

데이터 분석을 위한

Pandas

류영표 강사

ryp1662@gmail.com

Copyright © "Youngpyo Ryu" All Rights Reserved.

This document was created for the exclusive use of "Youngpyo Ryu".

It must not be passed on to third parties except with the explicit prior consent of "Youngpyo Ryu".



류영표

Youngpyo Ryu

동국대학교 수학과/응용수학 석사수료

現 Upstage AI X 네이버 부스트 캠프 AI tech 1~4기 멘토

前 Innovation on Quantum & CT(IQCT) 이사

前 한국파스퇴르연구소 Image Mining 인턴(Deep learning)

前 (주)셈웨어(수학컨텐츠, 데이터 분석 개발 및 연구인턴)

강의 경력

- 현대자동차 연구원 강의 (인공지능/머신러닝/딥러닝/강화학습)
- (주) 모두의연구소 Aiffel 1기 퍼실리테이터(인공지능 교육)
- 인공지능 자연어처리(NLP) 기업데이터 분석 전문가 양성과정 멘토
- 공공데이터 청년 인턴 / SW공개개발자대회 멘토
- 고려대학교 선도대학 소속 30명 딥러닝 집중 강의
- 이젠 종로 아카데미(파이썬, ADSP 강사)
- 최적화된 도구(R/파이썬)을 활용한 애널리스트 양성과정(국비과정) 강사
- 한화, 하나금융사, 한전 KDN 교육
- 인공지능 신뢰성 확보를 위한 실무 전문가 자문위원
- 보건·바이오 AI활용 S/W개발 및 응용전문가 양성과정 강사
- Upstage AI X KT 융합기술원 기업교육 모델최적화 담당 조교

주요 프로젝트

및 기타사항

- 개인 맞춤형 당뇨병 예방·관리 인공지능 시스템 개발 및 고도화(안정화)
- 폐플라스틱 이미지 객체 검출 경진대회 3위
- 인공지능(AI)기반 데이터 사이언티스트 전문가 양성과정 1기 수료
- 제 1회 산업 수학 스터디 그룹 (질병에 영향을 미치는 유전자 정보 분석)
- 제 4,5회 산업 수학 스터디 그룹 (피부암, 유방암 분류)
- 빅데이터 여름학교 참석 (혼잡도를 최소화하는 새로운 노선 건설 위치의 최적화 문제)

Pandas

- Pandas란

- 데이터 처리, 분석용 라이브러리
- 표 형식 데이터, 시계열 데이터 등 다양한 형태의 데이터를 다루는데에 초점
(csv, text files, excel, SQL databases, …)
- numpy의 배열 기반 계산 스타일을 많이 차용

- Pandas 특징

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

Pandas

Series와 DataFrame

series

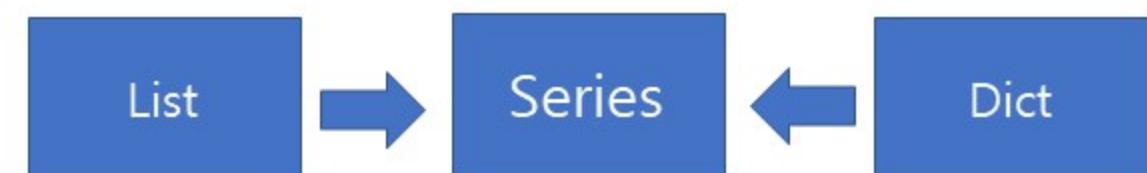
	Date	kospi	kosdaq	gold_fut_132030	Bond_273130
0	2020. 1. 2 오후 3:30:00	2175.17	674.02	10845	108215
1	2020. 1. 3 오후 3:30:00	2176.46	669.93	11000	108565
2	2020. 1. 6 오후 3:30:00	2155.07	655.31	11245	108745
3	2020. 1. 7 오후 3:30:00	2175.54	663.44	11180	108400
4	2020. 1. 8 오후 3:30:00	2151.31	640.94	11360	108270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

