



# 데이터 처리/분석-1

엄진영

## • 데이터 분석

- 현재 상황으로부터 이상적인 모습에 가장 빨리 도달하기 위해 문제를 추출하는 것에 주안점을 두고 문제 해결을 실시해 나가는 것



- 어떤 상품의 매상이 떨어지고 있다

- 회사에서 주력상품은 아니고 조만간 판매가 중단될 상품 → 문제 X
- 회사의 수익에 큰 영향을 미치는 상품 → 문제 O

- 어떤 상품의 매상이 오르고 있다

- 실제로 그 상품에 들고 있는 광고비에 맞지 않은 매상 상황 → 문제 O

- 이상적인 모습과 현실의 모습 사이에 차이가 있어야 문제

## • 현상과 문제 구분하기

현상	전제	이상적인 모습	문제인가?
매상이 떨어졌다	매상 비율이 낮다.	지금 상태로 OK	문자 X
	매상 비율이 높다.	호조기였을 때 매상	문제
매상이 올랐다	광고비용이 높다.	광고 비용을 낮춤	문제
	광고비용이 적절하다.	지금 상태로 OK	문자 X

## • 데이터 수집

- 문제를 검증하기 위해 어떤 데이터가 필요한가?
- 분석자가 사용할 수 있는 곳에 필요한 데이터가 보존되어 있는가?
- 분석자가 신청하면 필요한 데이터를 사용할 수 있는가?
- 필요한 데이터가 보존되어 있지 않을 경우 새로 데이터를 취득할 수 있는가?
- 필요한 데이터가 보존되어 있지 않고 또한 새로 취득하기에는 시간이나 비용이 많이 들 경우 그것을 대체할 수 있는 다른 데이터가 있는가?

## • 데이터 가공

- 데이터 결합
- 판정용 변수 작성
- 이산화 변수 작성

	의사결정지원	자동화/최적화
목적	사람의 행동결정을 지원	컴퓨터의 행동결정을 지원
목표	의사소통비용 절감	추정 정확도 향상, 계산량 삭감
주로 사용되는 기법	단순집계, 크로스집계	기계학습, 알고리즘 구축

의사결정지원에 도움이 되는 통계해석  
자동화/최적화에 도움이 되는 기계학습

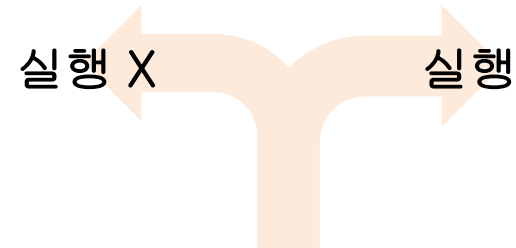


기획 담당자  
비즈니스 책임자

데이터 분석의 성과  
의사소통

데이터 분석 담당자

의사결정지원의 경우



시스템 이용자

데이터 분석의 성과  
의사소통

데이터 분석 담당자

자동화/최적화의 경우

가. 데이터 담기

나. Numpy 시작하기

다. Pandas 시작하기

라. Dask 시작하기

마. Numba 시작하기



## • 깔끔한 데이터(Tidy data)란?

- 우리는 데이터 분석을 수행하면서 다양한 데이터 변환 작업을 수행하게 된다. 이는 데이터가 원래 특정 분석을 염두에 두고 만들어지는 경우가 거의 없기 때문이며, 사실 애초 데이터 설계를 할 때 분석 목적을 알기도 불가능하다는게 가장 큰 원인이 아닐까 한다. 이런 연유로 전체 데이터 분석 작업에서 70% 혹은 80% 이상이 이런 데이터 변환 및 전처리 작업에서 소모된다.

- Wikipedia

- ✓ 밑바닥 부터 시작할 필요 없는 데이터

- Jeff Leek, “The Elements of Data Analytic Style”

- 각 변수는 개별의 열(column)으로 존재한다.
- 각 관측치는 행(row)를 구성한다.
- 각 표는 단 하나의 관측기준에 의해서 조직된 데이터를 저장한다.
- 만약 여러 개의 표가 존재한다면, 적어도 하나 이상의 열(column)이 공유되어야 한다.

