Pandas

박 윤진

DATA 공공데이터포털 .GO.KR

로그인 회원가입 사이트맵 ENGLISH

데이터셋 제공신청 활용사례 정보공유 이용안내

데이터를 검색해보세요!

5.경제

최근 사회현안 및 이슈 저출산/고령화 데이터를 확인해 보세요!

국가중점데이터

데이터 카테고리

건축정보 교통사고정보 국민건강정보 상권정보 수산정보 실시간 수도정보 농수축산가격정보 등산로 정보

부동산종합정보 통합재정정보 지방행정정보 부동산거래정보 식의약품종합정보 지방재정정보 법령정보 + 더보기

기준년도	가입자일련번호	성별코드	연령대코드(5세단위)	시도코드	신장(5Cm단위)	체중(5Kg단위)	허리둘레	시력(좌)	시력(우)	청력(좌)	청력(우)
2018	1	2	7	48	160	60	79.5	1.5	1.5	1	1
2018	2	1	6	26	170	55	69.3	1.2	0.8	1	1
2018	3	1	12	28	165	70	85	0.8	0.8	2	1
2018	4	2	15	27	150	45	71.5	0.4	0.3	1	1
2018	5	2	14	41	145	50	77	0.7	0.6	1	1
2018	6	2	12	27	155	50	75	0.2	1.2	1	1
2018	7	1	12	31	175	65	80	0.5	9.9	1	1
2018	8	1	13	44	165	85	98	1	0.9	1	1
2018	9	2	11	41	155	55	69	0.8	0.8	1	1

pandas

- 데이터 처리, 분석용 라이브러리
- 표 형식의 데이터, 시계열 데이터 등에 적합

pandas 핵심 기능

- missing data 처리가 용이
- 축의 이름에 따라 데이터를 정렬할 수 있는 자료구조 제공
- 시계열 기능
- 일반 데이터베이스처럼 데이터를 합치고 관계연산을 수행하는 기능

자료구조

Data Frame

	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
5	2020. 1. 9 오후 3:30:00	2186.45	666.09	11055	107980
6	2020. 1. 10 오후 3:30:00	2206.39	673.03	11035	107760
7	2020. 1. 13 오후 3:30:00	2229.26	679.22	11080	107695
8	2020. 1. 14 오후 3:30:00	2238.88	678.71	10975	107860

자료구조

Series

	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
5	2020. 1. 9 오후 3:30:00	2186.45	666.09	11055	107980
6	2020. 1. 10 오후 3:30:00	2206.39	673.03	11035	107760
7	2020. 1. 13 오후 3:30:00	2229.26	679.22	11080	107695
8	2020. 1. 14 오후 3:30:00	2238.88	678.71	10975	107860

자료구조

Series

Data Frame

	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
5	2020. 1. 9 오후 3:30:00	2186.45	666.09	11055	107980
6	2020. 1. 10 오후 3:30:00	2206.39	673.03	11035	107760
7	2020. 1. 13 오후 3:30:00	2229.26	679.22	11080	107695
8	2020. 1. 14 오후 3:30:00	2238.88	678.71	10975	107860

자료구조: Series

- 1차원 배열 같은 자료구조

```
obj1 = pd.Series([10, 20, 30, 40])
obj1
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

자료구조: Data Frame

- 표 같은 스프레드시트 형식의 자료구조

	제목	감독	관객수
0	극한직업	이병헌	16264944
1	어벤져스: 엔드게임	안소니 루소,조 루소	13934592
2	알라딘	가이 리치	12551956
3	기생충	봉준호	10084564
4	엑시트	이상근	9424431

Indexing

- Series indexing

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

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

```
obj['c']
```

```
2
```

```
obj[2]
```

```
2
```

Indexing

- Data Frame indexing

```
frame = pd.DataFrame(np.arange(12).reshape(3,4),
                     columns = ['c1', 'c2', 'c3', 'c4'],
                     index = ['r1', 'r2', 'r3'])
frame
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11

```
frame[['c1', 'c4']]
```

	c1	c4
r1	0	3
r2	4	7
r3	8	11

Indexing

- Data Frame indexing

  column 먼저 선택!!

```
frame['c1'][['r1', 'r2']]
```

```
r1    0
r2    4
Name: c1, dtype: int64
```

Indexing

- .loc : label-based indexing

- .iloc : Purely integer-location based indexing for selection by position.