DataFrame

Pandas

Series

Index	Data
1	'A'
2	'B'
3	'C'
4	'D'
5	'E'

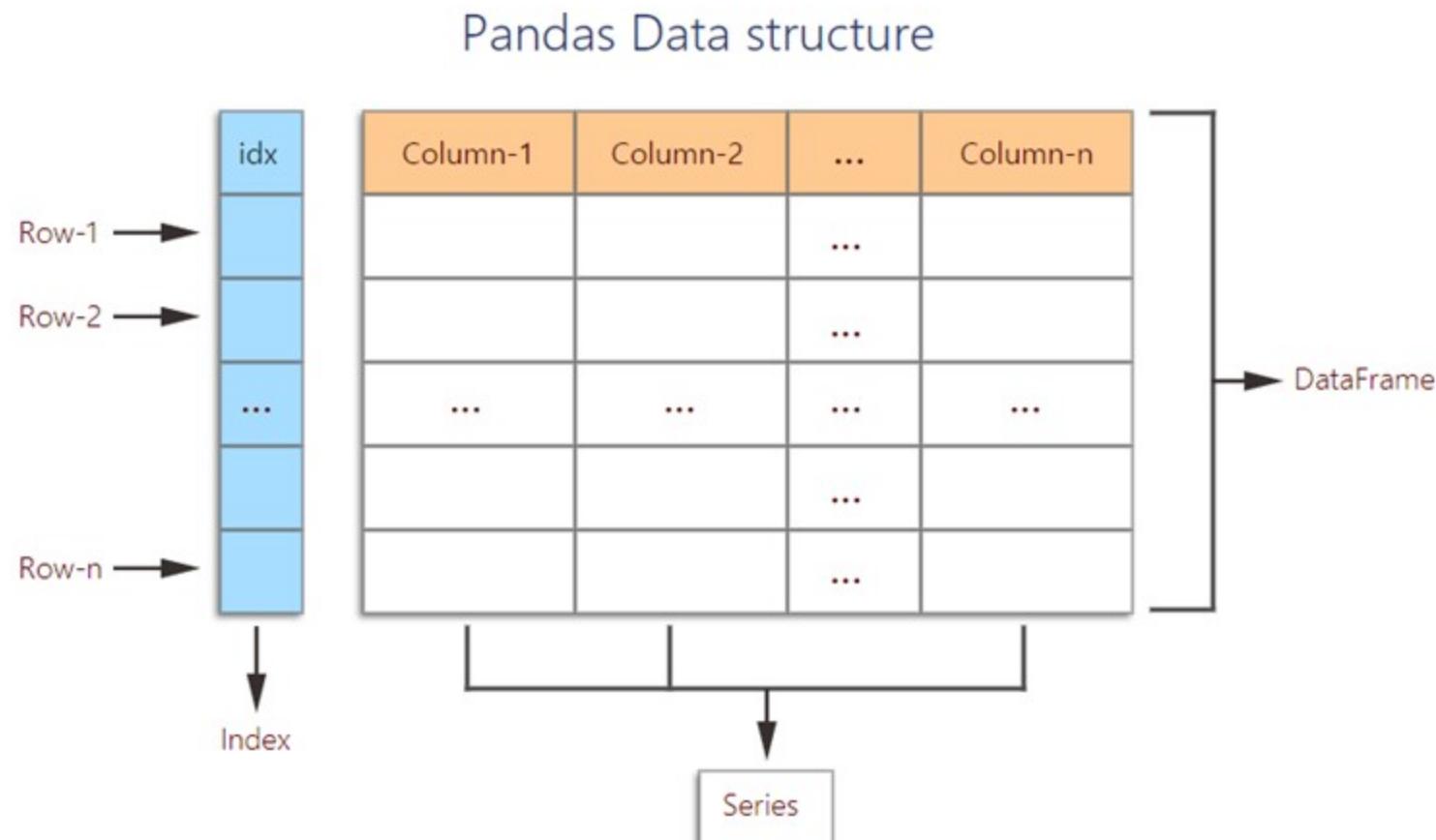


Series와 DataFrame

- Series
 - 시리즈는 1차원 배열 같은 자료구조
 - 1) Series 생성하기
 - list, numpy, dictionary 등을 이용하여 생성 가능
 - `pd.Series(data, index, dtype, name)`
 - `index` : 각 row의 이름, 기본적으로는 숫자로 되어있음
 - `dtype` : numpy의 `dtype`과 동일
 - `name` : 각 column의 이름, 기본값이 없음

Pandas

DataFrame



Series와 DataFrame

- DataFrame
 - 표 같은 스프레드 시트 형식의 자료구조
 - index가 같은 series가 여러개 모여있는 형태

1) DataFrame 생성하기

- dictionary나 numpy배열을 이용하여 생성

pd.DataFrame(data, index, dtype, columns)

- **index** : 각 row의 이름, 기본적으로는 숫자로 되어있음
- **dtype** : numpy의 dtype과 동일
- **columns** : 각 column의 이름, 기본값이 없음

Pandas

데이터 다루기

- 데이터 로딩

pandas file 파싱 함수

	설명
read_csv	파일, url, 파일과 유사한 객체로부터 구분된 데이터 읽어옴. 데이터 구분자는 쉼표(,)가 기본
read_excel	엑셀(xls, xlsx)에서 표 형식의 데이터를 읽어옴.
read_json	JSON 문자열에서 데이터를 읽어옴.
read_pickle	파이썬 피클 포맷으로 저장된 객체를 읽어옴..

- 데이터 저장

pandas file 저장 함수

	설명
to_csv	csv 형식으로 저장
to_json	JSON 형식으로 저장
to_pickle	파이썬 피클 포맷으로 저장

(참고) 더많은 데이터파싱, 저장 함수

Series

1. integer-location based

-numpy indexing과 유사하게 동작함

2. label-location based

-label(index)를 기준으로 동작함

3. Series.iloc

-integer-location based property

4. Series.loc

-label-location based property

Series 생성

```
