시키는 메소드
- **axis** : 기준이 되는 축으로 0이면 row, 1이면 columns
- **how** : NA를 찾았을 때, 어떤 방식으로 찾아서 지울 지에 대한 인자 {'any', 'all'}
- **thresh** : 지우는 NA value를 몇 개까지 허용할지에 대한 인자

```
[17] obj
```

```
↳ 0    apple
   1    mango
   2      NaN
   3     None
   4    peach
   dtype: object
```

```
▶ obj.dropna()
```

```
↳ 0    apple
   1    mango
   4    peach
   dtype: object
```

Missing data

- drop

dropna(axis=0, how='any', thresh)

[5] frame

	x1	x2	x3	y
0	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN

[6] frame.dropna()

	x1	x2	x3	y
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0

Missing data

- drop

dropna(axis=0, how='any', thresh)

[5] frame



	x1	x2	x3	y
0	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN



```
# how  
frame.dropna(how='all')
```



	x1	x2	x3	y
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN

```
frame['e'] = np.nan  
frame
```

	x1	x2	x3	y	e
0	NaN	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0	NaN
2	5.0	2.0	30.0	8.0	NaN
3	20.0	NaN	20.0	6.0	NaN
4	15.0	3.0	10.0	NaN	NaN

Missing data


- fillna

fillna(value, **method**='None')

- 누락된 데이터를 대신할 값으로 채우거나, 'ffill'이나 'bfill' 같은 보간 메소드를 적용
- method : 보간 방법 {'ffill', 'bfill', 'backfill', 'pad', None}

[16] frame

	x1	x2	x3	y	e
0	NaN	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0	NaN
2	5.0	2.0	30.0	8.0	NaN
3	20.0	NaN	20.0	6.0	NaN
4	15.0	3.0	10.0	NaN	NaN

 frame.fillna(0)

	x1	x2	x3	y	e
0	0.0	0.0	0.0	0.0	0.0
1	10.0	5.0	40.0	6.0	0.0
2	5.0	2.0	30.0	8.0	0.0
3	20.0	0.0	20.0	6.0	0.0
4	15.0	3.0	10.0	0.0	0.0


Missing data

- fillna

fillna(value, **method**='None')

[16] frame

	x1	x2	x3	y	e
0	NaN	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0	NaN
2	5.0	2.0	30.0	8.0	NaN
3	20.0	NaN	20.0	6.0	NaN
4	15.0	3.0	10.0	NaN	NaN

 frame.fillna({'x1': 10, 'y': 0})

	x1	x2	x3	y	e
0	10.0	NaN	NaN	0.0	NaN
1	10.0	5.0	40.0	6.0	NaN
2	5.0	2.0	30.0	8.0	NaN
3	20.0	NaN	20.0	6.0	NaN
4	15.0	3.0	10.0	0.0	NaN

→ 컬럼 이름을 키로, 대체하고자하는 값을 **value**로 하는 **dict**를 넘겨주는 각 컬럼에 매칭하여 값을 변경

데이터 변형

- 중복 제거

1) `uplicated()`: 각 로우가 중복인지(True) 아닌지(False) 알려주는
불리언 `series` 반환

2) `drop_duplicates()`: `uplicated`를 적용한 결과가 `False`인 것들만 모아서
`dataframe` 반환

[22] data

```
↳
```

	id	name	phone
0	0001	a	0
1	0002	b	1
2	0003	c	2
3	0001	a	3

[18] data.duplicated()

```
↳
```

0	False
1	False
2	False
3	True

dtype: bool



data.drop_duplicates()



	id	name
0	0001	a
1	0002	b
2	0003	c

데이터 변형

- 중복 제거

- 1) `uplicated()`: 각 로우가 중복인지(True) 아닌지(False) 알려주는 불리언 series 반환
- 2) `drop_duplicates()`: `uplicated`를 적용한 결과가 False인 것들만 모아서 dataframe 반환

data



	id	name	phone
0	0001	a	0
1	0002	b	1
2	0003	c	2
3	0001	a	3



```
data.drop_duplicates(subset=['id'], keep='last')
```



	id	name	phone
1	0002	b	1
2	0003	c	2
3	0001	a	3

→ **subset** : 중복을 검색할 기준

keep : 중복 자료에서 어떤 것을 남길지를 파악하는 인자

데이터 변형

- 치환

1) replace : 특정 값을 치환하는 함수

```
[23] obj = pd.Series([10, -999, 4, 5, 7, 'n'])
```

```
[24] obj.replace(-999, np.nan)
```

```
0    10
1   NaN
2     4
3     5
4     7
5     n
dtype: object
```

```
[25] obj.replace([-999, 'n'], np.nan)
```

```
0    10.0
1   NaN
2    4.0
3    5.0
4    7.0
5   NaN
dtype: float64
```