Indexing

```
frame.loc[['r1','r2'],['c2','c3','c4']]
```

	c2	c3	c4
r1	1	2	3
r2	5	6	7

```
frame.iloc[[0,1], 1:4]
```

	c2	c3	c4
r1	1	2	3
r2	5	6	7

drop

```
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

```
frame.drop('r1')
```

	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

산술연산

```
s1 = pd.Series([1, 2, 3, 4], index = ['a', 'b', 'c', 'd'])
```

```
s2 = pd.Series([10, 20, 30, 40], index = ['a', 'b', 'c', 'd'])
```

```
s1+s2
```

```
a    11
b    22
c    33
d    44
dtype: int64
```

산술연산

```
s1 = pd.Series([1, 2, 3, 4], index = ['a', 'b', 'c', 'd'])
```

```
s3 = pd.Series([2, 2, 2, 2], index = ['b', 'c', 'd', 'e'])
```

```
s1+s3
```

```
a    NaN
b    4.0
c    5.0
d    6.0
e    NaN
dtype: float64
```

산술연산

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

apply, applymap

DataFrame.apply(func[, axis, raw, …]): 축 별로 함수 적용.

DataFrame.applymap(func): element 별로 함수 적용.

apply, applymap

```
frame
```

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

```
f = lambda x: x.max()-x.min()
frame.apply(f)
```

```
a    8
b    8
c    8
d    8
dtype: int64
```

```
f = lambda x: x**2
frame.applymap(f)
```

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

apply, map

Series.apply: element 별로 함수 적용.

map 보다 더 복잡한 함수 사용 가능.

Series.map: element 별로 함수 적용.

apply, applymap

```
series
```

```
0    0
1    1
2    2
dtype: int64
```

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

```
series.apply(f)
```

```
0    3
1    4
2    5
dtype: int64
```

```
series.map(f)
```

```
0    3
1    4
2    5
dtype: int64
```

apply, applymap

```
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
```

sort

Series.sort_index : index를 기준으로 정렬

Series.sort_values: values를 기준으로 정렬

```
obj
```

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

```
obj.sort_index()
```

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

```
obj.sort_values()
```

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

sort

DataFrame.sort_index: index를 기준으로 정렬

DataFrame.sort_values: values를 기준으로 정렬

```
frame
```

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

```
frame.sort_index()
```

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

```
frame.sort_index(axis=1)
```

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

sort

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

frame

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

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

기술 통계 계산과 요약

```
frame
```

	c1	c2	c3	c4
i1	10.0	4.0	20.0	10
i2	10.0	2.5	10.5	-10
i3	NaN	10.0	10.0	5

```
frame.describe()
```

	c1	c2	c3	c4
count	2.0	3.000000	3.000000	3.000000
mean	10.0	5.500000	13.500000	1.666667
std	0.0	3.968627	5.634714	10.408330
min	10.0	2.500000	10.000000	-10.000000
25%	10.0	3.250000	10.250000	-2.500000
50%	10.0	4.000000	10.500000	5.000000
75%	10.0	7.000000	15.250000	7.500000
max	10.0	10.000000	20.000000	10.000000

기술 통계 계산과 요약

```
frame.max()
```

```
c1    10.0
c2    10.0
c3    20.0
c4    10.0
dtype: float64
```

```
frame.min(axis=1)
```

```
i1     4.0
i2   -10.0
i3     5.0
dtype: float64
```

기술 통계 계산과 요약

unique: series에서 중복되는 값을 제외하고 유일값만 포함하는 배열을 반환.

value_conuts: series에서 유일값에 대한 색인과 도수를 계산

```
obj
```

```
0    a
1    b
2    c
3    a
4    b
5    c
6    a
7    a
8    a
dtype: object
```

```
obj.unique()
```

```
array(['a', 'b', 'c'], dtype=object)
```

```
obj.value_counts()
```

```
a    5
c    2
b    2
dtype: int64
```

missing data 처리

- missing data 있는지 확인하기

isnull

notnull

```
obj = pd.Series(['apple', 'mango', np.nan, None, 'peach'])
```

```
obj.isnull()
0    False
1    False
2     True
3     True
4    False
dtype: bool
```

```
obj.notnull()
0     True
1     True
2    False
3    False
4     True
dtype: bool
```

missing data 처리

인자	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

중복제거

- .duplicated() : 각 로우가 중복인지(True) 아닌지(False) 알려주는 불리언 series 반환

- .drop_duplicates(): duplicated 배열이 False인 dataframe 반환

```
data
```

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

```
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

replace

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

```
obj.replace(-999, np.nan)
```

