책 - 파이썬 라이브러리를 활용한 데이터 분석 - 웨스 맥키니 지음

Chapter 5 pandas 시작하기

pandas

- 고수준의 자료 구조와 파이썬을 통한 빠르고 쉬운 데이터 분석 도구를 포함한다.
- 데이터 프레임 구조를 위한 모듈
- 자동적으로 혹은 명시적으로 축의 이름에 따라 데이터를 정렬할 수 있는 자료 구조. 잘못 정렬된 데이터에 의한 일반적인 오류를 예방하고 다양한 소스에서 가져온 다양한 방식으로 색인되어 있는 데이터를 다룰 수 있는 기능
- 통합된 시계열 기능
- 시계열 데이터와 비시계열 데이터를 함께 다룰 수 있는 통합 자료 구조
- 산술연산과 한 축의 모든 값을 더하는 등 데이터의 축약연산은 축의 이름 같은 메타데이터로 전달
- 누락된 데이터를 유연하게 처리할 수 있는 기능
- SQL같은 일반 데이터베이스처럼 데이터를 합치고 관계연산을 수행하는 기능

5.1 pandas 자료 구조 소개

5.1.1 Series

- 일련의 객체를 담을 수 있는 1차원 배열 같은 자료 구조
- 색인(index)라고 하는 배열 데이터에 연관된 이름을 가지고 있다.
- 표현 : 왼쪽에 색인을 보여주고 오른쪽에 해당 색인의 값을 보여준다.
- index 생략할 경우 정수 0에서 N-1까지의 숫자가 표현됨.

cf. map - 1차원만 전달 가능 -> Series(row이름을 가지는 1차원 자료구조) 사용 가능
apply -> Series 사용 불가

```
In [34]: obj=Series([4,7,-5,3])    # 리스트로 전달
In [35]: obj
Out[35]:
0    4
1    7
2   -5
3    3
dtype: int64

In [36]: obj.values
Out[36]: array([ 4,  7, -5,  3], dtype=int64)
In [37]: obj.index
Out[37]: RangeIndex(start=0, stop=4, step=1)
```

각각의 데이터를 지칭하는 색인을 지정해 Series 객체를 생성할 때

In [38]: **obj2=Series([4,7,-5,3], index=['d','b','a','c'])**
 # value, index - 리스트로 전달 / index는 value와 순서대로 짝 맞춰서 저장됨
In [39]: **obj2**
Out[39]:
d 4
b 7
a -5
c 3
dtype: int64
In [40]: **obj2.index**
Out[40]: Index(['d', 'b', 'a', 'c'], dtype='object')

색인 사용
-index 이름으로 가능 (default는 0,1,2,3)

In [47]: **obj2['a']**
Out[47]: -5

In [48]: **obj2['d']**
Out[48]: 4

In [49]: **obj2[['c','a','d']]** **# 여러 개 - 리스트로 전달 (팬시색인([[]]형태의))**
Out[49]:
c 3
a -5
d 4
dtype: int64

-불리언 배열 사용 가능
In [51]: **obj2[obj2>0]**
Out[51]:
d 4
b 7
c 3
dtype: int64

- **산술곱셈, 수학함수를 적용해도 원본 데이터 변경 X**

In [52]: **obj2*2**

Out[52]:

```
d     8
b    14
a   -10
c     6
dtype: int64
```

In [54]: **np.exp(obj2)**

Out[54]:

```
d      54.598150
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

In [55]: **obj2**

Out[55]:

```
d    4
b    7
a   -5
c    3
dtype: int64
```

Series를 이해하는 다른 방법

- 고정길이의 정렬된 사전형
- 파이썬의 사전형을 인자로 받아야 하는 많은 함수에서 사전형을 대체하여 사용할 수 있다.
- 파이썬 사전 객체로부터 Series 객체를 생성할 수도 있다.
- 리스트, 배열도 가능하지만 index이름이 없다. (자동으로 0,1,2,3)

In [56]: **'b' in obj2** **# index값으로 찾음!**
Out[56]: True
In [57]: **'e' in obj2**
Out[57]: False

In [58]: **sdata={'Ohio' : 35000, 'Texas' : 71000, 'Oregon' : 16000, 'Utah' : 5000}**
In [59]: **obj3=Series(sdata)**
In [60]: **obj3**
Out[60]:
Ohio 35000
Texas 71000
Oregon 16000
Utah 5000
dtype: int64

In [61]: **states=['California','Ohio','Oregon','Texas']**
In [62]: **obj4=Series(sdata,index=states)**
 # 원래 있는 Series에 index 옵션 작성
-->원래 index 이름과 match 후 새로 쓴 index 이름 순서대로 재정렬,
맞는 이름이 없으면 남은 new index의 첫 이름으로 강제 지정 후 NaN으로 표시

In [63]: **obj4**
Out[63]:
California **NaN**
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
dtype: float64

누락된 데이터 찾기

In [74]: **pd.isnull(obj4)** **# 함수**
Out[74]:
California True
Ohio False
Oregon False
Texas False
dtype: bool

In [75]: **pd.notnull(obj4)** **# 함수**
Out[75]:
California False
Ohio True
Oregon True
Texas True
dtype: bool

In [76]: **obj4.isnull()** **# 메서드 (isnull 만 메서드로 있다)**
Out[76]:
California True
Ohio False
Oregon False
Texas False
dtype: bool

Series의 산술연산

- 같은 위치끼리 산술연산이 아님(cf. array in numpy)

- key(index)가 같은 값끼리 대응되어 연산됨

->둘 중 하나의 키가 없으면 결과는 NaN

->NaN의 산술연산 결과는 NaN

In [77]: **obj3**

Out[77]:

Ohio 35000

Texas 71000

Oregon 16000

Utah 5000

dtype: int64

In [78]: **obj4**

Out[78]:

California NaN

Ohio 35000.0

Oregon 16000.0

Texas 71000.0

dtype: float64

In [79]: **obj3+obj4**

Out[79]:

California NaN

Ohio 70000.0

Oregon 32000.0

Texas 142000.0

Utah NaN

dtype: float64

outer join이랑 비슷한 것 같기도..?0?

name속성

In [80]: **obj4.name='poplulation'** **# Series의 이름**

In [81]: **obj4.index.name='state'** **# index 이름**

In [82]: **obj4**

Out[82]:

state

California NaN

Ohio 35000.0

Oregon 16000.0

Texas 71000.0

Name: poplulation, dtype: float64

Series index의 변경

.index 객체를 통해 변경

In [83]: **obj.index=['Bob','Steve','Jeff','Ryan']**

#앞에서의 index가 무엇이었든, 이미 생성된 객체의 index를 대입하며 변경 가능

#Series(데이터(기존에 index가 존재하는 데이터), index=[새로운 index 입력 원함])와 다른 결과

In [84]: **obj**

Out[84]:

Bob 4

Steve 7

Jeff -5

Ryan 3

dtype: int64

5.1.2 DataFrame

DataFrame(data, columns=[col이름], index=[row이름])

- 표 같은 스프레드시트 형식의 자료 구조
- 각 칼럼은 서로 다른 종류의 값 (숫자, 문자열, 불리언 등)을 담을 수 있다.
- 로우 연산과 칼럼 연산은 거의 대칭적으로 취급된다.
- 내부적으로 데이터는 하나 이상의 2차원 형식으로 저장하므로, 고차원의 표 형식 데이터를 계층적 색인을 통해 쉽게 표현할 수 있다.
- 같은 길이의 리스트에 담긴 사전을 이용하거나 NumPy 배열을 이용한다.

```
In [86]: data={'state' : ['Ohio','Ohio','Ohio','Nevada','Nevada'],
    ...:       'year' : [2000,2001,2002,2001,2002],
    ...:       'pop' : [1.5,1.7,3.6,2.4,2.9]}        # 딕셔너리 형태

In [87]: frame=DataFrame(data)

In [88]: frame                    # index는 Series와 같은 방식으로 자동으로 대입됨
Out[88]:
   state  year  pop
0   Ohio  2000  1.5
1   Ohio  2001  1.7
2   Ohio  2002  3.6
3 Nevada  2001  2.4
4 Nevada  2002  2.9

In [89]: DataFrame(data,columns=['year','state','pop'])
#(이미 col이름 있는 경우) col이름대로 순서를 재배치함
#cf. (이름이 없는 경우) 원래 데이터 col 순서대로 새 col 이름이 입력됨
Out[89]:
   year  state  pop
0  2000   Ohio  1.5
1  2001   Ohio  1.7
2  2002   Ohio  3.6
3  2001 Nevada  2.4
4  2002 Nevada  2.9
```

#Series와 마찬가지로 data에 없는 값은 넘기면 NA값이 저장됨

In [90]: **frame2=DataFrame(data, columns=['year','state','pop','debt'],**
...: index=['one','two','three','four','five'])
...:

#index 옵션 - row이름

In [91]: **frame2**

Out[91]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

In [92]: **frame2.columns** **# .columns 메서드 통해 colname 불러오기 가능**

Out[92]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

색인 - key값 - col이름 색인

frame['key(col)이름'] / frame.key(col)이름

- row보다 col이 우선시됨

In [94]: **frame2['state']** **# [ ] 색인**

Out[94]:

one Ohio

two Ohio

three Ohio

four Nevada

five Nevada

Name: state, dtype: object

In [95]: **frame2.year** **# 메서드 형태**

Out[95]:

one 2000

two 2001

three 2002

four 2001

five 2002

Name: year, dtype: int64

- row이름 색인 loc / iloc 메서드

- **loc** : row이름 색인
iloc : 정수표현(순서) row 색인

In [97]: **frame2.loc['three']**
Out[97]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three, dtype: object
 # col이름이 다시 row이름으로(index) 전달되어 출력
 # series 형태로 출력

In [98]: **frame2.iloc[2]**
Out[98]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three, dtype: object

(원래는 .ix 메서든데 바뀔거라는 경고문 뜸! loc와 iloc에 익숙해지기)

칼럼에 대입하기 - 스칼라, 배열

In [101]: **frame2['debt']=16.5** **# 스칼라 값 대입**

In [102]: **frame2**

Out[102]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5

In [103]: **frame2['debt']=np.arange(5)** **# 배열 값 대입**

In [104]: **frame2**

Out[104]:

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4

리스트, 배열을 칼럼에 대입 시 대입하려는 값의 길이가 DataFrame의 크기와 같아야 함.

칼럼에 대입하기 - Series

- Series 대입 시 **DataFrame index 값에 따라 값이 대입**되며, **없는 색인에는 NaN 대입됨**.

In [105]: **val=Series([-1.2,-1.5,-1.7],index=['two','four','five'])** **# new 생성 -> 순서쌍으로 Series**

In [106]: **frame2['debt']=val**

 # index가 일치하는 것끼리 재배치(index 맞지 않을 시 대입 안 됨)

In [107]: **frame2**

Out[107]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

0622

~색인~

1. 없는 칼럼을 대입하면 새로운 칼럼이 생성
2. 색인을 이용해서 생성된 칼럼은 내부 데이터에 대한 뷰이며 얕은 복사 이루어짐

In [66]: **frame2['eastern'] = frame2.state=='Ohio'**

 # 없는 칼럼 [' ']의 이름으로 T/F 불리언 원소를 갖는 칼럼 추가

In [67]: **frame2** **# 얕은복사**

Out[67]:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

In [68]: **del frame2['eastern']** **# del 예약어를 사용해 칼럼 삭제 가능**

In [69]: **frame2.columns**

Out[69]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

중첩된 딕셔너리 ~ 데이터프레임

In [70]: **pop={'Nevada' : {2001 : 2.4, 2002 : 2.9}, 'Ohio' : {2000 : 1.5, 2001 :**
...: 1.7, 2002 : 3.6}} **# 중첩된 딕셔너리**

In [72]: **frame3=DataFrame(pop)**

In [73]: **frame3**

 # 중첩된 딕셔너리 -> 데이터 프레임시 바깥에 있는 사전의 키 값 ; 칼럼, 내부 키 : 로우

Out[73]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

 # pop에서 바깥 키 값 **# pop에서 내부 키 값**

.T : 전치

In [74]: **frame3.T**

Out[74]:

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

cf. 중첩된 사전에서 색인을 직접 지정하면 지정된 색인으로 DataFrame을 생성한다.