import pandas as pd
import numpy as np

obj = pd.Series([0,1,2,3,4,5,6,7], index=['a','b','c','d','e','f','g','h'], dtype = 'int64')
obj
```

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
h    7
dtype: int64
```

```
# index
obj2 = pd.Series(['a', 'b', 'c'], index = ['i1','i2','i3'])
obj2
```

```
i1    a
i2    b
i3    c
dtype: object
```

Pandas

Series

1. integer-location based

- numpy indexing과 유사하게 동작함

```
[10] obj
```

```
↳ a    0  
  b    1  
  c    2  
  d    3  
  e    4  
  f    5  
  g    6  
  h    7  
  
dtype: int64
```

```
[5] obj[3]
```

```
↳ 3
```

```
[6] obj[-1] → 리스트에 위치(int)를 넘겨주면  
                    그위치에 해당되는 자료들을  
                    index(label)값과 함께반환
```

```
[7] obj[[1,3,5]]
```

```
↳ b    1  
  d    3  
  f    5  
  
dtype: int64
```

Pandas

Series

1. integer-location based

- numpy indexing과 유사하게 동작함

```
[10] obj
```

```
↳ a    0  
  b    1  
  c    2  
  d    3  
  e    4  
  f    5  
  g    6  
  h    7  
dtype: int64
```

```
▶ obj[1:3]
```

```
↳ b    1  
  c    2  
dtype: int64
```

```
[9] # boolean  
    obj[obj < 3]
```

```
↳ a    0  
  b    1  
  c    2  
dtype: int64
```

→ slicing 또한 numpy slicing
과 유사하게 동작

→ boolean indexing 처럼

Pandas

Series

2. label-location based

- label(index)를 기준으로 동작함