	Treatment A	Treatment B
John Smith	-	2
Jane Doe	16	11
Mary Johnson	3	1

지저분한 데이터의 예

Name	Treatment	Result
John Smith	a	-
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

깔끔한 데이터(Tidy data)의 예

## • 지저분한 데이터의 일반적인 모습

- 열 이름(Column header)이 변수 이름이 아니고 값인 경우
- 같은 표에 다양한 관측 단위(observational units)가 있는 경우
- 하나의 열(column)에 여러 값이 들어 있는 경우
- 변수가 행과 열에 모두 포함되어 있는 경우
- 하나의 관측 단위(observational units)가 여러 파일로 나누어져 있는 경우



- 파이썬에서 여러 개의 자료를 한 변수에 담기

- 리스트 자료형

- ✓ 리스트는 하나의 변수에 여러 값을 할당하는 자료형
    - ✓ 여러 개의 자료를 순서에 따라 저장 가능

리스트변수 = [자료1, 자료2, 자료3]

- 딕셔너리 자료형

- ✓ 여러 개의 자료를 이름을 붙여서 저장

딕셔너리 변수 = {자료이름1: 자료값1, 자료이름2:자료값2}

- ✓ 자료의 이름을 키(key), 자료의 값은 값(value)

- `colors = ['red', 'blue', 'green']`

colors	'red'	'blue'	'green'
인덱스	0	1	2

- `colors[0:2]`

`['red', 'blue']`

- 마지막 인덱스 -1 까지만 출력

- `colors.append('white')`

- `colors`

`['red', 'blue', 'green', 'white']`

- 리스트 맨 끝 인덱스에 새로운 값 추가

- `colors.extend(['black', 'purple'])`

- `colors`

`['red', 'blue', 'green', 'white', 'black', 'purple']`

- 기존 리스트에 그대로 새로운 리스트를 합치는 기능

- `colors.insert(0, 'orange')`

- `colors`

`['orange', 'red', 'blue', 'green', 'white', 'black', 'purple']`

- 리스트의 특정 위치에 값을 추가
- 리스트의 맨 끝에 값이 추가되는 것이 아니라 지정한 위치에 값이 추가됨

- `colors.remove('red')`

- `colors`

`['orange', 'blue', 'green', 'white', 'black', 'purple']`

- 리스트에 있는 특정 값을 지우는 역할

- `colors[0] = 'red'`

- `colors`

`['red', 'blue', 'green', 'white', 'black', 'purple']`

- `del colors[0]`

- `colors`

`['blue', 'green', 'white', 'black', 'purple']`

- `student_info = {20140012:'Janhyeok', 20140059: 'Jiyong', 20150234:'JaeHong', 20140058:'Wonchul'}`
  - `student_info`라는 변수를 먼저 선언한 후 해당 변수에 {키:값} 형태로 값을 입력
  - 값에는 다양한 자료형이 들어갈 수 있다.

학번 (키)	이름 (값)
20140012	Janhyeok
20140059	Jiyong
20150234	JaeHong
20140058	Wonchul

- `student_info[20150234]`  
`'JaeHong'`

- 해당 값의 키를 대괄호 [ ] 안에 넣어 호출
- 키는 문자열로 선언할 수 있고, 정수형으로 선언할 수도 있음

학번	이름
20140012	Janhyeok
20140059	Jiyong
20150234	JaeHong
20140058	Wonchul
20140039	Jaechul

- **student\_info[20140039] = 'Jaechul'**

- 딕셔너리 자료형에서의 재할당은 키를 이용하여 해당 변수를 호출한 후, 새로운 값을 할당

- **student\_info.key()**

**dict\_keys([20140039, 20140057, 20150234, 20140058, 20140039])**

- 키만 출력하기 위해서는 keys() 함수를 사용

- **student\_info.values()**

**dict\_values(['Janhyeok', 'Jiyong', 'JaeHong', 'Wonchul', 'Jaechul'])**

- 값을 출력하기 위해서는 values() 함수를 사용

- **student\_info.items()**

**dict\_items([(20140012, 'Janhyeok'), (20140059, 'Jiyong'), (20150234, 'JaeHong'), (20140058, 'Wonchul'), (20140039, 'Jaechul')])**

- 키-값 쌍을 모두 보여주기 위해서는 items() 함수를 사용



```
In [2]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89 ]  
        fam
```

```
Out[2]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

인덱스	0	1	2	3	4	5	6	7
-----	---	---	---	---	---	---	---	---

리버스 인덱스	-8	-7	-6	-5	-4	-3	-2	-1
---------	----	----	----	----	----	----	----	----