```
In [70]: DataFrame(pop,index=[2001,2002,2003])
```

Out[70]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

Series 객체를 담고 있는 사전 데이터도 같은 방식

In [90]: **pdata={'Ohio' : frame3['Ohio'][:-1],**
...: 'Nevada' : frame3['Nevada'][:2]}

In [92]: **DataFrame(pdata)**
Out[92]:

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

#참조

In [91]: **pdata**
Out[91]:
{'Ohio': 2000 1.5
2001 1.7
Name: Ohio, dtype: float64, 'Nevada': 2000 NaN
2001 2.4
Name: Nevada, dtype: float64}

cf.
.name 메서드

In [95]: **frame3.index.name = 'year' ; frame3.columns.name = 'state'**
In [96]: **frame3**
Out[96]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

.Values 속성 -- DataFrame에 저장된 데이터를 2차원 배열로 반환한다.

In [97]: **frame3.values**

Out[97]:

```
array([[nan, 1.5],
       [2.4, 1.7],
       [2.9, 3.6]])
```

In [98]: **frame2.values**

 # DataFrame의 칼럼에 서로 다른 dtype이 있다면, 모든 칼럼을 수용하기 위해 그 칼럼 배열의 dtype이 선택된다.

Out[98]:

```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7]], dtype=object)
```

DataFrame 생성자에서 사용 가능한 입력 데이터

형	설명
2차원 ndarray	데이터를 담고 있는 행렬, 선택적으로 로우와 칼럼의 이름을 전달할 수 있다.
배열, 리스트, 튜플의 딕셔너리	딕셔너리의 모든 항목은 같은 길이를 가져야 하며, 각 항목의 내용이 DataFrame의 칼럼이 된다.
Series 사전	Series의 각 값이 칼럼이 된다. 명시적으로 색인을 넘겨주지 않으면, 각 Series의 index값이 하나로 합쳐져서 로우의 색인이 된다.
사전의 사전	외부에 있는 사전 key 값이 column이름, 내부 key 값이 row이름이다.
사전이나 Series의 리스트	리스트의 각 항목이 DataFrame의 로우가 된다. 합쳐진 사전의 키 값이나 Series의 색인이 DataFrame 칼럼의 이름이 된다.
리스트나 튜플의 리스트	'2차원 ndarray'와 같은 방식으로 취급된다.
다른 DataFrame	색인이 따로 지정되지 않는다면, DataFrame의 색인이 그대로 사용된다.
NumPy Masked Array	'2차원 ndarray'와 같은 방식으로 취급되지만, 마스크 값은 반환되는 DataFrame에서 NA 값이 된다.

 # NA처리 유의 ---------- 결측치 처리 어떻게?0? 함수에서 옵션으로 !

5.1.3 색인 객체

 # 색인 객체는 only for searching, 특정 색인 결과만 수정할 수 없다. (전체 수정은 가능)

```
In [99]: obj=Series(range(3), index=['a','b','c'])
In [100]: index=obj.index
In [101]: index
Out[101]: Index(['a', 'b', 'c'], dtype='object')
In [102]: index[1:]
Out[102]: Index(['b', 'c'], dtype='object')
In [103]: index[1]='d'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-103-8be6e68dba2d> in <module>()
----> 1 index[1]='d'
~₩Anaconda3₩lib₩site-packages₩pandas₩core₩indexes₩base.py in __setitem__(self, k
ey, value)
   2048
   2049     def __setitem__(self, key, value):
-> 2050         raise TypeError("Index does not support mutable operations")
   2051
   2052     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

 # 색인 객체는 변경할 수 없기에 자료 구조 사이에서 안전하게 공유될 수 있다.

```
In [104]: index=pd.Index(np.arange(3))
In [105]: obj2=Series([1.5, -2.5, 0], index=index)
In [106]: obj2.index is index
Out[106]: True
```

pandas의 주요 Index 객체

클래스	설명
Index	가장 일반적인 Index 객체, 파이썬 객체의 NumPy 배열 형식으로 축의 이름을 표현한다.
Int64Index	정수 값을 위한 특수한 Index
MultiIndex	단일 축에 여러 단계의 색인을 표현하는 계층적 색인 객체, 튜플의 배열과 유사함
DatetimeIndex	나노초 타임스탬프를 저장한다.(NumPy의 datetime64 dtype으로 표현됨)
PeriodIndex	기간 데이터를 위한 특수한 Index

In [109]: **frame3**

Out[109]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

In [110]: **'Ohio' in frame3.columns**

Out[110]: True

In [111]: **2003 in frame3.index**

Out[111]: False

인덱스 객체의 메서드와 속성 (크게 활용 X)

메서드	설명
append	추가적인 Index 객체를 덧붙여 새로운 색인을 반환한다.
diff	색인의 차집합을 반환한다.
intersection	색인의 교집합을 반환한다.
union	색인의 합집합을 반환한다.
isin	넘겨받은 값이 해당 색인 위치에 존재하는지 알려주는 불리언 배열을 반환한다.
delete	i 위치의 색인이 삭제된 새로운 색인을 반환한다.
drop	넘겨받은 값이 삭제된 새로운 색인을 반환한다.
insert	i위치에 값이 추가된 새로운 색인을 반환한다.
is_monotonic	색인이 단조성을 가지면 True를 반환한다.
is_unique	중복되는 색인이 없다면 True를 반환한다.
unique	색인에서 중복되는 요소를 제거하고 유일한 값만을 반환한다.

5.2 핵심 기능

5.2.1 재색인

reindex 메서드

- 새로운 색인에 맞도록 객체를 새로 생성하는 기능
- reindex 호출시 데이터를 새로운 색인에 맞게 재배열하고 (사용자 지정 순서) , 없는 색인 값이 있다면 비어있는 값을 새로 추가한다.
- **reindex의 기본 값 : 행**

reindex 함수 인자

인자	설명
index	색인으로 사용할 새로운 순서. 색인은 deepcopy X, 그대로 사용된다.
method	보간 메서드
fill_value	재색인 과정에서 새로 나타난 빈 데이터를 채우기
limit	전/후 보간 시 사용할 최대 갭 크기
level	multiindex 단계에 단순색인을 맞춘다. 그렇지 않으면 멀티인덱스의 하위 부분집합에 맞춘다.
copy	True인 경우 새로운 색이 이전과 같더라도 데이터를 복사한다. False라면 두 색인이 같을 경우 데이터를 복사하지 않는다.
columns	columns=[,] 형태를 통해 columns의 재배치! 기본은 행 재배치

Null값 처리방법 옵션 in reindex 메서드

1. **fill_value** 옵션을 통해 Null값 지정
2. **method** 옵션을 통해 Null값을 이전 / 이후 값으로 채우기 가능

(**index가 이미 순차적**일 때만 가능) + 행에만 적용 가능

method = 'ffill' 또는 'pad' : 앞의 값으로 채워 넣는다.

method = 'bfill' 또는 'backfill' : 뒤의 값으로 채워 넣는다.

cf. **fill_na()** 함수 존재 , 함수 내에 전전값, 전전전값 등 더 다양한 범위 옵션 사용 가능

- > 정렬과 다른 점

1. 정렬 : 정해진 순서 (오름차순 / 내림차순)

reindex : 사용자 지정 순서

2. 정렬 : 가지고 있는 data 내에서만

reindex : index 나열로 원하는 값을 추가 + 삭제 가능

3. reindex - 컬럼/행 동시 재배치 가능

예제

--Series ~ reindex

In [112]: **obj = Series([4.5,7.2,-5.3,3.6], index=['d','b','a','c'])**

In [113]: **obj**

Out[113]:

d 4.5

b 7.2

a -5.3

c 3.6

dtype: float64

In [160]: **obj2=obj.reindex(['a','b','c','d','e'])**

In [161]: **obj2**

Out[161]:

a -5.3

b 7.2

c 3.6

d 4.5

e NaN

dtype: float64

In [162]: obj.reindex(['a','b','c','d','e'], fill_value=0) **# Null일시 지정 value설정 옵션**

Out[162]:

a -5.3

b 7.2

c 3.6

d 4.5

e 0.0

dtype: float64

In [163]: obj2.reindex(['a','b','c','d','f'],method='ffill') **# obj2는 인덱스가 순차적으로 존재하는 Series**

Out[163]:

a -5.3

b 7.2

c 3.6

d 4.5

f NaN

dtype: float64

--DataFrame ~ reindex

In [165]: **frame=DataFrame(np.arange(9).reshape((3,3)), index=['a','c','d'], col**
...: umns=['Ohio','Texas','California'])

In [166]: frame

Out[166]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [167]: **frame2=frame.reindex(['a','b','c','d'])** **# reindex 기본단위 - 행 / 재배치**

In [168]: **frame2**

Out[168]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

In [169]: **states=['Texas','Utah','California']**

In [170]: frame.reindex(['d','a'], **columns=states)** **# 열은 columns = 를 통해 재색인 할 수 있다.**

Out[170]:

	Texas	Utah	California
d	7	NaN	8
a	1	NaN	2

(i)loc[행,열] 형태로 색인

- .(i)loc method도 원하는 행 순서 + 추가/삭제 가능 !
- 여러 행 / 열 -> 리스트로 형태로 전달

In [10]: **frame.loc[['a','b','c','d'],states]**

Out[10]:

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

ix (loc-문자, iloc-정수) 를 이용해서 R처럼 간결하게 색인 가능!