```
[10] obj  
↳ a    0  
    b    1  
    c    2  
    d    3  
    e    4  
    f    5  
    g    6  
    h    7  
dtype: int64
```

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

```
↳ 2
```

```
[14] obj.c
```

```
↳ 2
```

```
[15] obj[['e', 'c']]
```

```
↳ e    4  
    c    2  
dtype: int64
```

→ indexing 처럼

→ method나 attribute처럼

→ 리스트에 담아서
여러개의 자료를 indexing

Pandas

Series

2. label-location based

- label(index)를 기준으로 동작함

```
[10] obj
```

```
↳ a    0  
    b    1  
    c    2  
    d    3  
    e    4  
    f    5  
    g    6  
    h    7  
dtype: int64
```

```
[16] obj['a':'c']
```

```
↳ a    0  
    b    1  
    c    2  
dtype: int64
```

주의

라벨값을 이용해 슬라이싱을
하면 끝점까지 모두 포함

Pandas

Series

2. label-location based

- label(index)를 기준으로 동작함

```
[10] obj
```

```
↳ a    0  
  b    1  
  c    2  
  d    3  
  e    4  
  f    5  
  g    6  
  h    7  
dtype: int64
```

```
[17] obj['d':'e']=100
```

→ slicing을 이용해 값을 할당하면
값이 잘 변경되는 것을 확인할 수 있음

```
[▶] obj
```

```
↳ a      0  
  b      1  
  c      2  
  d     100  
  e     100  
  f      5  
  g      6  
  h      7  
dtype: int64
```

Pandas

Series

3. Series.iloc

- integer-location based property

```
[10] obj
```

```
↳ a    0  
  b    1  
  c    2  
  d    3  
  e    4  
  f    5  
  g    6  
  h    7  
dtype: int64
```

```
[27] obj.iloc[2]
```

```
↳ 2
```

```
[28] obj.iloc[[2]]
```

```
↳ c    2  
      dtype: int64
```

```
[29] obj.iloc[1:4]
```

```
↳ b    1  
  c    2  
  d    3  
dtype: int64
```

→ 위치에 해당하는 값만

→ 리스트에 담아 위치를 넘기면 index(label)과 함께 반환

→ 위치기반 슬라이싱

Pandas

Series

3. Series.loc

- label-location based property

```
[10] obj
```

```
↳ a    0  
  b    1  
  c    2  
  d    3  
  e    4  
  f    5  
  g    6  
  h    7  
dtype: int64
```

```
[10] obj.loc['a':'c']
```

```
↳ a    0  
  b    1  
  c    2  
dtype: int64
```

→ 라벨로 슬라이싱을 하기
때문에 끝점까지 포함

DataFrame

1. indexing

- 컬럼을 기준으로

2. slicing

- 인덱스(라벨)을 기준으로

3. DataFrame.iloc

- integer-location based property

4. DataFrame.loc

- label-location based property

Pandas

DataFrame

1. indexing

- 컬럼을 기준으로

```
frame = pd.DataFrame(np.arange(24).reshape(4,-1),  
                      columns = ['c1', 'c2', 'c3', 'c4', 'c5', 'c6'],  
                      index = ['r1', 'r2', 'r3', 'r4'])
```

```
frame
```

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

```
[7] frame['c3']
```

```
↳ r1      2  
    r2      8  
    r3     14  
    r4     20  
Name: c3, dtype: int64
```