→ 한 개 이상의 값도 치환 가능

데이터 변형

- binning

1) cut

```
[26] ages = [20, 35, 67, 39, 59, 44, 56, 77, 28, 20, 22, 80, 32, 46, 52, 19, 33, 5, 15, 50, 29, 21, 33, 48, 85, 80, 31, 10]
```

```
[27] bins = [0, 20, 40, 60, 100]
```

```
[28] cuts = pd.cut(ages, bins)
cuts
```

```
↳ [(0, 20], (20, 40], (60, 100], (20, 40], (40, 60], ..., (40, 60], (60, 100], (60, 100], (20, 40], (0, 20]]
Length: 28
Categories (4, interval[int64]): [(0, 20] < (20, 40] < (40, 60] < (60, 100]]
```

- cut 메소드의 결과는 Categorical이라는 특수한 객체

데이터 변형

- binning

- 1) cut

- cut 메소드의 결과는 Categorical이라는 특수한 객체

```
[29] cuts.categories
```

```
↳ IntervalIndex([(0, 20], (20, 40], (40, 60], (60, 100]],  
                 closed='right',  
                 dtype='interval[int64]')
```

```
[30] cuts.codes
```

```
↳ array([0, 1, 3, 1, 2, 2, 2, 3, 1, 0, 1, 3, 1, 2, 2, 0, 1, 0, 0, 2, 1, 1,  
        1, 2, 3, 3, 1, 0], dtype=int8)
```

데이터 변형

- binning

- 1) cut

- 구간으로 주지않고 숫자로 주는 경우, 구간을 균등한 길이로 나누어줌

```
[36] # 구간을 균등한 길이로 나눔  
pd.cut(ages, 4, precision=1).value_counts()
```

```
↳ (4.9, 25.0]      8  
   (25.0, 45.0]     9  
   (45.0, 65.0]     6  
   (65.0, 85.0]     5  
dtype: int64
```

데이터 변형

- binning

2) qcut

- 개수들이 균등한 비율이 되도록 나누어줌

```
[34] # 개수들이 균등한 비율이 되도록 나눔  
pd.qcut(ages, 4).value_counts()
```

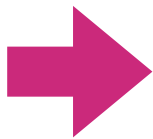
```
↳ (4.999, 21.75]    7  
   (21.75, 34.0]    7  
   (34.0, 53.0]     7  
   (53.0, 85.0]     7  
dtype: int64
```

데이터 변형

- `get_dummies`

- categorical variable(명목형 변수)를 one-hot encoding 해줌

label
dog
cat
apple



dog	cat	apple
1	0	0
0	1	0
0	0	1

데이터 변형

- get_dummies
 - categorical variable(명목형 변수)를 one-hot encoding 해줌

```
[37] df = pd.DataFrame({'col1': [10, 20, 30],  
                        'col2': ['a', 'b', 'a']})
```

df



	col1	col2
0	10	a
1	20	b
2	30	a



```
pd.get_dummies(df)
```



	col1	col2_a	col2_b
0	10	1	0
1	20	0	1
2	30	1	0

데이터 변형

- get_dummies
 - categorical variable(명목형 변수)를 one-hot encoding 해줌

```
df = pd.DataFrame({'col1': ['001', '002', '003', '004', '005', '006'],  
                  'col2': [10, 20, 30, 40, 50, 60],  
                  'col3': ['서울시', '경기도', '서울시', '제주도', '경기도', '서울시']})
```

df

	col1	col2	col3
0	001	10	서울시
1	002	20	경기도
2	003	30	서울시
3	004	40	제주도
4	005	50	경기도
5	006	60	서울시

pd.get_dummies(df)

	col2	col1_001	col1_002	col1_003	col1_004	col1_005	col1_006	col3_경기도	col3_서울시	col3_제주도
0	10	1	0	0	0	0	0	0	1	0
1	20	0	1	0	0	0	0	1	0	0
2	30	0	0	1	0	0	0	0	1	0
3	40	0	0	0	1	0	0	0	0	1
4	50	0	0	0	0	1	0	1	0	0
5	60	0	0	0	0	0	1	0	1	0