```
In [3]: fam[3]
```

```
Out[3]: 1.68
```

```
In [4]: fam[6]
```

```
Out[4]: 'dad'
```

```
In [5]: fam[-1]
```

```
Out[5]: 1.89
```

```
In [6]: fam[3:5]
```

```
Out[6]: [1.68, 'mom']
```

➔ 마지막 값부터 -1을 할당하여  
첫 번째 값까지 역순으로 올라오는 방식

```
In [8]: x = ["a","b","c"]  
        y = x  
        y[1] = "z"  
        y
```

```
Out [8]: ['a', 'z', 'c']
```

```
In [9]: x
```

```
Out [9]: ['a', 'z', 'c']
```

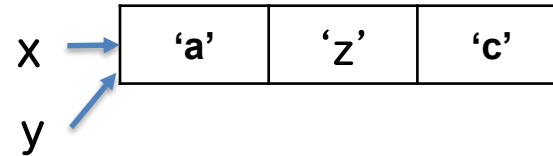
```
In [12]: x = ["a","b","c"]  
         y = x[:]  
         y[1] = "z"  
         y
```

```
Out [12]: ['a', 'z', 'c']
```

```
In [13]: x
```

```
Out [13]: ['a', 'b', 'c']
```

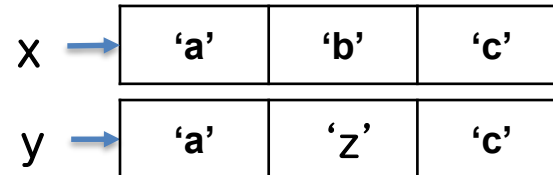
y 리스트가 x 리스트의 메모리 주소와 같이 연결되기 때문  
두 변수가 같은 메모리 주소로 연결 되어 있으므로 하나의  
변수 값만 바뀌더라도 둘 다 영향 받음



‘=’의 의미는 같다가 아닌  
메모리 주소에 해당값을 할당(연결)한다는 의미

x 변수의 처음부터 끝까지 슬라이싱해서 리스트 y에 대입

슬라이싱(slicing) : 리스트의 인덱스를 사용하여  
전체 리스트에서 일부를 잘라내어 반환



In [14]: `help(round)`

help : 도움말을 보여주는 명령어

Help on built-in function round in module builtins:

`round(number, ndigits=None)`

Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None. Otherwise the return value has the same type as the number. ndigits may be negative.

In [15]: `round(1.68, 1)`

round()

number  
ndigits

Out [15]: 1.7

## • Numpy

- 파이썬의 패키지 중 하나로 과학 계산을 위한 라이브러리
- Numerical Python 의 약자
- 다차원 배열과 행렬 연산을 처리하는데 필요한 유용한 기능을 제공
- 외부 패키지이므로 импорт 명령어를 통해 불러옴

```
In [17]: import numpy as np
```

- 만약, 버전 차이로 안될 경우 명령창에 'pip install numpy'를 실행시켜 설치 후 파이썬 코드에서 사용하기 위해서 'import numpy as np'로 패키지 추가 후 사용

# Numpy를 활용한 일차원 배열 계산

$$\bullet \quad x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \dots \\ 999 \\ 1000 \end{bmatrix}, y = \begin{bmatrix} 1001 \\ 1002 \\ 1003 \\ \dots \\ 1999 \\ 2000 \end{bmatrix}, z = x + y = \begin{bmatrix} 1 + 1001 \\ 2 + 1002 \\ 3 + 1003 \\ \dots \\ 999 + 1999 \\ 1000 + 2000 \end{bmatrix} = \begin{bmatrix} 1002 \\ 1004 \\ 1006 \\ \dots \\ 2998 \\ 3000 \end{bmatrix}$$

```
In [19]: import numpy as np
x = np.arange(1,1001)
y = np.arange(1001,2001)
z = np.zeros_like(x)
for i in range(1000):
    z[i] = x[i] + y[i]
z[:10]
```

```
Out[19]: array([1002, 1004, 1006, 1008, 1010, 1012, 1014, 1016, 1018, 1020])
```

동일한 데이터 타입(dtype)과 크기의 0만  
가진 행렬을 만들어 줌

파이썬 3에서는 기본 내장함수 range()를 활용하여  
0부터 n-1까지 1씩 증가하는 n개의 정수로 이루어지는  
수열을 만들 수 있음