```
[8] frame.c3
```

```
↳ r1      2  
    r2      8  
    r3     14  
    r4     20  
Name: c3, dtype: int64
```

Pandas

DataFrame

1. indexing

- 컬럼을 기준으로

frame

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

(9)

frame[['c1', 'c2']]

↳

c1 c2

r1	0	1
r2	6	7
r3	12	13
r4	18	19

Pandas

DataFrame

2. slicing

- 라벨(row)을 기준으로

frame						
	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

[10] frame['r1':'r2']

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11

[11] frame['c1':'c2']

	c1	c2	c3	c4	c5	c6
--	----	----	----	----	----	----

→ 컬럼을 기준으로 슬라이싱을 하게 되면 아무런
값도 가져오지 않음

Q) C1,C2 으로 접근하려면?

frame[['c1','c2']]

	c1	c2
r1	0	1
r2	6	7
r3	12	13
r4	18	19

Pandas

DataFrame

3. DataFrame.iloc

- integer-location based property

df.iloc(row, column)

frame

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

[23] frame.iloc[[0], [3]]

c4

r1 3

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

c2 c3 c4

r1 1 2 3

r2 7 8 9

Pandas

DataFrame

df.loc(row, column)

4. DataFrame.loc

- label-location based property

frame						
	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

[25] frame.loc[['r1'], ['c4']]

→ c4

r1 3

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

→ c2 c3 c4

	c2	c3	c4
r1	1	2	3
r2	7	8	9

→ label-based는 슬라이싱을
하면 끝점까지 포함

산술 연산

- 산술 연산자 혹은 산술 연산 메소드를 사용하여 연산

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

Pandas

Series

- index를 기준으로 연산

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

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

```
[6] s1+s2
```

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

Pandas

Series

- 짹이 맞지 않는 인덱스가 있는 경우, SQL의 **outer join**과 유사하게 동작
→ 결과 값에 두 인덱스의 값이 통합

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

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

```
[9] # a, e  
s1+s3
```

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

→ 단, 서로 짹이 맞지 않는 인덱스는 결과값이 NaN(Not-a-Number)을 의미하는 단어로 pandas에서는 누락된값 혹은 NA로 취급)

Pandas

Series

- 짹이 맞지 않는 인덱스가 있는 경우, 결측치의 기본값을 지정해줄 수 있음
→ 메소드의 `fill_value` 인자를 이용하면 결측치를 채워줄 수 있음

```
[▶] s1.add(s3, fill_value=0)
```

```
↳ a    1.0
   b    4.0
   c    5.0
   d    6.0
   e    2.0
dtype: float64
```

Pandas

DataFrame

- index, column을 기준으로 연산

[16] d1

	c1	c2
id001	100.0	100.0
id002	100.0	100.0
id003	100.0	100.0

d1 + d2

	c1	c2	c3
id001	NaN	NaN	NaN
id002	NaN	100.0	NaN
id003	NaN	500.0	NaN
id004	NaN	NaN	NaN

[17] d2

	c2	c3
id002	0	200
id003	400	600
id004	800	1000

Pandas

DataFrame

- index, column을 기준으로 연산

```
[16] d1
```

	c1	c2
id001	100.0	100.0
id002	100.0	100.0
id003	100.0	100.0

```
[17] d2
```

	c2	c3
id002	0	200
id003	400	600
id004	800	1000

```
[19] # (id001, c3), (id004, c1)  
d1.add(d2, fill_value=0)
```

	c1	c2	c3
id001	100.0	100.0	NaN
id002	100.0	100.0	200.0
id003	100.0	500.0	600.0
id004	NaN	800.0	1000.0

→ 둘다없는경우는NaN으로 채워짐

Pandas

Series와 DataFrame

- Series의 index(label)을 frame의 column으로 맞추고 아래 row로 전파

[29] frame

```
↳   a   b   c  
0   1   1   1  
1   1   1   1  
2   1   1   1
```

▶ frame + series

```
↳   a   b   c  
0   101  201  301  
1   101  201  301  
2   101  201  301
```

[30] series

```
↳   a      100  
     b      200  
     c      300  
dtype: int64
```

Series와 DataFrame

- 메소드를 사용하면, axis 인자로 축을 지정해 column으로 전파할 수 있음
- axis = 0 또는 axis = 'index'

```
[29] frame
```

```
↳      a   b   c  
0    1   1   1  
1    1   1   1  
2    1   1   1
```

```
[▶] frame.add(series2, axis=0)
```

```
↳      a     b     c  
0   101  101  101  
1   201  201  201  
2   301  301  301
```