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

binning

cut

```
ages
```

```
[20, 35, 67, 39, 59, 44, 56, 77, 28, 52, 19, 33, 5, 15, 50, 29, 21, 33, 45, 85]
```

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

```
cuts = pd.cut(ages, bins)
cuts
```

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

groupby

```
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

groupby

```
kbo.groupby('팀').mean()
```

팀	연도	순위	경기수	승	패	무	승률	게임차
KIA	2018.0	4.333333	144.0	73.000000	70.000000	1.000000	0.510333	11.333333
KT	2018.0	8.333333	144.0	60.000000	82.333333	1.666667	0.421667	24.000000
LG	2018.0	6.000000	144.0	72.000000	70.333333	1.666667	0.505667	12.000000
NC	2018.0	6.333333	144.0	70.000000	72.000000	2.000000	0.493333	13.833333
SK	2018.0	3.333333	144.0	80.333333	62.666667	1.000000	0.561333	4.000000
넥센	2017.5	5.500000	144.0	72.000000	71.000000	1.000000	0.503500	10.500000
두산	2018.0	1.333333	144.0	88.333333	54.333333	1.333333	0.619000	-4.166667

groupby

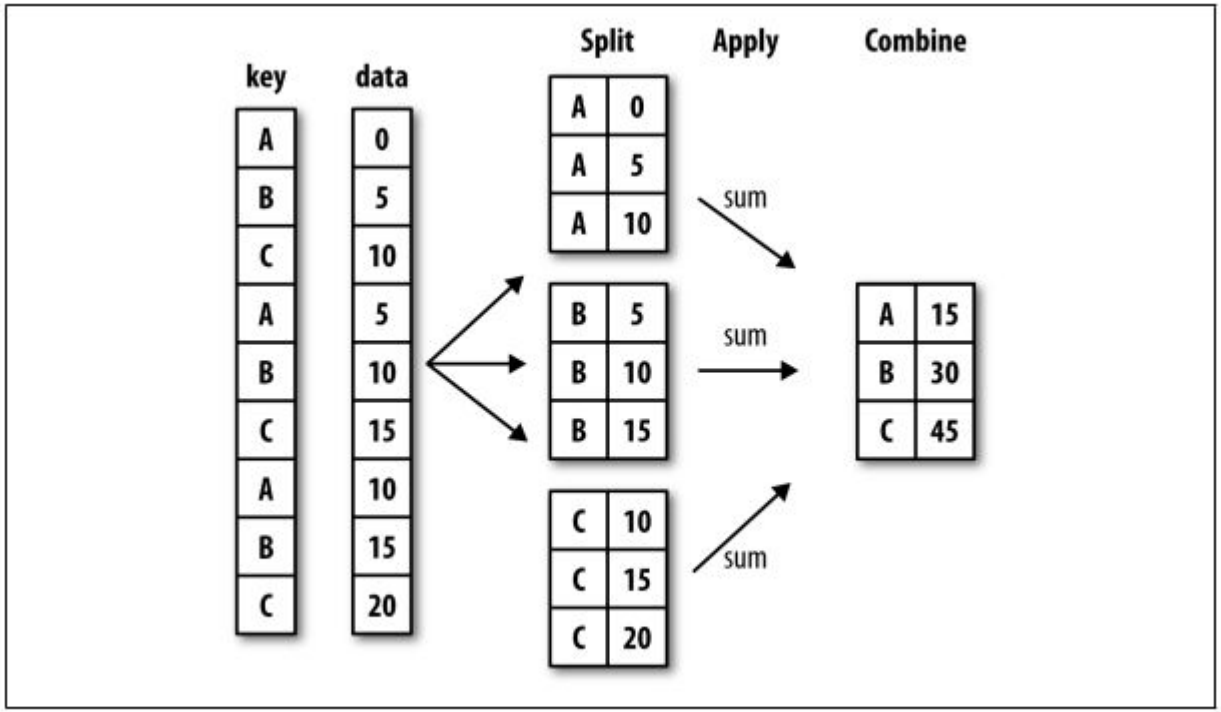

Figure 10-1. Illustration of a group aggregation

groupby

```
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'].sum()
```

```
연도    팀
2017  KIA    0.608
      KT     0.347
      LG     0.489
      NC     0.560
      SK     0.524
      넥센     0.486
      두산     0.596
      롯데     0.563
      삼성     0.396
      한화     0.430
2018  KIA    0.486
      KT     0.418
      LG     0.476
      NC     0.406
      SK     0.545
      넥센     0.521
      두산     0.646
      롯데     0.479
      삼성     0.486
      한화     0.535
2019  KIA    0.437
      KT     0.500
      LG     0.552
      NC     0.514
      SK     0.615
      두산     0.615
      롯데     0.340
      삼성     0.420
      키움     0.601
      한화     0.403
Name: 승률, dtype: float64
```

groupby