5.2.2 하나의 로우 또는 칼럼 삭제하기

.drop 메서드

- **기본** : 행이름 기반 삭제
- **axis=1** 옵션 : col(key값) 삭제
- R에서 색인시 - 로 삭제 -> .drop 메서드 in python
- **--> 첫 번째 행 삭제 : data.drop('첫 번째 행 이름')**

```
# -- Series
In [13]: obj=Series(np.arange(5.), index=['a','b','c','d','e'])
In [14]: new_obj=obj.drop('c')    # .drop -- series에서 index 이름 사용
In [15]: new_obj
Out[15]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [17]: obj.drop(['d','c'])          # 여러 개 삭제할 경우 삭제할 index이름 리스트로 전달
Out[17]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

 # -- DataFrame

In [19]: **data=DataFrame(np.arange(16).reshape((4,4)),**
...: index=['Ohio','Colorado','Utah','New York'],
...: columns=['one','two','three','four'])

In [20]: **data.drop(['Colorado','Ohio'])**
Out[20]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

In [21]: **data.drop('two',axis=1)**
Out[21]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [22]: **data.drop(['two','four'],axis=1)**
Out[22]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

5.2.3. 색인하기, 선택하기, 거르기

 # Series에서의 색인

In [32]: **obj=Series(np.arange(4.), index=['a','b','c','d'])**

In [33]: **obj['b']**

Out[33]: 1.0

In [34]: **obj[1]**

Out[34]: 1.0

In [35]: **obj[2:4]**

Out[35]:

c 2.0

d 3.0

dtype: float64

In [36]: **obj[['b','d','a']]**

Out[36]:

b 1.0

d 3.0

a 0.0

dtype: float64

In [38]: **obj[obj<2]** **# 조건 색인도 가능**

Out[38]:

a 0.0

b 1.0

dtype: float64

In [39]: **obj['b':'c']** **# 라벨 이름으로 슬라이싱 : 시작점과 끝값을 포함**

Out[39]:

b 1.0

c 2.0

dtype: float64

In [40]: **obj['b':'c']=5**

In [41]: **obj**

Out[41]:

a 0.0

b 5.0

c 5.0

d 3.0

dtype: float64

#DataFrame에서의 색인

['컬럼명'] ~ 컬럼색인 **cf. 열 순서로 하려면 .iloc**

[: 슬라이싱], [불리언] ~ row색인 **cf. 행 이름으로 하려면 .loc**

In [42]: **data**

Out[42]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [43]: **data['two']**

Out[43]:

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

In [44]: **data[['three','one']]** **# 여러 개 지정 시 리스트 형태**

Out[44]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

슬라이싱으로 로우를 선택하기

In [56]: **data[:2]**

Out[56]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

불리언 배열로 선택하기

In [57]: **data[data['three']>5]**

Out[57]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

과정

In [58]: **data<5**

Out[58]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [59]: **data[data<5]=0**

In [60]: **data**

Out[60]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

loc / iloc를 통해 로우+칼럼 동시 색인 가능 (R과 비슷)

In [159]: **data.loc['Colorado', ['two', 'three']]**

행색인 칼럼색인(여러 개--리스트로 전달)

Out[159]:

two 5

three 6

Name: Colorado, dtype: int32

In [163]: **data.iloc[2]**

Out[163]:

one 8

two 9

three 10

four 11

Name: Utah, dtype: int32

In [164]: **data.loc[:'Utah','two']** **# 이름색인 : 끝 범위 포함**

Out[164]:

Ohio 1

Colorado 5

Utah 9

Name: two, dtype: int32

In [165]: **data.loc[data.three>5, :'three']** **# 조건도 가능!**

Out[165]:

	one	two	three
Colorado	4	5	6
Utah	8	9	10
New York	12	13	14

 # loc / iloc -> 문자색인 / 정수색인, 서로 혼용 불가

 # ix는 혼용 가능

방식	설명
obj[val]	DataFrame에서 하나의 칼럼 / 여러 칼럼을 선택한다. 편의를 위해 불리언 배열, 슬라이스, 불리언 DataFrame을 사용할 수 있다.
obj.loc[val]	DataFrame에서 로우의 부분집합을 선택한다.
obj.loc[:,val]	DataFrame에서 칼럼의 부분집합을 선택한다.
obj.loc[val1,val2]	DataFrame에서 로우와 칼럼의 부분집합을 선택한다.
reindex 메서드	하나 이상의 축을 사용자가 원하는 색인으로 맞춘다.
xs메서드	라벨 이름으로 단일 로우나 칼럼을 Series 형식으로 선택한다.
icol, irow 메서드	각각 정수색인으로 단일 로우나 칼럼을 Series 형식으로 선택한다.
get_value, set_value 메서드	로우와 칼럼 이름으로 DataFrame의 값을 선택한다.

5.2.4 산술연산과 데이터 정렬

- pandas에서 객체를 더할 때 짝이 맞지 않는 색인이 있다면 결과에 두 색인이 통합됨
- 서로 겹치는 색인이 없다면 데이터는 NA값이 됨.
- NaN(결측치) 처리 ? .add 등의 method에서 option 지정

In [166]: **s1=Series([7.3,1.2,3.4,-1.2],index=['a','c','d','e'])**
In [168]: **s2=Series([-2.1,1.2,0.5,1.8,1.5],index=['a','c','e','f','g'])**
In [169]: **s1+s2**
Out[169]:

```
a    5.2
c    2.4
d    NaN
e   -0.7
f    NaN
g    NaN
dtype: float64
```

In [172]: **df1=DataFrame(np.arange(9).reshape(3,3),**
columns=list('bcd') ,index=['Ohio','Texas','Colorado'])
 # list('bcd') = ['b','c','d] 세 개로 전달됨!

In [173]: **df2=DataFrame(np.arange(12).reshape(4,3), columns=list('bde'),**
index=['Utah','Ohio','Texas','Oregon'])
 # list('bde')=['b','d','e'] 세 개로 전달됨!

In [174]: **df1**
Out[174]:

	b	c	d
Ohio	0	1	2
Texas	3	4	5
Colorado	6	7	8

In [175]: **df2**
Out[175]:

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

In [176]: **df1+df2**

Out[176]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

 # broadcasting 적용 X, 매칭되는 값 끼리만 연결한다. 매칭되지 않으면 NaN!

산술연산 메서드에 채워 넣을 값 지정하기

- fill_value 옵션을 통해 NaN(결측치)를 처리할 수 있다.
- 서로 다른 데이터 프레임끼리 산술연산 시 더 권장함

산술연산 메서드

메서드	설명
add	덧셈(+)을 위한 메서드
sub	뺄셈(-)을 위한 메서드
div	나눗셈(/)을 위한 메서드
mul	곱셈(*)을 위한 메서드
mod	나머지(%) 연산을 위한 메서드
pow	제곱(**) 연산을 위한 메서드

```
In [179]: df1=DataFrame(np.arange(12).reshape((3,4)), columns=list('abcd'))
In [180]: df2=DataFrame(np.arange(20).reshape((4,5)), columns=list('abcde'))
In [181]: df1
Out[181]:
   a  b   c   d
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
In [182]: df2
Out[182]:
    a   b   c   d   e
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
In [183]: df1.add(df2, fill_value=0)        # 결측치 값을 0으로 만듦
Out[183]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0  11.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

cf. reindex로 재색인 시 fill_value 옵션 사용 가능

In [184]: **df1.reindex(columns=df2.columns, fill_value=0)**

Out[184]:

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

Data Frame과 Series 간의 연산

 # Series의 index를 DataFrame의 Column에 맞추고 로우에 전파함
 # 만약 찾을 수 없다면 형식을 맞추기 위해 새로 추가 됨
 # 로우에 대해 연산을 수행하고 싶으면 산술연산 메서드 사용(axis=0 옵션 중요!!)

In [191]: **frame=DataFrame(np.arange(12).reshape((4,3)), columns=list('bde'),**
index=['Utah','Ohio','Texas','Oregon'])
In [192]: **series=frame.iloc[0]**
In [193]: **frame**
Out[193]:

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

In [194]: **series**
Out[194]:
b 0
d 1
e 2
Name: Utah, dtype: int32

In [195]: **frame-series**
Out[195]:

	b	d	e
Utah	0	0	0
Ohio	3	3	3
Texas	6	6	6
Oregon	9	9	9

In [196]: **series2=Series(range(3), index=list('bef'))**
In [197]: **frame+series2**
Out[197]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

In [200]: **series3=frame['d']**

In [201]: **frame**

Out[201]:

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

In [202]: **series3**

Out[202]:

```
Utah      1
Ohio      4
Texas     7
Oregon    10
Name: d, dtype: int32
```

In [203]: **frame.sub(series3, axis=0)** **# key 무시 후 행별 연산 : 브로드캐스팅 - axis 옵션 필요**

Out[203]:

	b	d	e
Utah	-1	0	1
Ohio	-1	0	1
Texas	-1	0	1
Oregon	-1	0	1

cf. frame.sub(series3) -- 모두 NaN (왜냐면 series3의 index이름->frame의 col이름에 없음)

In [204]: **frame.sub(series3)**

Out[204]:

	Ohio	Oregon	Texas	Utah	b	d	e
Utah	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Ohio	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Texas	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Oregon	NaN	NaN	NaN	NaN	NaN	NaN	NaN

cf. 배열(in numpy)의 연산

```
In [185]: arr=np.arange(12).reshape((3,4))

In [186]: arr
Out[186]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [187]: arr[0]
Out[187]: array([0, 1, 2, 3])

In [188]: arr-arr[0]
Out[188]:
array([[0, 0, 0, 0],
       [4, 4, 4, 4],
       [8, 8, 8, 8]])
```

 # 배열-> size 동일해야 함.

그런데 맞지 않을 때에는 broadcasting 사용--> 행/열 하나만 size 맞아도 괜찮다.

5.2.5 함수 적용과 매핑

- pandas 객체에도 NumPy의 유니버설 함수(배열의 각 원소에 적용되는 메서드) 적용 가능
- apply 메서드 : 행 / 열 별
- applymap 메서드 : 원소별

(axis=0(열끼리) 기본값 , axis=1 행끼리)

NumPy 메서드 사용

In [205]: **frame=DataFrame(np.random.randn(4,3), columns=list('bde'),**
index=['Utah','Ohio','Texas','Oregon'])

In [206]: **frame**

Out[206]:

	b	d	e
Utah	-0.373095	-0.943785	-1.341892
Ohio	-0.523223	-0.412099	-0.488077
Texas	-1.211279	1.447290	-0.498082
Oregon	1.305385	0.408180	-0.107880

In [207]: **np.abs(frame)** **# np의 메서드**

Out[207]:

	b	d	e
Utah	0.373095	0.943785	1.341892
Ohio	0.523223	0.412099	0.488077
Texas	1.211279	1.447290	0.498082
Oregon	1.305385	0.408180	0.107880

apply 메서드 사용

In [208]: **f = lambda x : x.max() - x.min()**

In [209]: **frame.apply(f)**

Out[209]:

```
b       2.516664
d       2.391075
e       1.234011
dtype: float64
```

In [210]: **frame.apply(f, axis=1)**

Out[210]:

```
Utah          0.968797
Ohio          0.111124
Texas         2.658569
Oregon        1.413265
dtype: float64
```

 # apply -> 동시에 여러 값을 반환하는 함수도 가능

In [211]: **def f(x) : # Series로 list 묶어서 여러 개 한 번에 반환**
...: **return Series([x.min(), x.max()], index=['min','max'])**

In [212]: **frame.apply(f) # 열끼리 함수 적용(기본값)**

Out[212]:

	b	d	e
min	-1.211279	-0.943785	-1.341892
max	1.305385	1.447290	-0.107880

과정

1. frame 열끼리 함수 적용 -- 원 열의 이름 b , d , e 고정
2. 각 col마다 Series[ ] 동시 return (열로)

cf. f(x)에서는 index였지만 col이름이 될 수도 있다. - 기존 df 행/열 위치 우선

In [214]: **frame.apply(lambda x : Series([x.max(),x.min()], index=['max','min']), axis=1)**

Out[214]:

	min	max
Utah	-1.341892	-0.373095
Ohio	-0.523223	-0.412099
Texas	-1.211279	1.447290
Oregon	-0.107880	1.305385

 # 행끼리 함수 적용

cf. 나중에 산술 기초통계 관련해서 함수 만들 때 Series로 묶는 명령어 만들어 사용

In [29]: **dff.apply(sum,axis=1) # 그룹함수이므로 산술연산도 가능**

Out[29]:

0 10
1 26
2 42
dtype: int64

In [24]: **f2 = lambda x : x.max() - x.min()**

 # x에 들어가야 하는 자료는 벡터 --> apply 사용

 # lambda 사용 시 x의 형태 생각하면서 만들기

applymap → 포맷변경시 무조건 !!

In [215]: **format=lambda x : '%.2f' % x** **# 이 때 출력되는 값은 str!**
In [216]: **frame.applymap(format)** **# 데이터프레임만 사용 가능한 메서드**
Out[216]:

	b	d	e
Utah	-0.37	-0.94	-1.34
Ohio	-0.52	-0.41	-0.49
Texas	-1.21	1.45	-0.50
Oregon	1.31	0.41	-0.11

In [23]: **dff.applymap(f1)**
Out[23]:

	0	1	2	3
0	1.00	2.00	3.00	4.00
1	5.00	6.00	7.00	8.00
2	9.00	10.00	11.00	12.00

map 메서드 사용? -- 특정 col 선택시

In [217]: **frame['e'].map(format)** **# 특정 col (Series) -> map사용**

Data Frame -> applymap 사용

Out[217]:

Utah -1.34

Ohio -0.49

Texas -0.50

Oregon -0.11

Name: e, dtype: object

############################

- map의 전제조건 : **1차원 대상으로** 원소별로 기능을 전달함

In [5]: **f1 = lambda x : '%.2f' % x**

In [6]: **l1=[1,2,3,4,5]**

In [7]: **list(map(f1,l1))**

Out[7]: ['1.00', '2.00', '3.00', '4.00', '5.00']

In [8]: **dff=DataFrame(np.arange(1,13).reshape(3,4))**

In [9]: **dff**

Out[9]:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

In [10]: **list(map(f1, dff))**

Out[10]: ['0.00', '1.00', '2.00', '3.00'] **# → col 이름이 전달됨**

=> map의 한계

2차원 형태(DataFrame)일 때에는 2차원의 모든 원소를 전달하지 않는다.