```
series2
```

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

→ 단, index 이름이 일치해야 함

Series와 DataFrame

```
import pandas as pd
s = set([i for i in dir(pd.Series) if not i.startswith("_")])
d = set([i for i in dir(pd.DataFrame) if not i.startswith("_")])
```

```
count=0
for i in (s-d):
    count +=1
    print(i,end=', ')
    if (count%4==0):
        print()
```

기술통계계산과 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

기술통계 계산과 요약

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

df.**head(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

df.**tail(개수)**

기술통계 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 요약제공

기술통계 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

Pandas

기술통계 계산과 요약

- describe

```
[13] frame
```

```
   c1  c2  c3
```

	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
```

	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.

기술통계 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 row를 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 넘겨주면 됨

기술통계 계산과 요약

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

series.unique()

[▶] obj

```
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

기술통계 계산과 요약

- `value_counts`

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

```
[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
```

```
[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
```

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
```

```
[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
```

```
[14] 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 정해준 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 ascending 인자도 multi-value를 줄 수 있음

Series

- map

map은 **series**의 각각의 **element**들을 다른 어떤 값을 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
```

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

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

→ arg가 dict인 경우, 원본 series에 있는 값을 key로 바꿔
줄 값을 value로 설정
만일, 원본 데이터에 있는 값이 dict에 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
```

```
[29] # 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, **kwds)

- **func** : series에 적용될 함수
- **args** : series를 제외한 함수에 들어갈 다른 매개변수
- ****kwds** : 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
```

```
[15] 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

Pandas

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())

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

	a	b	c	d
0	8	8	8	8
1	8	8	8	8
2	8	8	8	8

[22] dtype: int64

Pandas

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)
```

```
[23] frame.apply(lambda x: x.max()-x.min(), axis=1)
[23]      0      3
[23]      1      3
[23]      2      3
[23]      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

데이터삭제- 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

→ 기본axis = 0

데이터삭제- drop

- drop
row나 column에서 특정한 label을 삭제하는 함수

```
[31] frame
```

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

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

	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

데이터삭제- 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 붙일지 {'outer', 'inner'}

참고) Outer : 합집합으로 DataFrame 합치기

inner : 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 쪽이어붙이게 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 쪽이어붙이게 됨.
즉, 새로운 컬럼이 생기고 dataframe이 됨

Pandas

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
```

Pandas

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], 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

Pandas

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_on`으로 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

[14] 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
```

```
[18] 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

[16] `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, 대체하고자 하는 값을 value로 하는 dict를 넘겨주는 각 컬럼에 매칭하여 값을 변경

데이터 변형

- 중복 제거

- 1) `duplicated()`: 각로우가 중복인지(True) 아닌지(False) 알려주는
불리언 series 반환
- 2) `drop_duplicates()`: `duplicated`를 적용한 결과가 False인 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

Pandas

데이터 변형

- 중복 제거

- 1) `duplicated()`: 각 행이 중복인지(`True`) 아닌지(`False`) 알려주는 불리언 series 반환
- 2) `drop_duplicates()`: `duplicated`를 적용한 결과가 `False`인 것들만 모아서 dataframe 반환

data

	<code>id</code>	<code>name</code>	<code>phone</code>
0	0001	a	0
1	0002	b	1
2	0003	c	2
3	0001	a	3

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

	<code>id</code>	<code>name</code>	<code>phone</code>
1	0002	b	1
2	0003	c	2
3	0001	a	3

→ `subset` : 중복을 검색할 기준
`keep` : 중복자료에서 어떤 것을 남길지를 파악하는 인자

데이터변형

- 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
```

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

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

→ 한개 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], (40, 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

Pandas

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 주지 않고 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
```

Pandas

데이터 변형

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

label
dog
cat
apple



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

Pandas

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