```
>> number_list = list(range(6))
>> print(number_list)
[0, 1, 2, 3, 4, 5]
```

Range()함수 인자로 ('시작하는 수', '끝나는 수+1', '간격')  
으로 1이 아닌 다른 수의 간격으로 증가하는 수열을 만들

```
>> number_step_list = list(range(3,10,3))
>> print(number_step_list)
[3, 6, 9]
```

자연수 간격이 아니고, 0.1, 0.5 등 소수 형태로 간격이  
필요한 경우 arange()함수를 사용

➔ 정수 뿐만 아니라 소수 형태의 간격 지정이 가능

1.73	1.68	1.71	1.89	1.79
65.4	59.2	63.6	88.4	68.7

```
In [22]: np_2d = np.array([[1.73, 1.68, 1.71, 1.89, 1.79], [65.4, 59.2, 63.6, 88.4, 68.7]])  
np_2d
```

```
Out[22]: array([[ 1.73,  1.68,  1.71,  1.89,  1.79],  
               [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
In [23]: np_2d.shape
```

→ Numpy에서 배열은 동일한 타입의 값들을 가지며 배열의 차원을 rank라 하고, 각 차원의 크기를 튜플로 표시하는 것을 shape라 한다.  
배열 np\_2d는 행이 2이고, 열이 5인 2차원 배열로 rank는 2, shape는 (2,5)

```
Out[23]: (2, 5)
```

```
In [26]: np_2d.dtype
```

→ 만들어진 배열의 자료형을 알아봄  
Ex) float64 : 실수형 64비트(실수형의 기본형)  
Int32 : 정수형 32비트(정수형의 기본형)  
<U32 : unicode 32비트(문자형)

```
Out[26]: dtype('float64')
```

```
In [25]: np.array([[1.73, 1.68, 1.71, 1.89, 1.79], [65.4, 59.2, 63.6, 88.4, "68.7"]])
```

```
Out[25]: array([[ '1.73', '1.68', '1.71', '1.89', '1.79'],  
               ['65.4', '59.2', '63.6', '88.4', '68.7']], dtype='<U32')
```

# Numpy를 활용한 이차원 배열 계산

1.73	1.68	1.71	1.89	1.79
65.4	59.2	63.6	88.4	68.7

```
In [27]: np_2d[0]
```

```
Out[27]: array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
In [28]: np_2d[0][2]
```

```
Out[28]: 1.71
```

```
In [29]: np_2d[0,2]
```

```
Out[29]: 1.71
```

```
In [30]: np_2d[:, 1:3]
```

```
Out[30]: array([[ 1.68,  1.71],
                [59.2 , 63.6 ]])
```

```
In [31]: np_2d[1,:]
```

```
Out[31]: array([65.4, 59.2, 63.6, 88.4, 68.7])
```

```
In [1]: import numpy as np  
x = np.arange(1,10)  
x
```

```
Out[1]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [2]: x ** 2
```

```
Out[2]: array([ 1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32)
```

```
In [3]: [val ** 2 for val in range(1,10)]
```

```
Out[3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [4]: M = x.reshape((3,3))
```

→ 일단 만들어진 배열의 내부 데이터는 보존한 채로 형태만 바꾼다.  
배열 x를 3 \* 3 행렬로 변환

```
In [5]: M
```

```
Out[5]: array([[1, 2, 3],  
               [4, 5, 6],  
               [7, 8, 9]])
```



```
In [5]: M
```

```
Out [5]: array([[1, 2, 3],  
               [4, 5, 6],  
               [7, 8, 9]])
```

```
In [6]: M.T
```

→ 전치 행렬 (주어진  $m \times n$  행렬에서 행과 열을 바꾸어 만든 행렬)

```
Out [6]: array([[1, 4, 7],  
               [2, 5, 8],  
               [3, 6, 9]])
```