한 번의 일차원만 처리하고 끝 (반복 X)

dff.map(f1)의 결과

AttributeError: 'DataFrame' object has no attribute 'map'

dff.apply(f1)의 결과

TypeError: ("cannot convert the series to <class 'float'>", 'occurred at index 0')

특정 컬럼만 처리는 map으로 가능

In [18]: **list(map(f1,dff.ix[:,1]))**

Out[18]: ['2.00', '6.00', '10.00']

---

index나 columns의 이름을 바꾸고 싶을 때도 사용할 수 있겠다.

In [21]: **dff.index.map(f1)** **# dff.index()는 series임**

Out[21]: Index(['0.00', '1.00', '2.00'], dtype='object')

map은 함수도 가능하고 메서드도 가능함! ,

단, map 메서드는 시리즈일 때만 사용 가능 (리스트는 시리즈로 만든 후)

-- pandas 용으로 만들어졌기 때문에 1차원인 series에 적용함

메서드는 기능이 제한되는 점이 있어서 함수를 쓰는 편을 권장한다.

map, apply, applymap의 차이 -- 메서드

-map : 특정 col 설정할 때 사용. Series 적용 가능 (index 적용 가능)

-> index 이름 변경할 때 주로 사용

-apply : Series, DataFrame 가능, **index 적용 불가** , 각 축별 적용(col별, row별)

-> mean, std등 **그룹연산 가능**

-applymap : DataFrame에서만 가능, 각 요소별 적용, => **mean, std등 그룹연산 불가**

절대 Series 등 특정 col만 선택한 형태엔 적용 **불가능**

[문제1]

1. 5*4의 구조를 갖는 임시 배열을 생성 한 후 텍스트 파일 형태로 저장하여라

In [115]: **arr1=np.arange(20).reshape(5,4)**

In [117]: **np.savetxt('q1.txt',arr1, fmt='%d')**

np.savetxt 의 dtype 기본값이 굉장히 복잡함! **(%.18e)**

따라서 정수로 저장하고 싶다면 **fmt='%d'** 작성

np.savetxt? 로 도움말 -- **매뉴얼 확인 습관 들이기**

만일 줄 맞추고 싶으면 fmt='%2d' 작성

2. 다음의 구조를 갖는 DataFrame을 생성하자.

단, 텍스트 파일 형태로 저장되어 있는 파일을 불러온 후 DataFrame 생성

```
1   500    5
2   200    2
3   200    7
4    50    9
```

	name	price	qty
1	1	500	5
2	2	200	2
3	3	200	7
4	4	50	9

==========

강사님 방법

In [79]: **df2=DataFrame(np.loadtxt("q2.txt",dtype='int'))**

#dtype ! 앞 문제의 fmt과 다름 ! 매뉴얼 확인 습관 들이기

In [83]: **df2.index=[1,2,3,4]** **# 메서드로 수정**

In [84]: **df2.columns=['name','price','qty']**

In [85]: **df2**

Out[85]:

	name	price	qty
1	1	500	5
2	2	200	2
3	3	200	7
4	4	50	9

내 방법

In [145]:

df2=DataFrame(np.loadtxt('q2.txt',dtype='int'),index=['1','2','3','4'],columns=['name','price','q
...: ty'])

In [146]: **df2**

Out[146]:

	name	price	qty
1	1	500	5
2	2	200	2
3	3	200	7
4	4	50	9

3. 위의 DataFrame 구조에서 각 행 이름을 'apple','banana','orange','peach'로 변경

In [149]: **df2.index=['apple','banana','orange','peach']** **# index 사용시 일부 X, 모두 바꾸기 O**

In [150]: **df2**

Out[150]:

	name	price	qty
apple	1	500	5
banana	2	200	2
orange	3	200	7
peach	4	50	9

4. name 컬럼 뒤에 code라는 컬럼을 생성한 후 각각 값을 c001 c002 c003 c004로 변경

#1

In [153]: **df3['code']=['c001','c002','c003','c004']**

In [154]: **df3**

Out[154]:

	name	price	qty	code
apple	1	500	5	c001
banana	2	200	2	c002
orange	3	200	7	c003
peach	4	50	9	c004

cf. reindex ->사용자 지정 순서로 재배치 + 추가/삭제 가능 // 변경 X, 출력

정렬해보기

In [96]: df2.reindex(columns=['name','qty','code','price'])

Out[96]:

	name	qty	code	price
1	1	5	c001	500

2	2	2	c002	200
3	3	7	c003	200
4	4	9	c004	50

#2

In [99]: **DataFrame(df2,columns=['name','code','price','qty'])**

원래 있는 key값을 DF 함수 통해 할 때는 재배치! not 새로 작성!! 이렇게 정렬할 수도 있음

이 명령을 변수로 넣은 후 code값을 수정하기. # 이 방식은 reindex도 마찬가지

Out[99]:

	name	code	price	qty
1	1	NaN	500	5
2	2	NaN	200	2
3	3	NaN	200	7
4	4	NaN	50	9

5. apple과 orange의 price와 qty 데이터만 출력하도록 하자

In [155]: **df3.loc[['apple','orange'],['price','qty']]**

Out[155]:

	price	qty
apple	500	5
orange	200	7

행 선택 후 col 선택 -- 동시에 쓰는 경우

In [103]: **df3[['price','qty']].loc[['apple','orange']]**

Out[103]:

	price	qty
apple	500	5
orange	200	7

#col 선택 후 행 선택

+

df3.reindex(['apple','orange'])[['price','qty']]

6. qty가 4보다 큰 과일은 몇 개가 있는지 확인해보자

In [157]: **sum(df3['qty']>4)** **# 함수**

Out[157]: 3

In [115]: **(df2['qty']>4).sum()** **# 메서드**

Out[115]: 3

[문제2]

[데이터]

7.5	3.6	3.5	3.3	1.5
7.4	3.2	3	2.8	1.2
6.6	2.9	2	2	1.1
7.7	3	2.2	2.2	1
7.9	3.1	2.3	2.3	1.2
7.7	3.3	2.6	2.5	1.3
7.7	3	2.3	2.2	1.4
7.1	3.2	2	2.1	1.4
7	3.1	2.1	2	1.2
7.9	3.6	2.5	2.5	1.6
7.8	3.5	2.5	2.4	3
7.4	3.4	2.1	2.1	2.7
7.5	3	2.1	2.1	2.5
7.9	3	2	1.9	1.9

[문제]

1. 다음의 데이터를 읽어들여 다음과 같은 프레임 형태로 변경

	20대	30대	40대	50대	60세이상
2000년	7.5	3.6	3.5	3.3	1.5
2001년	7.4	3.2	3	2.8	1.2
2002년	6.6	2.9	2	2	1.1
.....					
2011년	7.4	3.4	2.1	2.1	2.7
2012년	7.5	3	2.1	2.1	2.5
2013년	7.9	3	2	1.9	1.9

강사님 방법

In [3]: **q2=np.loadtxt('qq2.txt')**

In [119]: **df4=DataFrame(q2)**

In [122]: **ll=np.arange(2000,2014)**

 # 지금은 마지막 숫자를 알 수 있지만, 데이터가 너무 많을 땐 길이로 !

len(df4.index) - 행n / len(df4.columns) - 열 n

In [123]: **f2=lambda x : str(x) + '년'**

 # 숫자 + 문자 --> 오류, 앞에를 문자로 바꿔줘야 결합됨

In [125]: **nm=list(map(f2,ll))**

In [126]: **df4.index=nm**

In [120]: **df4.columns=['20대','30대','40대','50대','60세 이상']**

내 방법

In [3]: **q2=DataFrame(np.loadtxt('qq2.txt',dtype='int'),columns=['20대','30대','40대','50대','60대이상'])**

In [16]: **q2.index=[str(i)+'년' for i in range(2000,2014)]**

2. 2010년부터의 20~40대 실업률만 추출하여 새로운 데이터프레임을 만들어라

In [142]: **df4.loc['2010년':, '20대':'40대']**

Out[142]:

	20대	30대	40대
2010년	7.8	3.5	2.5
2011년	7.4	3.4	2.1
2012년	7.5	3.0	2.1
2013년	7.9	3.0	2.0

 # 문자슬라이싱 : 시작, 끝값 포함

3. 전체 데이터에서 각 년도별 20대와 30대의 실업률 차이를 구하라(20대 - 30대)

In [46]: **df1['20대'].sub(df1['30대'])**

 # 또 다른 키 값 접근 - method 형태 // 그런데 df1.20대는 불가능 (숫자+문자인 경우 안 됨!)

 # 만일 다른 데이터프레임인 경우 .add / .sub 등의 메서드 활용이 좋음.

why? NaN값을 채워줄 수 있음. (fill_value 옵션)

Out[46]:

2000년 3.9
2001년 4.2
2002년 3.7
2003년 4.7
2004년 4.8
2005년 4.4
2006년 4.7
2007년 3.9
2008년 3.9
2009년 4.3
2010년 4.3
2011년 4.0
2012년 4.5
2013년 4.9
dtype: float64

4. 모든 실업률 지수를 소수점 둘째 짜리로 표현하여라

In [53]: **f2=lambda x : '%.2f' % x** **# format 변경 함수 -> 원소별로 해야 함 !**

In [54]: **df1.applymap(f2)** **# applymap : 원소별**

Out[54]:

	20대	30대	40대	50대	60대 이상
2000년	7.50	3.60	3.50	3.30	1.50
2001년	7.40	3.20	3.00	2.80	1.20
2002년	6.60	2.90	2.00	2.00	1.10
2003년	7.70	3.00	2.20	2.20	1.00
2004년	7.90	3.10	2.30	2.30	1.20
2005년	7.70	3.30	2.60	2.50	1.30
2006년	7.70	3.00	2.30	2.20	1.40
2007년	7.10	3.20	2.00	2.10	1.40
2008년	7.00	3.10	2.10	2.00	1.20
2009년	7.90	3.60	2.50	2.50	1.60
2010년	7.80	3.50	2.50	2.40	3.00
2011년	7.40	3.40	2.10	2.10	2.70
2012년	7.50	3.00	2.10	2.10	2.50
2013년	7.90	3.00	2.00	1.90	1.90

#In [47]: f2 = lambda x : round(x,2) --> 원 데이터랑 같게 나옴 이유는 원래 한 자리 자료여서!

cf. apply-> 시리즈, 데이터프레임 사용 가능.
데이터프레임 : 행 / 열 별 적용
시리즈 : 경우 원소별 적용(포괄적) / 원소별 적용할 때엔 map 사용이 더 정확한 의미.
+ 그룹함수보다 메서드 ex(sum() / .sum) 가 더 빠름! + 메서드 : 벡터연산 가능

5. 연령별 실업률의 평균과 연도별 실업률 평균을 각각 구하여라(np.average)

In [56]: **df1.apply(np.average, axis=0)** **# apply 함수 -> 그룹함수와 같이 쓰임, axis=0 기본값**

Out[56]:

```
20대          7.507143
30대          3.207143
40대          2.371429
50대          2.314286
60대 이상      1.642857
dtype: float64
```

In [57]: **df1.apply(np.average, axis=1)**

Out[57]:

```
2000년    3.88
```

2001년 3.52
2002년 2.92
2003년 3.22
2004년 3.36
2005년 3.48
2006년 3.32
2007년 3.16
2008년 3.08
2009년 3.62
2010년 3.84
2011년 3.54
2012년 3.44
2013년 3.34
dtype: float64

6. 전체 데이터에서 20대 실업률이 7이하인 연도만 출력하여라.
In [65]: **df1[df1['20대']<=7].index**
Out[65]: Index(['2002년', '2008년'], dtype='object')
 # df1.index[df1['20대']<=7] 도 가능!!