```
[38] pd.get_dummies(df)
```

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

Pandas

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
pd.get_dummies(df)
```

	col1	col2	col3	col2	col1_001	col1_002	col1_003	col1_004	col1_005	col1_006	col3_경기도	col3_서울시	col3_제주도	
0	001	10	서울시	0	10	1	0	0	0	0	0	1	0	0
1	002	20	경기도	1	20	0	1	0	0	0	1	0	0	0
2	003	30	서울시	2	30	0	0	1	0	0	0	1	0	0
3	004	40	제주도	3	40	0	0	0	1	0	0	0	1	0
4	005	50	경기도	4	50	0	0	0	0	1	0	1	0	0
5	006	60	서울시	5	60	0	0	0	0	0	1	0	1	0

groupby

- 그룹연산은 분리-적용-결합의 과정

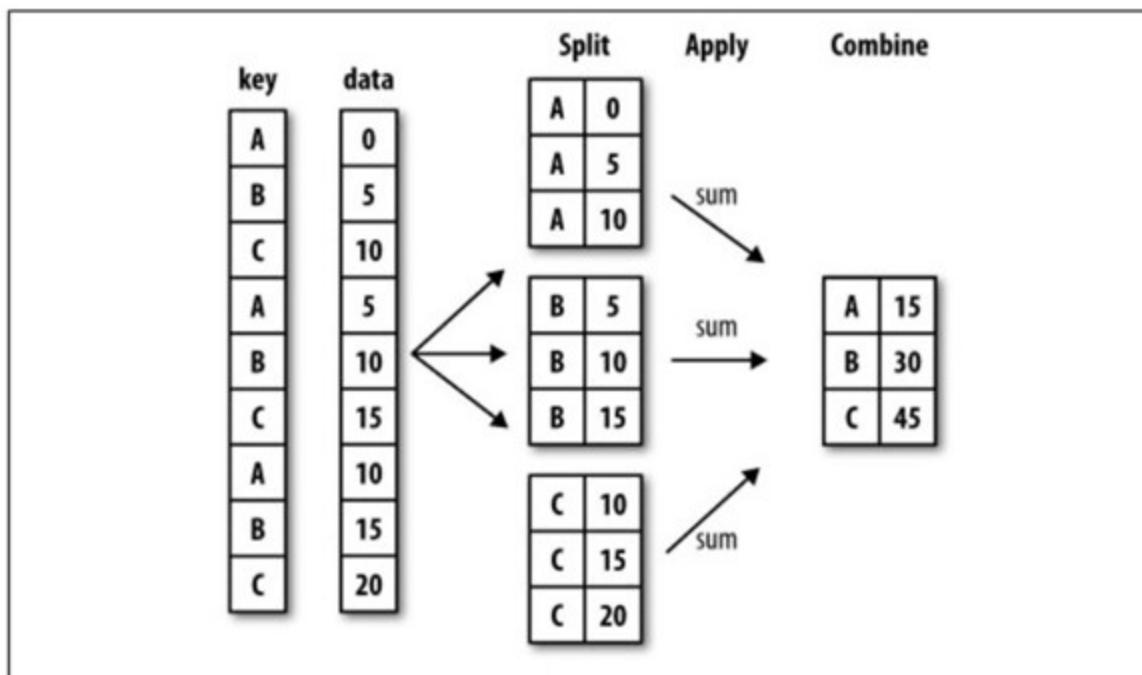


Figure 10-1. Illustration of a group aggregation

Pandas

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)
```

```
kbo.shape  
(30, 9)
```

```
: kbo.columns  
: Index(['연도', '순위', '팀', '경기수', '승', '패', '무', '승률', '게임차'],  
:         dtype='object')
```

Pandas

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)
```

```
kbo.shape  
(30, 9)
```

```
: kbo.columns  
: Index(['연도', '순위', '팀', '경기수', '승', '패', '무', '승률', '게임차'],  
: dtype='object')
```

groupby

.info() 함수는 데이터에 대한 전반적인 정보를 나타냅니다.