groupby

- 그룹연산은 분리-적용-결합의 과정

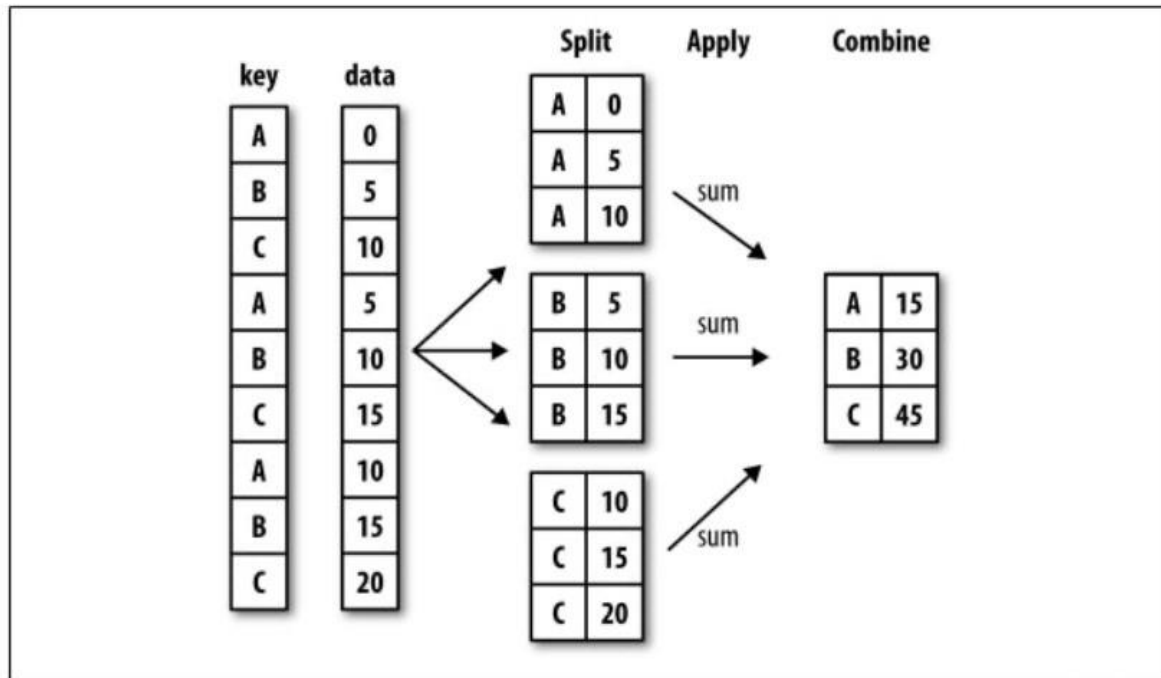


Figure 10-1. Illustration of a group aggregation

groupby

```
[55] kbo = pd.read_csv('kbo.csv')  
      kbo.head()
```

☞

	연도	순위	팀	경기수	승	패	무	승률	게임차
0	2019	1	두산	144	88	55	1	0.615	0.0
1	2019	2	키움	144	86	57	1	0.601	2.0
2	2019	3	SK	144	88	55	1	0.615	0.0
3	2019	4	LG	144	79	64	1	0.552	9.0
4	2019	5	NC	144	73	69	2	0.514	14.5

▶ kbo['팀'].unique()

☞ array(['두산', '키움', 'SK', 'LG', 'NC', 'KT', 'KIA', '삼성', '한화', '롯데', '넥센'],
dtype=object)

groupby

```
[▶] kbo.groupby('팀')
```

```
↳ <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fb8d4dcb7b8>
```

→ groupby를 하면 group으로 묶인 GroupBy 객체를 반환.

이 객체는 그룹 연산을 위해 필요 한 모든 정보를 가지고 있음

groupby

최적화 된 groupby 메소드

메소드	설명
count	그룹에서 NA가 아닌 값의 수를 반환
sum	NA가 아닌 값들의 합
mean	NA가 아닌 값들의 평균
median	NA가 아닌 값들의 산술 중간값
std, var	편향되지 않은($n-1$ 을 분모로 하는) 표준편차, 분산
min, max	NA가 아닌 값들 중 최소값과 최대값
prod	NA가 아닌 값들의 곱
first, last	NA가 아닌 값들 중 첫째 값과 마지막 값

groupby

최적화 된 groupby 메소드

```
kbo.groupby('팀').count()
```

연도 순위 경기수 승 패 무 승률 게임차

팀	연도	순위	경기수	승	패	무	승률	게임차
KIA	3	3	3	3	3	3	3	3
KT	3	3	3	3	3	3	3	3
LG	3	3	3	3	3	3	3	3
NC	3	3	3	3	3	3	3	3
SK	3	3	3	3	3	3	3	3
넥센	2	2	2	2	2	2	2	2
두산	3	3	3	3	3	3	3	3
롯데	3	3	3	3	3	3	3	3
삼성	3	3	3	3	3	3	3	3
키움	1	1	1	1	1	1	1	1
한화	3	3	3	3	3	3	3	3

→ '팀' 컬럼에 있었던 **unique** 값을 기준으로
groupby함, 새롭게 생성된 색인

groupby

- 두 개 이상의 색인으로도 groupby 할 수 있음
→ 인자로 넘기는 순서에 대응하여
계층적 인덱스를 만들어 냄

```
kbo.groupby(['연도', '팀']).sum()
```