7. 각 년도 별 총 실업률에서 20대 실업률이 차지하고 있는 비율을 출력하여라.(함수생성)
내 방법
In [71]: **f3=lambda x : x[0] / x.sum() * 100**
 # f3=lambda x : x[0] / sum(x) * 100 함수 -> for문 형태 --> 메서드 사용 권장

 # applymap을 쓰면 원소 하나씩 전달하는 과정이 맞지만, 그 원소가 포함된 행 / 열 처리도 가능함

In [72]: **df1.apply(f3,axis=1)**
Out[72]:
2000년 38.659794
2001년 42.045455
2002년 45.205479
2003년 47.826087
2004년 47.023810
2005년 44.252874
2006년 46.385542
2007년 44.936709
2008년 45.454545

2009년 43.646409
2010년 40.625000
2011년 41.807910
2012년 43.604651
2013년 47.305389
dtype: float64

 # cf. 년도별 비율 ~ 총 합을 구하고 나누기

In [162]: **df4.div(df4.apply(sum, axis=1), axis=0)**

 # .div 메서드를 통해 축 지정 -- 시리즈 ~ 데이터프레임 브로드캐스팅 in NumPy

Out[162]:

	20대	30대	40대	50대	60세 이상
2000년	0.386598	0.185567	0.180412	0.170103	0.077320
2001년	0.420455	0.181818	0.170455	0.159091	0.068182
2002년	0.452055	0.198630	0.136986	0.136986	0.075342
2003년	0.478261	0.186335	0.136646	0.136646	0.062112
2004년	0.470238	0.184524	0.136905	0.136905	0.071429
2005년	0.442529	0.189655	0.149425	0.143678	0.074713
2006년	0.463855	0.180723	0.138554	0.132530	0.084337
2007년	0.449367	0.202532	0.126582	0.132911	0.088608
2008년	0.454545	0.201299	0.136364	0.129870	0.077922
2009년	0.436464	0.198895	0.138122	0.138122	0.088398
2010년	0.406250	0.182292	0.130208	0.125000	0.156250
2011년	0.418079	0.192090	0.118644	0.118644	0.152542
2012년	0.436047	0.174419	0.122093	0.122093	0.145349
2013년	0.473054	0.179641	0.119760	0.113772	0.113772

In [165]: **f3=lambda x : x / x.sum() * 100**

In [166]: **df4.apply(f3,axis=1)**

Out[166]:

	20대	30대	40대	50대	60세 이상
2000년	38.659794	18.556701	18.041237	17.010309	7.731959
2001년	42.045455	18.181818	17.045455	15.909091	6.818182
2002년	45.205479	19.863014	13.698630	13.698630	7.534247
2003년	47.826087	18.633540	13.664596	13.664596	6.211180

2004년	47.023810	18.452381	13.690476	13.690476	7.142857
2005년	44.252874	18.965517	14.942529	14.367816	7.471264
2006년	46.385542	18.072289	13.855422	13.253012	8.433735
2007년	44.936709	20.253165	12.658228	13.291139	8.860759
2008년	45.454545	20.129870	13.636364	12.987013	7.792208
2009년	43.646409	19.889503	13.812155	13.812155	8.839779
2010년	40.625000	18.229167	13.020833	12.500000	15.625000
2011년	41.807910	19.209040	11.864407	11.864407	15.254237
2012년	43.604651	17.441860	12.209302	12.209302	14.534884
2013년	47.305389	17.964072	11.976048	11.377246	11.377246

DF 구조

dfdata[ ] 색인 한 개

1. 이름 -- 컬럼이름
2. 슬라이스 색인 -- 행

+

넘파이의 메서드() - > 벡터연산 가능함 + 빠름

+

sum,sub,div 메서드 -> 축 설정 가능 + apply보다 빠름

++

내가 만든 함수명이 f1일 경우 f1?? 으로 코드 확인

0625

5. 2. 6. 정렬과 순위

-- sort, sorted 가능하지만 인덱스 활용 관련한 메서드는 아래의 메서드들

.sort_index

-- **컬럼 / 행의 이름별**로 정렬할 때 사용하는 인덱스

-- 기본 축 값 axis=0, row별 정렬

-- 축 값 **axis=1** 지정시 col별 정렬

-- 내림차순 정렬시 **ascending=False** 옵션 사용

(매뉴얼 확인 후 조금 다른 옵션으로 쓰임 확인하기 !)

cf. reindex는 각각 지정해야 하는 불편함. 한 번에 처리할 수 있는 sort_index 메서드 (아니면 index값을 먼저 전달한 후 reindex로 전달해도 괜찮)

#시리즈

In [2]: **obj=Series(range(4),index=list('badc'))**

In [3]: **obj.sort_index()**

Out[3]:

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

#데이터프레임

In [4]: **frame=DataFrame(np.arange(8).reshape((2,4)), index=['three','one'], columns=list('dabc'))**

In [6]: **frame.sort_index()**

Out[6]:

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

In [8]: **frame.sort_index(axis=1)**

Out[8]:

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

.sort_values()

-- **원소별** 크기 재정렬

-- 값이 정렬되어 바로 출력됨 (R에서의 sort()와 비슷)

-- **na_position ='first' / 'last' (default)** Null값을 맨 앞 / 맨 뒤 중 위치를 어떻게 표현할지

-- **by=컬럼명 옵션 in DataFrame --** (oracle에서의 order by)

여러 개 이상의 column을 전달할 때 순서 중요! 정렬의 1순위 앞 컬럼 / 2순위 뒤 컬럼

-- **axis=0(기본값)/1 옵션**. 기본 행별 정렬. col별 정렬할 이유 거의 없음..!

-- **ascending=**True(기본값) - 오름차순 / False - 내림차순

```
# in Series
In [61]: obj=Series([4,6,-3,2])
In [62]: obj.sort_values()
Out[62]:
2    -3
3     2
0     4
1     6
dtype: int64

In [63]: obj=Series([4, np.nan, 7, np.nan, -3, 2])    #결측치 : 기본적으로 맨 마지막에 정렬됨
In [64]: obj.sort_values()
Out[64]:
4    -3.0
5     2.0
0     4.0
2     7.0
1     NaN
3     NaN
dtype: float64


#in DataFrame
In [65]: frame=DataFrame({'b':[4,7,-3,2],'a':[0,1,0,1]})
In [66]: frame
Out[66]:
   b  a
0  4  0
1  7  1
2 -3  0
```

3 2 1

In [67]: **frame.sort_values(by='b')** **# 칼럼에 있는 값으로 정렬시 by=옵션 사용**

Out[67]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

In [68]: **frame.sort_values(by=['a','b'])**

 # 여러 개의 칼럼을 정렬하려면 칼럼의 이름이 담긴 리스트를 전달. 작성한 순서대로 정렬됨

Out[68]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

.rank() 순위 메서드
-- 순위는 정렬과 거의 흡사하며 1부터 배열의 유효한 데이터 개수까지 순위를 매김
-- 동률 순위 처리하는 방식이 numpy.argsort의 간접 정렬 색인과 다름
-- 동률을 처리하는 default 방법은 평균 (4th, 5th 값이 같음 -> (4+5)/2 = 4.5 둘 다 4.5)
-- 순위를 출력
-- **method=average'** 기본값
-- **method='first' 옵션** --같은 값일 경우 먼저 발견된 것에 높은 순위를 부여함(not평균 순위)
-- method = 'min' 옵션 -- 같은 값을 가지는 그룹을 낮은 순위로 매김
-- **method='max'** 같은 값을 가지는 그룹을 높은 순위로 매김
-- ascending=False 내림차순 순서대로 순서를 매김

```
# in Series
In [69]: obj=Series([7,-3,7,4,2,0,4])
In [70]: obj.rank()
Out[70]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64

In [71]: obj.rank(method='first')  # 데이터 상 나타나는 순서에 따라 순위를 매김
Out[71]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64

In [72]: obj.rank(ascending=False, method='max')  # 내림차순 순위 매기기
Out[72]:
0    2.0
1    7.0
2    2.0
```

```
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

 # in DataFrame -> 로우나 칼럼에 대해 순위를 정할 수 있다.

```
In [73]: frame=DataFrame({'b':[4.3,7,-2,2], 'a':[0,1,0,1], 'c':[-2,5,8,-2.5]})
In [74]: frame
Out[74]:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -2.0  0  8.0
3  2.0  1 -2.5
In [75]: frame.rank(axis=1)
Out[75]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

5. 2. 7 중복 색인

- 색인 값은 중복될 수 있다. 하지만 중복색인만! 쓰는 경우는 없음.
- 중복색인을 쓴다면 멀티인덱스를 추가로 설정.

#in Series

In [76]: **obj=Series(range(5), index=['a','a','b','b','c'])**

In [77]: **obj**

Out[77]:

a 0

a 1

b 2

b 3

c 4

dtype: int64

In [78]: **obj.index.is_unique** **# .is_unique : 해당 값이 유일한지 아닌지 알려줌**

Out[78]: False

In [79]: **obj['a']** **# 중복되는 색인 값이 있으면 색인을 이용한 데이터 선택은 Series로 반환**

Out[79]:

a 0

a 1

dtype: int64

--> 중복 색인이 가능하다는 의미로만 기억하기! 이후 멀티인덱스에서 더 자세히 살펴보기

5.3 기술통계 계산과 요약

- Numpy 배열의 연산과는 다르게 **누락된 데이터를 제외하도록 설계**하여 연산이 수월함
- 배열 구조인 경우 넘파이의 메서드 적용됨
- 데이터프레임 구조인 경우 판다스의 메서드 적용됨

sum / mean 등의 메서드

축소 메서드 옵션

옵션	설명
axis	연산을 수행할 축. axis=0 기본값
skipna	누락된 값을 제외할 것인지 정하는 옵션. 기본 값은 True
level	계산하려는 축이 계층적 색인(다중 색인)이라면 레벨에 따라 묶어서 계산한다.

★★ .idxmax메서드 -- 중요함!!!!! ★★

- 각 축별 최대 값을 갖는 **index값** 리턴
- 열 이름 + **index 이름** 출력
- axis 옵션으로 축 지정 가능 axis=0(기본값)
- axis=1일 경우 행별 최대값을 구하여 **column 이름** 출력

In [88]: **df**

Out[88]:

	s	h	i	n
one	-3.294332	1.498115	-0.797540	0.340189
two	-1.172887	-0.456873	0.633697	1.406143
three	1.159694	0.341467	0.942452	0.499382
foru	0.125035	-1.683702	-1.363560	1.292295

In [89]: **df.idxmax()**

Out[89]:

```
s      three
h        one
i      three
n        two
dtype: object
```

In [90]: **df.idxmax(axis=1)**

Out[90]:

```
one        h
two        n
three      s
foru       n
dtype: object
```

.cumsum() 메서드

- 누적합 (배열과 같음)
- NA 제외하고 계산됨 (numpy 랑 다른 점 !)
- NA값이 있는 경우 fillna 등의 함수를 통해 NaN을 앞의 값으로 대치하는 처리 방법 필요해 보임

In [93]: **DataFrame(np.arange(16).reshape(4,4)).cumsum()**

Out[93]:

	0	1	2	3
0	0	1	2	3
1	4	6	8	10
2	12	15	18	21
3	24	28	32	36

.describe() 메서드

- 숫자 : 기초통계 정보
- 숫자 외 : 다른 요약통계

In [95]: **df.describe()**

Out[95]:

	s	h	i	n
count	4.000000	4.000000	4.000000	4.000000
mean	-0.795622	-0.075248	-0.146238	0.884502
std	1.919787	1.339388	1.110497	0.542525
min	-3.294332	-1.683702	-1.363560	0.340189
25%	-1.703248	-0.763580	-0.939045	0.459584
50%	-0.523926	-0.057703	-0.081921	0.895838
75%	0.383700	0.630629	0.710886	1.320757
max	1.159694	1.498115	0.942452	1.406143

In [98]: **obj=Series(['a','a','b','c'] * 4)**

In [99]: **obj.describe()**

Out[99]:

count	16
unique	3
top	a
freq	8

dtype: object

기술통계와 요약통계와 관련된 메서드

메서드	설명
value_counts	NA값을 제외한 수를 반환한다. Series만 사용 가능
describe	각 칼럼에 대한 요약통계를 계산한다.
min, max	최소/ 최대 값을 계산한다.
argmin, argmax	**각각 최소, 최대 값을 갖고 있는 색인의 위치(정수)를 반환한다. (in Series)**
idxmin, idxmax	**각각 최소, 최대 값을 갖고 있는 색인의 값을 반환한다.**
quantile	0부터 1까지의 분위수를 계산한다.
sum	합을 계산한다.
mean	평균을 계산한다.
median	중간 값(50% 분위)을 반환한다.
mad	평균 값에서 절대 평균편차를 구한다.
var	표본 분산의 값을 구한다.
std	표본 정규 분산의 값을 구한다.
skew	표본 비대칭도(3차 적률)의 값을 구한다.
kurt	표본 첨도(4차 적률)의 값을 구한다.
cumsum	누적 합을 구한다.
cummin, cummax	각각 누적 최소값과 누적 최대 값을 계산한다.
cumprod	누적 곱을 구한다.
diff	1차 산술 차를 구한다. (시계열 데이터 처리 시 유용하다.)
pct_change	퍼센트 변화율을 계산한다.