```
h_index = kbo.groupby(['연도','팀'])['승률'].mean()
h_index.index
```

```
MultiIndex([(2017, 'KIA'),
            (2017,  'KT'),
            (2017,  'LG'),
            (2017,  'NC'),
            (2017,  'SK'),
            (2017,  '넥센'),
            (2017,  '두산'),
            (2017,  '롯데'),
            (2017,  '삼성'),
            (2017,  '한화'),
            (2018, 'KIA'),
            (2018,  'KT'),
            (2018,  'LG'),
            (2018,  'NC'),
            (2018,  'SK'),
            (2018,  '넥센'),
            (2018,  '두산'),
            (2018,  '롯데'),
            (2018,  '삼성'),
            (2018,  '한화'),
            (2019, 'KIA'),
            (2019,  'KT'),
            (2019,  'LG'),
            (2019,  'NC'),
            (2019,  'SK'),
            (2019,  '두산'),
            (2019,  '롯데'),
            (2019,  '삼성'),
            (2019,  '키움'),
            (2019,  '한화')],
           names=['연도', '팀'])
```

- Index level을 기준으로 기본 연산 수행 가능

```
kbo_h.sum(level=0)
```

```
연도
2017    4.999
2018    4.998
2019    4.997
Name: 승률, dtype: float64
```

groupby

```
h_index = kbo.groupby(['연도','팀'])['승률'].mean()
h_index.index
```

```
MultiIndex([(2017, 'KIA'),
            (2017,  'KT'),
            (2017,  'LG'),
            (2017,  'NC'),
            (2017,  'SK'),
            (2017,  '넥센'),
            (2017,  '두산'),
            (2017,  '롯데'),
            (2017,  '삼성'),
            (2017,  '한화'),
            (2018, 'KIA'),
            (2018,  'KT'),
            (2018,  'LG'),
            (2018,  'NC'),
            (2018,  'SK'),
            (2018,  '넥센'),
            (2018,  '두산'),
            (2018,  '롯데'),
            (2018,  '삼성'),
            (2018,  '한화'),
            (2019, 'KIA'),
            (2019,  'KT'),
            (2019,  'LG'),
            (2019,  'NC'),
            (2019,  'SK'),
            (2019,  '두산'),
            (2019,  '롯데'),
            (2019,  '삼성'),
            (2019,  '키움'),
            (2019,  '한화')],
           names=['연도', '팀'])
```

```
kbo_h.mean(level=1).sort_values(ascending=False)
```

```
팀
두산     0.619000
키움     0.601000
SK      0.561333
KIA     0.510333
LG      0.505667
넥센      0.503500
NC      0.493333
롯데      0.460667
한화      0.456000
삼성      0.434000
KT      0.421667
Name: 승률, dtype: float64
```

groupby

groupby.get_group : 그룹화 후 특정 그룹을 선택 가능

groupby.agg: 그룹별로 aggregating 해줌.

groupby.filter: 그룹화하여 필터로 걸러냄.

merge

keys를 이용해 데이터의 row를 연결시켜 합침.

(sql의 join과 유사)

how: {‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘inner’

merge

data1

	id	col1	col2
0	01	19	1625
1	02	16	1091
2	03	17	1769
3	04	2	1502
4	05	25	1072
5	06	3	1092

data2

	id	col1
0	04	2920
1	05	3360
2	06	2733
3	07	2548

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

	id	col1_x	col2	col1_y
0	04	2	1502	2920
1	05	25	1072	3360
2	06	3	1092	2733

merge

data1

	id	col1	col2
0	01	19	1625
1	02	16	1091
2	03	17	1769
3	04	2	1502
4	05	25	1072
5	06	3	1092

data2

	id	col1
0	04	2920
1	05	3360
2	06	2733
3	07	2548

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

	id	col1_x	col2	col1_y
0	01	19	1625	NaN
1	02	16	1091	NaN
2	03	17	1769	NaN
3	04	2	1502	2920.0
4	05	25	1072	3360.0
5	06	3	1092	2733.0

Concatenate

```
s1 = pd.Series([100, 200], index=['c', 'b'])
s2 = pd.Series([300, 300, 300], index=['c', 'd', 'e'])
s3 = pd.Series([500, 600], index=['f', 'g'])
```

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

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