```
In [7]: np.dot(M, [5, 6, 7])
```

→ 행렬의 곱셈, 벡터의 내적, 벡터와 행렬의 곱을 위해 사용

```
Out [7]: array([ 38,  92, 146])
```

## - Numpy 행렬을 생성

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
array1 = np.array([[-0.048, 0.5433, -0.2349, 1.2792],
                  [-0.268, 0.5465, 0.0939, -2.0445],
                  [-0.047, -2.026, 0.7719, 0.3103],
                  [ 2.1452, 0.8799, -0.0523, 0.0672],
                  [-1.0023, -0.1698, 1.1503, 1.7289],
                  [ 0.1913, 0.4544, 0.4519, 0.5535],
                  [ 0.5994, 0.8174, -0.9297, -1.2564]])
```

- ① array1의 첫번째, 네번째 행 출력 (hint!: Bob이 첫번째 네번째에 있음)

```
array([[-0.048, 0.5433, -0.2349, 1.2792],
       [ 2.1452, 0.8799, -0.0523, 0.0672]])
```

- ② array1의 첫번째, 네번째 행 중 3,4열 출력

```
array([[-0.2349, 1.2792],
       [-0.0523, 0.0672]])
```

- ③ array1의 첫번째, 네번째 행 중 4열 출력

```
array([[1.2792],
       [0.0672]])
```

- ④ 7\*4 행렬인 array1을 4\*7행렬로 변환

```
array([[-0.048, 0.5433, -0.2349, 1.2792, -0.268, 0.5465, 0.0939],
       [-2.0445, -0.047, -2.026, 0.7719, 0.3103, 2.1452, 0.8799],
       [-0.0523, 0.0672, -1.0023, -0.1698, 1.1503, 1.7289, 0.1913],
       [ 0.4544, 0.4519, 0.5535, 0.5994, 0.8174, -0.9297, -1.2564]])
```

## • Pandas

- 데이터 분석, 데이터 처리 등을 쉽게 하기 위해 만든 패키지
- Numpy를 기반으로 만듦
- 기능
  - ✓ 축의 이름에 따라 데이터를 정렬할 수 있는 자료구조
  - ✓ 다양한 방식으로 색인(index)된 데이터를 핸들링 가능한 기능
  - ✓ 통합된 시계열(series) 기능
  - ✓ 시계열, 비시계열 데이터 모두 다룰 수 있는 자료 구조
  - ✓ 산술 연산 및 한 축(column)의 모든 값을 더하는 것과 같은 축약 연산 가능
  - ✓ 누락된 데이터의 유연한 처리 기능
  - ✓ SQL 같은 관계연산 수행기능
    - 자료구조 : 데이터의 특징을 고려하여 데이터를 저장하는 방법
    - 시계열 : 시간 경과에 따라 연속적으로 관측된 관측값의 계열을 말하는 것으로 동일한 시간간격으로 측정되는 것
- 외부 패키지이므로 импорт 명령어를 통해 불러옴

```
import pandas as pd
```

- **Series = 값(value : 데이터 자체) + 인덱스(index : 각 데이터의 의미를 표시)**
  - 1차원 배열 같은 구조
    - ✓ 인덱스 중복 가능하고, 산술 연산 가능
  - 엑셀로 보면 복수의 행으로 이루어진 하나의 열(column) 또는 복수의 열(columns)로 이루어짐

```
In [12]: from pandas import Series, DataFrame
obj = Series([4,7,-5,3])
obj
```

```
Out[12]: 0    4
         1    7
         2   -5
         3    3
dtype: int64
```

Series를 확인해보면, index와 values가 동시에 확인  
리스트의 성분 개수 = index의 개수  
Index를 생략한 경우 0부터 차례대로 정수가 할당

```
In [13]: import pandas as pd
obj2 = pd.Series([4,7,-5,3])
obj2
```

```
Out[13]: 0    4
         1    7
         2   -5
         3    3
dtype: int64
```

```
In [15]: print(obj.values) → series변수.values를 호출하면 값만 따로 확인가능
         print(obj.index) → series변수.index 를 이용하면 index 범위값만 얻을 수 있음
```

```
[ 4  7 -5  3]
```

```
RangeIndex(start=0, stop=4, step=1)
```



## • Series

- 값과 함께 원하는 index를 입력할 수 있음
- list나 array인자 이외에 추가로 index=[]를 이용해서 입력
  - ✓ 인덱스의 성분을 줄 때는 “작은따옴표로 주고, 그 외에 키값, 시리즈 이름, 시리즈 인덱스 이름은 다 “”쌍따옴표로 주는 것이 좋음

```
In [19]: obj2 = pd.Series([4,7,-5,3], index=['d','b','a','c'])  
obj2
```

```
Out[19]: d    4  
        b    7  
        a   -5  
        c    3  
        dtype: int64
```

```
In [20]: print(obj2.values)  
         print(obj2.index)  
  
[ 4  7 -5  3]  
Index(['d', 'b', 'a', 'c'], dtype='object')
```

```
In [21]: obj2['a']
```

```
Out[21]: -5
```