		순위	경기수	승	패	무	승률	게임차
연도	팀							
2017	KIA	1	144	87	56	1	0.608	0.0
	KT	10	144	50	94	0	0.347	37.5
	LG	6	144	69	72	3	0.489	17.0
	NC	4	144	79	62	3	0.560	7.0
	SK	5	144	75	68	1	0.524	12.0
	넥센	7	144	69	73	2	0.486	17.5
	두산	2	144	84	57	3	0.596	2.0
	롯데	3	144	80	62	2	0.563	6.5
	삼성	9	144	55	84	5	0.396	30.0
	한화	8	144	61	81	2	0.430	25.5
2018	KIA	5	144	70	74	0	0.486	8.5
	KT	9	144	59	82	3	0.418	18.0

groupby

- 컬럼이나 컬럼의 일부만 선택하려면?
컬럼 이름이 담긴 배열 이용

```
kbo.groupby('팀')['승률'].max()
```

```
팀
KIA      0.608
KT        0.500
LG        0.552
NC        0.560
SK        0.615
넥센      0.521
두산      0.646
롯데      0.563
삼성      0.486
키움      0.601
한화      0.535
Name: 승률, dtype: float64
```

```
kbo.groupby(['연도', '팀'])['승률', '순위'].max()
```

연도	팀		
2017	KIA	0.608	1
	KT	0.347	10
	LG	0.489	6
	NC	0.560	4
	SK	0.524	5
	넥센	0.486	7
	두산	0.596	2
	롯데	0.563	3
	삼성	0.396	9
	한화	0.430	8
2018	KIA	0.486	5
	KT	0.418	9
	LG	0.476	8
	NC	0.406	10
	SK	0.545	2

groupby

- groupby.get_group

```
▶ grouped= kbo.groupby('팀')  
type(grouped)
```

```
☞ pandas.core.groupby.generic.DataFrameGroupBy
```

```
▶ for name, group in grouped:  
    print(name)  
    print(group)  
  
    print('-'*50)
```

```
☞ KIA
```

	연도	순위	팀	경기수	승	패	무	승률	게임차
6	2019	7	KIA	144	62	80	2	0.437	25.5
14	2018	5	KIA	144	70	74	0	0.486	8.5
20	2017	1	KIA	144	87	56	1	0.608	0.0

```
KT
```

	연도	순위	팀	경기수	승	패	무	승률	게임차
5	2019	6	KT	144	71	71	2	0.500	16.5
18	2018	9	KT	144	59	82	3	0.418	18.0
29	2017	10	KT	144	50	94	0	0.347	37.5

→ groupby를 하면
group으로 묶인 GroupBy 객체를 반환

groupby

- `groupby.get_group`

```
▶ grouped.get_group('한화')
```



	연도	순위	팀	경기수	승	패	무	승률	게임차
8	2019	9	한화	144	58	86	0	0.403	30.5
12	2018	3	한화	144	77	67	0	0.535	1.5
27	2017	8	한화	144	61	81	2	0.430	25.5

groupby

- groupby.agg
 - 그룹별로 특정한 집계함수를 적용

```
grouped['순위'].agg([('함수', lambda val : val.max()-val.min())])
```



함수

팀

KIA	6
KT	4
LG	4
NC	6
SK	3

→ 튜플을 이용하면 적용되는 함수의 이름을 따로 지정해줄 수 있음

groupby

- groupby.agg
 - 그룹별로 특정한 집계함수를 적용

```
grouped.agg({'순위':np.mean, '승률':[np.mean, np.std]})
```



팀	순위	승률	
	mean	mean	std
KIA	4.333333	0.510333	0.088059
KT	8.333333	0.421667	0.076566
LG	6.000000	0.505667	0.040649
NC	6.333333	0.493333	0.079053
SK	3.333333	0.561333	0.047648

→ 딕셔너리를 이용하면 컬럼별로 다른 함수를 적용 할 수 있음

groupby

- groupby.filter

- 그룹화 한 후 필터링

```
[15] kbo.groupby('팀').filter(lambda x: len(x)==2)
```




	연도	순위	팀	경기수	승	패	무	승률	게임차
13	2018	4	넥센	144	75	69	0	0.521	3.5
26	2017	7	넥센	144	69	73	2	0.486	17.5

groupby

- groupby.filter

- 그룹화 한 후 필터링

```
 kbo.groupby('팀').filter(lambda x: x['순위'].min()==1)
```



	연도	순위	팀	경기수	승	패	무	승률	게임차
0	2019	1	두산	144	88	55	1	0.615	0.0
6	2019	7	KIA	144	62	80	2	0.437	25.5
10	2018	1	두산	144	93	51	0	0.646	-14.5
14	2018	5	KIA	144	70	74	0	0.486	8.5
20	2017	1	KIA	144	87	56	1	0.608	0.0
21	2017	2	두산	144	84	57	3	0.596	2.0