df를 구성하는 행과 열의 크기, 컬럼명, 컬럼을 구성하는 값의 자료형 등을 출력함

```
kbo.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 9 columns):
 #   Column   Non-Null Count  Dtype  
--- 
 0   연도      30 non-null    int64  
 1   순위      30 non-null    int64  
 2   팀        30 non-null    object  
 3   경기 수    30 non-null    int64  
 4   승        30 non-null    int64  
 5   패        30 non-null    int64  
 6   무        30 non-null    int64  
 7   승률      30 non-null    float64 
 8   게임차    30 non-null    float64 
dtypes: float64(2), int64(6), object(1)
memory usage: 2.2+ KB
```

groupby

.describe() 수치형 변수들만 기준으로, count, 평균, 표준편차, 최소/최댓값, 4분위를 기준으로 25%, 50% 75%에 해당하는 값들을 테이블로 출력해줌

```
kbo.describe()
```

	연도	순위	경기수	승	패	무	승률	개임자
count	30.000000	30.000000	30.0	30.000000	30.000000	30.000000	30.000000	30.000000
mean	2018.000000	5.466667	144.0	71.200000	71.200000	1.600000	0.499800	12.833333
std	0.830455	2.921187	0.0	11.871147	11.591912	1.275769	0.082079	12.576204
min	2017.000000	1.000000	144.0	48.000000	51.000000	0.000000	0.340000	-14.500000
25%	2017.000000	3.000000	144.0	61.250000	62.500000	1.000000	0.431750	2.375000
50%	2018.000000	5.000000	144.0	70.500000	71.500000	1.000000	0.494500	9.750000
75%	2019.000000	8.000000	144.0	79.000000	80.750000	2.000000	0.558000	19.500000
max	2019.000000	10.000000	144.0	93.000000	94.000000	5.000000	0.646000	39.000000

Pandas

groupby

.describe() 수치형 변수들만 기준으로, count, 평균, 표준편차, 최소/최댓값, 4분위를 기준으로 25%, 50% 75%에 해당하는 값들을 테이블로 출력해줌

```
kbo.describe()
```

	연도	순위	경기수	승	패	무	승률	개임차
count	30.000000	30.000000	30.0	30.000000	30.000000	30.000000	30.000000	30.000000
mean	2018.000000	5.466667	144.0	71.200000	71.200000	1.600000	0.499800	12.833333
std	0.830455	2.921187	0.0	11.871147	11.591912	1.275769	0.082079	12.576204
min	2017.000000	1.000000	144.0	48.000000	51.000000	0.000000	0.340000	-14.500000
25%	2017.000000	3.000000	144.0	61.250000	62.500000	1.000000	0.431750	2.375000
50%	2018.000000	5.000000	144.0	70.500000	71.500000	1.000000	0.494500	9.750000
75%	2019.000000	8.000000	144.0	79.000000	80.750000	2.000000	0.558000	19.500000
max	2019.000000	10.000000	144.0	93.000000	94.000000	5.000000	0.646000	39.000000

```
kbo.isnull().sum()  
# kbo.isna().sum()
```

```
연도      0  
순위      0  
팀        0  
경기수    0  
승        0  
패        0  
무        0  
승률      0  
개임차    0  
dtype: int64
```

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

```
[1] 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()

연도	팀	순위	경기수	승	패	무	승률	게임차
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

Pandas

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

Pandas

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

→ 튜플을 이용하면 적용되는 함수의 이름을 따로 지정해줄 수 있음

KIA	6
KT	4
LG	4
NC	6
SK	3

groupby

- groupby.agg

- 그룹별로 특정한 집계함수를 적용

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

	순위	승률
	mean	mean
		std

팁

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

Thank you.

Pandas
ryp1662@gmail.com

Copyright © "Youngpyo Ryu" All Rights Reserved.
This document was created for the exclusive use of "Youngpyo Ryu".
It must not be passed on to third parties except with the explicit prior consent of "Youngpyo Ryu".