[문제]

disease 파일을 불러들여 다음을 수행

data1=pd.read_table('disease.txt',encoding='cp949', sep='₩s+')

(참고: cp949(윈도우 한글 포맷), ₩s+(공백이 한 개 이상)

In [129]: **data1.index=data1.월별**

In [136]: **data2=data1.drop('월별',axis=1)**

 # data1.drop(columns='월별') 도 가능

~ 특정 칼럼만 제거하는 drop 메서드 (R에서의 마이너스)

 # + 같은 뜻 data1.loc[:,'콜레라':]

In [137]: **data2**

Out[137]:

월별	콜레라	장티푸스	이질	대장균	A형간염
1월	6.0	175.0	550.0	7.0	351.0
2월	5.0	165.0	253.0	9.0	535.0
3월	1.0	200.0	NaN	13.0	1003.0
4월	1.0	200.0	507.0	NaN	856.0
5월	1.0	194.0	343.0	27.0	959.0
6월	3.0	227.0	272.0	92.0	928.0
7월	14.0	179.0	224.0	201.0	630.0
8월	NaN	163.0	347.0	114.0	505.0
9월	3.0	NaN	105.0	43.0	364.0
10월	1.0	107.0	142.0	39.0	230.0
11월	3.0	97.0	377.0	27.0	190.0
12월	4.0	121.0	679.0	21.0	NaN

1. 각 질병별 발생률 (월별)

In [142]: **data2.div(data2.sum(axis=0),axis=1)*100** # 내 방법

강사님

#dataframe에서의 메서드는 NA값 알아서 제외해줌

#dataframe ~ series의 계산 : series의 key와 dataframe의 key(칼럼)끼리 계산

data2/data2.sum() 으로 표현 가능!

Out[142]:

월별	콜레라	장티푸스	이질	대장균	A형간염
1월	14.285714	9.573304	14.477494	1.180438	5.357961
2월	11.904762	9.026258	6.659647	1.517707	8.166692
3월	2.380952	10.940919	NaN	2.192243	15.310640
4월	2.380952	10.940919	13.345617	NaN	13.066707
5월	2.380952	10.612691	9.028692	4.553120	14.638986
6월	7.142857	12.417943	7.159779	15.514334	14.165776
7월	33.333333	9.792123	5.896288	33.895447	9.616852
8월	NaN	8.916849	9.133983	19.224283	7.708747
9월	7.142857	NaN	2.763885	7.251265	5.556404
10월	2.380952	5.853392	3.737826	6.576728	3.510914
11월	7.142857	5.306346	9.923664	4.553120	2.900321
12월	9.523810	6.619256	17.873125	3.541315	NaN

 # cf. 월별은? -- 서로 키 매칭 X -----> div로 축 지정

In [11]: **data2.div(data2.sum(axis=1),axis=0)**

Out[11]:

월별	콜레라	장티푸스	이질	대장균	A형간염
1월	0.005510	0.160698	0.505051	0.006428	0.322314
2월	0.005171	0.170631	0.261634	0.009307	0.553257
3월	0.000822	0.164339	NaN	0.010682	0.824158
4월	0.000639	0.127877	0.324169	NaN	0.547315
5월	0.000656	0.127297	0.225066	0.017717	0.629265
6월	0.001971	0.149146	0.178712	0.060447	0.609724
7월	0.011218	0.143429	0.179487	0.161058	0.504808
8월	NaN	0.144376	0.307352	0.100974	0.447298
9월	0.005825	NaN	0.203883	0.083495	0.706796
10월	0.001927	0.206166	0.273603	0.075145	0.443160
11월	0.004323	0.139769	0.543228	0.038905	0.273775
12월	0.004848	0.146667	0.823030	0.025455	NaN

 # 키가 같지 않을 때, 사칙연산 method 말고 다른 함수를 쓰고 싶으면?

--> 축을 지정할 수 있는 메서드가 없는 경우

->무조건 이 방법 :

data2를 전치시킴! index가 key가 되도록 + 마지막 결과도 전치(for 원본형식)

In [18]: **(data2.T/data2.sum(axis=1)).T**

Out[18]:

	콜레라	장티푸스	이질	대장균	A형간염
월별					
1월	0.005510	0.160698	0.505051	0.006428	0.322314
2월	0.005171	0.170631	0.261634	0.009307	0.553257
3월	0.000822	0.164339	NaN	0.010682	0.824158
4월	0.000639	0.127877	0.324169	NaN	0.547315
5월	0.000656	0.127297	0.225066	0.017717	0.629265
6월	0.001971	0.149146	0.178712	0.060447	0.609724
7월	0.011218	0.143429	0.179487	0.161058	0.504808
8월	NaN	0.144376	0.307352	0.100974	0.447298
9월	0.005825	NaN	0.203883	0.083495	0.706796
10월	0.001927	0.206166	0.273603	0.075145	0.443160
11월	0.004323	0.139769	0.543228	0.038905	0.273775
12월	0.004848	0.146667	0.823030	0.025455	NaN

2. 대장균이 가장 많이 발병한 달을 출력

In [144]: **data2.idxmax()['대장균']**

Out[144]: '7월'

cf. 월별 가장 많이 발병한 병은?

In [22]: **data2.idxmax(axis=1)**

Out[22]:

월별

1월 이질

2월 A형간염

3월 A형간염

4월 A형간염

5월 A형간염

6월 A형간염

7월 A형간염

8월 A형간염

9월 A형간염

10월 A형간염

11월 이질

12월 이질

dtype: object

3. 데이터를 A형간염이 많은 순으로 정렬

In [145]: **data2.sort_values(by='A형간염', ascending=False)**

#data 전체정렬 -> sort_values (order by 역할) / 각 컬럼별 지정 순서는 불가능..?

Out[145]:

월별	콜레라	장티푸스	이질	대장균	A형간염
3월	1.0	200.0	NaN	13.0	1003.0
5월	1.0	194.0	343.0	27.0	959.0
6월	3.0	227.0	272.0	92.0	928.0
4월	1.0	200.0	507.0	NaN	856.0
7월	14.0	179.0	224.0	201.0	630.0
2월	5.0	165.0	253.0	9.0	535.0
8월	NaN	163.0	347.0	114.0	505.0
9월	3.0	NaN	105.0	43.0	364.0
1월	6.0	175.0	550.0	7.0	351.0
10월	1.0	107.0	142.0	39.0	230.0
11월	3.0	97.0	377.0	27.0	190.0
12월	4.0	121.0	679.0	21.0	NaN

4. 콜레라 기준으로 순위 출력

In [146]: **data2['콜레라'].rank()**

Out[146]:

월별

1월 10.0
2월 9.0
3월 2.5
4월 2.5
5월 2.5
6월 6.0
7월 11.0
8월 NaN
9월 6.0
10월 2.5
11월 6.0
12월 8.0

Name: 콜레라, dtype: float64

 # rank의 -> 전체 데이터에서 특정 column별 불가, 따로 column을 빼야 함.
=> .sort_values 쓰면 다시 정렬됨

5. 각 질병 별 발병횟수의 총 합을 출력

In [147]: **data2.sum(axis=0)**

Out[147]:

콜레라 42.0
장티푸스 1828.0
이질 3799.0
대장균 593.0
A형간염 6551.0

dtype: float64

6. 널 값이 존재하는 행 삭제 -- 못풀었따..

 # isna() --> 각 원소별 전달

cf. isnull()과 다름! null은 값이 안 들어온 것 @0@

In [34]: **data2.isna()**

Out[34]:

월별	콜레라	장티푸스	이질	대장균	A형간염
1월	False	False	False	False	False
2월	False	False	False	False	False
3월	False	False	True	False	False
4월	False	False	False	True	False
5월	False	False	False	False	False
6월	False	False	False	False	False
7월	False	False	False	False	False
8월	True	False	False	False	False
9월	False	True	False	False	False
10월	False	False	False	False	False
11월	False	False	False	False	False
12월	False	False	False	False	True

In [36]: **data2.isna().sum(axis=1)** **# 간단히!! row별sum -> NA 없는 경우 0**

Out[36]:

월별	
1월	0
2월	0
3월	1
4월	1
5월	0
6월	0
7월	0
8월	1
9월	1
10월	0
11월	0
12월	1

In [38]: **data2[data2.isna().sum(axis=1)==0]** **# 불리언 배열 사용해서 색인**

Out[38]:

월별	콜레라	장티푸스	이질	대장균	A형간염
1월	6.0	175.0	550.0	7.0	351.0
2월	5.0	165.0	253.0	9.0	535.0
5월	1.0	194.0	343.0	27.0	959.0
6월	3.0	227.0	272.0	92.0	928.0
7월	14.0	179.0	224.0	201.0	630.0
10월	1.0	107.0	142.0	39.0	230.0
11월	3.0	97.0	377.0	27.0	190.0

in DataFrame

이름색인 : 열

슬라이스 색인 : 행

불리언색인 : 행

0626

5.3.1. 상관관계와 공분산

- **.corr** : 상관관계 (cf. in R -- cor) --> DataFrame에 적용하면 다중공선성 체크
- **.cov** : 공분산
- 데이터프레임 + Series / 데이터프레임 + 데이터프레임인 경우
  .corrwith / .covwith 로 with 붙여서 사용!

In [113]: **returns=DataFrame(np.random.randn(5,4), columns=['AAPL','GOOG','IBM','MSFT'])**
In [116]: **returns**
Out[116]:

	AAPL	GOOG	IBM	MSFT
0	1.275236	0.039428	-0.095867	0.624548
1	0.789898	-0.212432	-0.447349	-1.752007
2	1.449682	-1.393918	-1.995837	0.026213
3	-0.395559	-0.770596	0.955790	-1.354834
4	0.414268	0.090413	0.497212	-0.517155

 # 두 Series끼리 계산
In [114]: **returns.MSFT.corr(returns.IBM)**
Out[114]: -0.34633294379769514

In [115]: **returns.MSFT.cov(returns.IBM)**
Out[115]: -0.3815799575773248

 # DataFrame에 적용 -- 각 col마다 (가장 많이 사용 for 다중공선성 체크)
In [117]: **returns.corr()**
Out[117]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	-0.111913	-0.829863	0.650907
GOOG	-0.111913	1.000000	0.595266	0.008696
IBM	-0.829863	0.595266	1.000000	-0.346333
MSFT	0.650907	0.008696	-0.346333	1.000000

In [118]: **returns.cov()**
Out[118]:

	AAPL	GOOG	IBM	MSFT
AAPL	0.545668	-0.051986	-0.693379	0.468354
GOOG	-0.051986	0.395446	0.423404	0.005327
IBM	-0.693379	0.423404	1.279382	-0.381580
MSFT	0.468354	0.005327	-0.381580	0.948819

In [119]: **returns.corrwith(returns.IBM)**

 # Series와 DataFrame, DF와 DF 적용시 메서드에 -with 붙이기

 # 같은 key끼리 계산됨.

Out[119]:

AAPL -0.829863

GOOG 0.595266

IBM 1.000000

MSFT -0.346333

dtype: float64

5.3.2 유일 값, 값 세기, 멤버십

.unique() 메서드

In [137]: **obj=Series(['c','a','d','a','a','b','b','c','c'])**

In [138]: **uniques=obj.unique()**

In [139]: **uniques**

Out[139]: array(['c', 'a', 'd', 'b'], dtype=object)

In [140]: **uniques.sort()** **# 정렬된 순서를 반환하고 싶을 때!**

In [141]: **uniques** **# 정렬 확인**

Out[141]: array(['a', 'b', 'c', 'd'], dtype=object)

.value_counts()메서드

In [142]: **obj.value_counts()** **# 도수 값의 내림차 순으로 정렬**

Out[142]:

c 3

a 3

b 2

d 1

dtype: int64

In [143]: **pd.value_counts(obj.values,sort=False)** **# 색인 값 순으로 정렬옵션 sort=False**

Out[143]:

a 3

b 2

c 3

d 1

dtype: int64

.isin 메서드

- 어떤 값이 Series에 있는지 나타내는 불리언 벡터를 반환
- Series나 D.F 칼럼에서 값을 골라내고 싶을 때(재색인) 유용하게 사용할 수 있다.
- 리스트의 .find 메서드와 다르게 원소 값과 동일한 값을 넣어야 함 !

```
In [145]: mask=obj.isin(['b','c'])    # 여러 개의 값 -> 리스트 형태
In [146]: mask
Out[146]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
In [147]: obj[mask]
Out[147]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

정리

메서드	설명
isin	**Series**의 각 원소가 넘겨 받은 연속된 값에 속하는지 나타내는 불리언 배열 반환
unique	**Series**에서 중복되는 값을 제거하고 유일한 값만 포함하는 배열을 반환한다. 결과는 발견된 순서대로 반환한다.
value_counts	**Series**에서 유일 값에 대한 색인 값과 도수를 계산한다. 결과는 도수 값의 내림차순으로 정렬된다.

 # DataFrame에서 여러 로우에 대해 히스토그램을 구해야 하는 경우

In [152]: **data=DataFrame({'Qu1':[1,3,4,3,4],**
...: **'Qu2':[2,3,1,2,3],**
...: **'Qu3':[1,5,2,4,4,]})**

In [153]: **data**

Out[153]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

In [154]: **result=data.apply(pd.value_counts).fillna(0)**

 # apply를 사용해 pd.value_counts 적용 (pd.value_counts 함수는 Series에 적용되는 함수)

In [155]: **result**

Out[155]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

pd.value_counts를 데이터 프레임에 적용할 때 단점

-칼럼끼리 범주가 다르면 범위가 넓어짐.

-데이터 프레임 전체를 대상으로는 잘 쓰지 않는다.

5.4 누락된 데이터 처리하기

cf. run profile에 np.nan을 NA로 Alias 지정했음. np.nan 대신 NA로 사용하면 됨!
+ 파이썬의 내장 None값도 NA 값으로 취급됨.

NA처리 메서드

인자	설명
dropna	누락된 데이터가 있는 축(로우, 칼럼)을 제외시킨다. 어느 정도의 누락 데이터까지 용인할 것인지 지정할 수 있다.
fillna	누락된 데이터를 대신할 값을 채우거나 'ffill' 또는 'bfill' 같은 보간 메서드를 적용한다.
isnull	누락되거나 NA인 값을 알려주는 불리언 값이 저장된, 같은 형의 객체를 반환한다.
notnull	isnull과 반대되는 메서드다.

5.4.1. 누락된 데이터 골라내기

.dropna() 메서드

- 기본적으로 NA가 하나라도 있는 로우는 제외시킴
- 로우 제외시키는 방법 -- 주로 시계열 데이터에 사용되는 경향이 있음.
- **how='all'** 옵션 -> 모든 값이 NA인 로우만 제외시킨다.
- **axis=0** 옵션(기본값) -> 행 / 열 축 설정 (axis=1은 아주 드문 경우)
- **thresh 옵션** : 몇 개 이상의 값이 들어있는 로우만 살펴볼 때

ex. thresh=3 : NA가 아닌 값이 3개 이상인 로우

```
#in series
In [9]: data=Series([1,NA,3.5,NA,7])
In [10]: data.dropna()
Out[10]:
0    1.0
2    3.5
4    7.0
dtype: float64
In [11]: data[data.notnull()]    # 불리언 색인을 이용해서 직접 계산하는 것도 가능
Out[11]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

```
# in DataFrame
In [12]: data=DataFrame([[1,6.5,3],[1,NA,NA],[NA,NA,NA],[NA,6.5,3]])
In [13]: cleaned = data.dropna()
In [14]: data.dropna()          #NA가 하나라도 있는 로우 삭제
Out[14]:
     0    1    2
0  1.0  6.5  3.0
In [15]: data.dropna(how='all')  # 모두 NA인 로우 삭제
Out[15]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
In [18]: data[4]=NA
In [19]: data
Out[19]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN
In [20]: data.dropna(axis=1,how='all')    # 모두 NA인 column 삭제
Out[20]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

In [21]: **df=DataFrame(np.random.randn(7,4))**

In [22]: **df.iloc[:4,1]=NA; df.iloc[:2,2]=NA; df.iloc[:5,0]=NA**

In [25]: **df**

Out[25]:

	0	1	2	3
0	NaN	NaN	NaN	0.284568
1	NaN	NaN	NaN	-0.930336
2	NaN	NaN	1.053583	-0.353948
3	NaN	NaN	0.404803	0.556907
4	NaN	-1.728137	-1.264588	0.980143
5	0.546879	-0.957132	0.112953	-0.119958
6	0.374052	-0.076945	0.341012	-0.812246

In [26]: **df.dropna(thresh=3)**

 # thresh 옵션 -- NA값이 아닌 값이 thresh 인자 이상 있는 로우 반환

Out[26]:

	0	1	2	3
4	NaN	-1.728137	-1.264588	0.980143
5	0.546879	-0.957132	0.112953	-0.119958
6	0.374052	-0.076945	0.341012	-0.812246

In [27]: **df.dropna(thresh=5, axis=1)** **# thresh옵션 -> 열별 적용 가능**

Out[27]:

	2	3
0	NaN	0.284568
1	NaN	-0.930336
2	1.053583	-0.353948
3	0.404803	0.556907
4	-1.264588	0.980143
5	0.112953	-0.119958
6	0.341012	-0.812246

5. 4. 2. 누락된 값 채우기

.filna() 메서드

- fillna 메서드에 채워 넣고 싶은 값을 넘겨주면 된다.
- fillna의 괄호 안에 사전(딕셔너리) 값을 넘겨서 각 칼럼마다 다른 값을 채워넣을 수 있다. ({칼럼 값 : NA일 때 채워넣을 값} 형태로 작성하기)
- 뷰가 아님, 기존 객체와 별개로 생성됨 / 단순히 print 결과만 보여줌, 원본 데이터 수정 X
- **inplace=True** 옵션을 통해 기존 객체를 변경할 수도 있다. for 메모리 낭비를 피하기 위해
  -->fillna의 기본값은 새로운 객체를 반환
- **method='ffill' / 'bfill'** --> reindex에서 사용했던 보간 메서드도 사용 가능
- limit 옵션 : 값을 앞 혹은 뒤에서부터 몇 개까지 채울지를 지정한다. (method랑 같이!)

```
In [323]: df=DataFrame(np.random.randn(7,3))
In [324]: df.iloc[:4,1]=NA; df.iloc[:2,2]=NA

In [325]: df.fillna(0)
Out[325]:
          0         1         2
0  1.965466  0.000000  0.000000
1 -0.681540  0.000000  0.000000
2 -0.329802  0.000000  0.018523
3  0.488013  0.000000 -0.476198
4 -0.245283  0.178982  1.345778
5  0.461850  1.088133  1.084988
6 -1.739458  0.251188  0.995239

In [326]: df.fillna({1:0.5, 3:-1})      #fillna에 사전 값을 넘겨서 각 칼럼마다 다른 값 채워넣기
                                        {컬럼이름 : 치환값}
Out[326]:
          0         1         2
0  1.965466  0.500000       NaN
1 -0.681540  0.500000       NaN
2 -0.329802  0.500000  0.018523
3  0.488013  0.500000 -0.476198
4 -0.245283  0.178982  1.345778
5  0.461850  1.088133  1.084988
6 -1.739458  0.251188  0.995239
```

In [327]: **_ = df.fillna(0,inplace=True)**

In [328]: **df**

Out[328]:

	0	1	2
0	1.965466	0.000000	0.000000
1	-0.681540	0.000000	0.000000
2	-0.329802	0.000000	0.018523
3	0.488013	0.000000	-0.476198
4	-0.245283	0.178982	1.345778
5	0.461850	1.088133	1.084988
6	-1.739458	0.251188	0.995239

 # _ : 이전변수라는 의미. 이미 옵션자체에서 이전 값에 대입했는데, _는 필요 없음. _사용시 따로 저장되지 않음 (굉장히 큰 객체에 대해 저장공간을 효율적으로 사용할 때 사용)

 # 만일 inplace=True 옵션 사용하지 않을 것이면 df=df.fillna~~~를 써도 됨

In [329]: **data=Series([1, NA, 3.5, NA, 7])**

In [330]: **data.fillna(data.mean())** ***# 각 열의 평균 값이나 중간 값도 전달 가능***

Out[330]:

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

5. 5 계층적 색인 - 멀티인덱스

- index나 columns 2차원 리스트로 전달
-[[1st레벨(상위레벨)], [2nd레벨]] 순으로 작성
-MultiIndex는 색인에 하위 색인이 존재하는 형태로 계층적 접근이 가능하다.
levels : 각 index range, 실제 표현되는 index 값 (각 층)
labels : 각 index number, 각각의 순서

##in Series

```
In [331]: data=Series(np.random.randn(10),
     ...:                 index=[['a','a','a','b','b','b','c','c','d','d'],
     ...:                 [1,2,3,1,2,3,1,2,2,23]])
In [332]: data
Out[332]:
a   1     -2.509886
    2      0.889261
    3      0.478503
b   1      0.301410
    2     -0.162482
    3     -0.939533
c   1      1.635976
    2     -0.431326
d   2      0.307141
    23     0.807117
dtype: float64
In [333]: data.index
Out[333]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3, 23]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 3]])
In [334]: data['b']
Out[334]:
1     0.301410
2    -0.162482
3    -0.939533
dtype: float64
In [335]: data['b':'c']
Out[335]:
b   1     0.301410
    2    -0.162482
    3    -0.939533
```

```
c   1       1.635976
    2      -0.431326
dtype: float64
```

In [336]: **data.loc[['b','d']]**

```
Out[336]:
b   1        0.301410
    2       -0.162482
    3       -0.939533
d   2        0.307141
    23       0.807117
dtype: float64
```

In [337]: **data[:,2]**

 # 하위 계층의 객체를 선택

 # 최상위레벨 : 전체, 하위레벨에서 특정 값 : 2 선택 (헷갈려서 잘 쓰이는 표현 아님)

--> .xs()메서드 사용

```
Out[337]:
a       0.889261
b      -0.162482
c      -0.431326
d       0.307141
dtype: float64
```

 # 멀티인덱스 일 땐 최상위레벨을 인덱싱

 # 상위인덱스 이름 색인, 슬라이스 색인, ix(loc/iloc) 사용 가능

--상위인덱스로서 동시 여러 개 나열 가능

데이터 프레임의 멀티인덱스 데이터프레임 -- 두 축 모두 계층적 색인을 가질 수 있음

```
In [31]: frame=DataFrame(np.arange(12).reshape((4,3)), index=[['a','a','b','b'], [1,2,1,2,]],
   ...:                  columns=[['Ohio','Ohio','Colorado'],['Green','Red','Green']])
   ...:
In [32]: frame
Out[32]:
```

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★

.xs() 메서드 (교재에 없음) **# default는 행범위**

- 레벨에 따른 선택이 가능한 멀티인덱스용 색인 인덱스
- 행, 열 둘 다 선택 가능 **(default : 행 선택)** --> axis=1 축값 설정해서 열 선택
- 상위 계층 level=0, 하위 계층level = 1
- level의 이름이 있으면 level 옵션에 이름 사용 가능
- **for 차원축소** **(만일 차원 유지하려면 .loc[(행,열)] 형태로 !**

In [34]: **frame.xs('a',level=0)**

Out[34]:

	Ohio		Colorado
	Green	Red	Green
1	0	1	2
2	3	4	5

In [36]: frame.xs(2,level=1) **# 모든 상위계층을 대상으로 하위계층이 2인 행을 나타내기**

Out[36]:

	Ohio		Colorado
	Green	Red	Green
a	3	4	5
b	9	10	11

In [44]: **frame.xs('Red', level=1,axis=1)** **# 열에 대한 색인시 axis=1 꼭 사용해줘야 함**

Out[44]:

		Ohio
a	1	1
	2	4
b	1	7
	2	10

cf. frame['Red'] ->오류 // 하위계층의 값을 키값으로 색인 불가능

+ frame[:,'Red']도 오류!!! 어려웁다!! --> .xs메서드 사용

frame['Ohio'] 는 가능!! 컬럼의 상위레벨 키값 색인은 가능

a(최상위 레벨 중) 특정 하위 레벨을 선택하고 싶을 때?

----->전달방식 : 튜플 형태로 순서대로 작성!

cf. **.xs 메서드 -> 하나의 키 값만 가능. 상위레벨 + 하위레벨 동시에 나열 불가능**

In [48]: **frame.loc[('a',2)]** **# 멀티인덱스 행 색인 ~ .loc 사용**

 # 멀티인덱스에 대한 셋트 전달.

 # 첫 번째 레벨이 a이면서, 다음 레벨이 2인 행을 찾아주세요

Out[48]:

Ohio	Green	3
	Red	4
Colorado	Green	5

Name: (a, 2), dtype: int32

In [50]**: frame[('Ohio','Green')]** **# 멀티인덱스 열 색인**

 # 상위레벨이 Ohio이면서 하위레벨이 Green인 값을 찾아주세요

 # 튜플 형태로 차례대로 전달 유의!

Out[50]:

a	1	0
	2	3
b	1	6
	2	9

Name: (Ohio, Green), dtype: int32

cf. 멀티인덱스 설정

data.set_index([[상위],[하위]])

data.index=[[상위],[하위]]

data.columns=[[상위],[하위]]

cf. .reset_index() : 인덱스->col화

cf. dataframe.iloc[0] ➔ (0,0) 튜플 전달 → 왜냐면 순서 중 첫번째니까!

.stack / unstack 메서드 for 멀티인덱스

- unstack 메서드 : 계층적 색인을 피벗 테이블 형태로 생성
계층을 가지고 있는 데이터가 계층을 가지지 않는 평평한 인덱스로-> 교차테이블 형태로 표현됨
(하위레벨이 컬럼화)
- stack 메서드 : 테이블 형태의 색인을 계층적 색인으로 변경(NA값은 제외됨)
(컬럼이 하위레벨로 들어감)

5.5.1 계층 순서 바꾸고 정렬하기

(cf.swapaxis, transpose ~ .T 메서드 in Numpy -- 행과 열 전치)

.swaplevel() 메서드

- 넘겨받은 두 개의 계층 번호나 이름이 뒤바뀐 새로운 객체를 반환한다.
- 데이터는 변경되지 않는다.
- 원본의 데이터는 수정되지 않고 재정렬되어서 출력된다.
- 이름이 없으면 레벨 번호로! (0, 1)
- 나중에 stack/unstack시 칼럼이 되는 하위레벨 미리 .swaplevel()로 수정해놓기에 의의.

((cf.멀티인덱스도 상위-하위 순서로 이름 지정 가능

In [109]: **frame.index.names=['key1','key2']**

In [110]: **frame.columns.names=['state','color']**

In [111]: **frame**

Out[111]:

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11**))**

swaplevel 메서드 예시

In [113]: **frame.swaplevel('key1','key2')**

Out[113]:

state		Ohio		Colorado
color		Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

In [115]: frame.sort_index(level=1)

 # sort_index 메서드에 level 지정해서 정렬할 레벨 설정(기본-axis=0)

Out[115]:

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

In [116]: **frame.swaplevel(0,1).sort_index(level=0)**

Out[116]:

state		Ohio		Colorado
color		Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

5.5.2 단계별 요약통계

In [120]: **frame.sum(level='key2')** **# 산술연산 method에서 level 옵션 사용**

Out[120]:

state	Ohio		Colorado
color	Green	Red	Green
key2			
1	6	8	10
2	12	14	16

In [121]: **frame.sum(level='color', axis=1)**

Out[121]:

color		Green	Red
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

실습문제

1. emp 데이터 로딩 후

In [175]: **emp=pd.read_csv("emp.csv")**

1) deptno별 sal의 합계를 구하여라

선생님

먼저 그냥 해보기!

In [34]: emp[emp['DEPTNO']==10].SAL.sum()

Out[34]: 8750

====이를 활용====

f1= lambda x : emp[emp['DEPTNO']==x].SAL.sum()

 # map을 통해 x의 원소가 한 개씩 전달됨 ! x의 각 원소(DEPTNO)에 따른 sum을 구한 값이 출력됨!

In [25]: **emp['DEPT_SUM']=emp['DEPTNO'].map(f1)**

In [26]: **emp**

Out[26]:

	EMPNO	ENAME	JOB	SAL	COMM	DEPTNO	DEPT_SUM
0	7369	SMITH	CLERK	800	NaN	20	10875
1	7499	ALLEN	SALESMAN	1600	300.0	30	9400
2	7521	WARD	SALESMAN	1250	500.0	30	9400
3	7566	JONES	MANAGER	2975	NaN	20	10875
4	7654	MARTIN	SALESMAN	1250	1400.0	30	9400
5	7698	BLAKE	MANAGER	2850	NaN	30	9400
6	7782	CLARK	MANAGER	2450	NaN	10	8750
7	7788	SCOTT	ANALYST	3000	NaN	20	10875
8	7839	KING	PRESIDENT	5000	NaN	10	8750
9	7844	TURNER	SALESMAN	1500	0.0	30	9400
10	7876	ADAMS	CLERK	1100	NaN	20	10875
11	7900	JAMES	CLERK	950	NaN	30	9400
12	7902	FORD	ANALYST	3000	NaN	20	10875
13	7934	MILLER	CLERK	1300	NaN	10	8750

cf.

[10,20,30] 에 대해서만 알고 싶으면?

map 메서드 자체에 unique ~ 각각에 관한 값이 출력될 것..

array는 .map 메서드에는 불가능 (Series로 바꿔주거나 map 함수 사용)

Series(emp['DEPTNO'].unique()).map(f1)

****
그렇지만 DEPTNO 값이 안 나온당 그래서

In [80]: **Series(emp['DEPTNO'].unique(),index=emp['DEPTNO'].unique()).map(f1)**
Out[80]:
20   10875
30   9400
10   8750
dtype: int64

DEPTNO 순서대로 정렬하려면
In [81]: **Series(emp['DEPTNO'].unique(),index=emp['DEPTNO'].unique()).map(f1).sort_index()**
Out[81]:
10   8750
20   10875
30   9400
dtype: int64

내가 푼 방법 -- 멀티인덱스로 구하기
In [316]: **emp2=emp['SAL']**
In [317]: **emp2.index=emp.DEPTNO**
In [319]: **emp2.sum(level=0)**
Out[319]:
DEPTNO
20   10875
30   9400
10   8750
Name: SAL, dtype: int64
In [321]: **emp2.sum(level=0).sort_values()**
Out[321]:
DEPTNO
10   8750
30   9400
20   10875
Name: SAL, dtype: int64

2) job이 analyst인 사원들의 평균연봉을 구하여라.

In [179]: **emp[emp.JOB=='ANALYST'].SAL.mean()**

Out[179]: 3000.0

강사님

emp[emp['JOB']=='ANALYST'].SAL.mean()

2. 병원현황 데이터 로딩 후

df=pd.read_csv("병원현황.csv",engine='python')

In [159]: **df=pd.read_csv("병원현황.csv",engine='python')**

 # 오류 발생시 engine='python' 옵션 쓰기

In [161]: **df.index=df.표시과목**

In [163]: **df2=df.drop(columns='표시과목')**

 # R에서의 -를 대체할 수 있는 .drop 메서드, 여러 개를 제외할 경우 list 형태로 작성

1) 각 병원(진료과목별)이 가장 많은 지역 출력

In [42]: **df2.idxmax(axis=1)**

Out[42]:

표시과목

내과 강남구

외과 강남구

정형외과 강남구

성형외과 강남구

산부인과 강남구

소아청소년과 강남구

안과 강남구

이비인후과 강남구

피부과 강남구

dtype: object

2) 각 진료과목별 병원수의 평균값과 총합을 동시 출력(요약통계)

강사님 lambda 사용

In [43]: **f2=lambda x : Series([x.mean(), x.sum()], index=['avg','sum'])**

#데이터프레임에 apply적용, Series로 리스트를 묶으면 dataframe 형태로 출력됨

apply 적용할 때 축값 지정 가능 --> f2 만들 때 작성 X

In [44]: **df2.apply(f2, axis=1)**

Out[44]:

표시과목	avg	sum
내과	44.0	440.0
외과	8.9	89.0
정형외과	19.8	198.0
성형외과	34.8	348.0
산부인과	18.0	180.0
소아청소년과	21.7	217.0
안과	19.3	193.0
이비인후과	25.5	255.0
피부과	22.3	223.0

#데이터프레임에 apply적용, Series로 리스트를 묶으면 dataframe 형태로 출력됨

과정 생각해보기 !

데이터프레임에 apply로 / Series를 리스트로 묶은 함수를 적용 /

.apply 메서드에서 axis 방향대로 Series를 계속 반환함 --> 여러 Series가 합쳐짐--> DataFrame!

위의 예제에서는 axis=1로 했으므로 원 데이터의 index 이름 살림 + f2의 index는 col로 감

만일 axis=0으로 하면 원 데이터의 column 살림 + f2의 index는 index로 !

내가 푼 방법 (딕셔너리를 데이터프레임화!)

In [174]: **DataFrame({'mean':df2.mean(axis=1),'sum':df2.sum(axis=1)})**

Out[174]:

표시과목	mean	sum
내과	44.0	440
외과	8.9	89
정형외과	19.8	198
성형외과	34.8	348
산부인과	18.0	180

...

+(실습추가2019) set_index (+ keys 옵션) 를 사용해 열 색인을 행으로 옮기기
reset_index (+level 옵션)를 사용해 행 색인을 열로 옮기기

In [81]:
```
np.random.seed(123789)
data=np.random.binomial(n=10,p=0.6,size=30).reshape(10,3)
df=DataFrame(data,columns=["V1","V2","V3"])
df["V4"]=["M","F","M","M","M","F","M","F","F","M"]
print("DataFrame₩n{0}".format(df))
```

```
DataFrame
   V1  V2  V3 V4
0   8   8   6  M
1   2   7   7  F
2   6   3   8  M
3   7   4   4  M
4   5   5   7  M
5   5   6   8  F
6   5   6   6  M
7   4   4   6  F
8   3   6   6  F
9   6   8   4  M
```

In [82]:
```
df1=df.set_index(keys=["V2","V4"]).sort_index()
print("V2, V4 열 색인->행 색인₩n{0}".format(df1))
```

```
V2, V4 열 색인->행 색인
       V1  V3
V2 V4
3  M    6   8
4  F    4   6
   M    7   4
5  M    5   7
6  F    5   8
   F    3   6
   M    5   6
7  F    2   7
8  M    8   6
   M    6   4
```

In [83]:
```
print("V4 열 색인->행 색인₩n{0}".format(df1.reset_index(level="V4")))
```

```
V4 열 색인->행 색인
   V4  V1  V3
V2
3   M   6   8
4   F   4   6
4   M   7   4
5   M   5   7
6   F   5   8
6   F   3   6
6   M   5   6
7   F   2   7
8   M   8   6
8   M   6   4
```