## • Series

```
In [19]: obj2 = pd.Series([4,7,-5,3], index=['d','b','a','c'])  
obj2
```

```
Out[19]: d    4  
         b    7  
         a   -5  
         c    3  
         dtype: int64
```

```
In [23]: obj2['d'] = 6  
obj2[['c','a','d']]
```

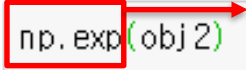
```
Out[23]: c    3  
         a   -5  
         d    6  
         dtype: int64
```

```
In [24]: obj2[obj2>0]
```

```
Out[24]: d    6  
         b    7  
         c    3  
         dtype: int64
```

```
In [25]: obj2 * 2
```

```
Out[25]: d    12  
         b    14  
         a   -10  
         c     6  
         dtype: int64
```

np.exp(obj2)  밀이 자연수 e인  
지수함수  $y = e^x$ 로  
변환

```
Out[26]: d    403.428793  
         b   1096.633158  
         a     0.006738  
         c    20.085537  
         dtype: float64
```

- Series

```
In [27]: obj2
```

```
Out[27]: d    6  
         b    7  
         a   -5  
         c    3  
         dtype: int64
```

```
In [28]: 'b' in obj2
```

```
Out[28]: True
```

```
In [29]: 'e' in obj2
```

```
Out[29]: False
```

```
from pandas import Series, DataFrame
```

```
In [30]: sdata = {'Ohio':35000, 'Texas':71000, 'Oregon':16000, 'Utah':5000}
obj3 = Series(sdata)
```

Series를 만들때 인자를 (arr, index=[,])가 아니라

```
Out[30]: Ohio      35000
Texas      71000
Oregon     16000
Utah        5000
dtype: int64
```

딕셔너리를 줘서 원하는 index를 줄 수 있음

pd.Series인자에는 list/np.array() / 파이썬의 딕셔너리가 들어갈 수 있음

```
In [31]: states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = Series(sdata, index = states)
obj4
```

```
Out[31]: California      NaN
Ohio      35000.0
Oregon     16000.0
Texas      71000.0
dtype: float64
```

NaN은 실수형에서 정의되기때문에 값의 자료형(아래의 dtype:float 64를 봐도 알 수 있고, .0이 붙은 걸 봐도 알수있음) 모두 float형으로 바뀜

```
In [32]: pd.isnull(obj4)
```

누락된 데이터가 있는지 없는지 확인할 때 사용

```
Out[32]: California      True
Ohio      False
Oregon     False
Texas      False
dtype: bool
```

```
In [33]: pd.notnull(obj4)
```

```
Out[33]: California      False
Ohio      True
Oregon     True
Texas      True
dtype: bool
```



```
In [34]: obj4.isnull()
```

```
Out[34]: California    True
Ohio                False
Oregon              False
Texas               False
dtype: bool
```

```
In [35]: obj3
```

```
Out[35]: Ohio        35000
Texas       71000
Oregon     16000
Utah        5000
dtype: int64
```

```
In [36]: obj4
```

```
Out[36]: California    NaN
Ohio        35000.0
Oregon     16000.0
Texas       71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
```

두 객체를 더하면 index가 통합되고 짝이 맞지 않는 index의 데이터는 NaN

```
Out[37]: California    NaN
Ohio        70000.0
Oregon     32000.0
Texas      142000.0
Utah         NaN
dtype: float64
```

```
In [38]: obj4.name = 'population'
obj4.index.name = 'state'
obj4
```

```
Out[38]: state
California    NaN
Ohio        35000.0
Oregon     16000.0
Texas       71000.0
Name: population, dtype: float64
```

```
In [39]: obj
```

```
Out[39]: 0    4
1    7
2   -5
3    3
dtype: int64
```

```
In [40]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
obj
```

```
Out[40]: Bob        4
Steve       7
Jeff       -5
Ryan        3
dtype: int64
```

- **Dataframe = 행과 열에 레이블을 가진 2차원 데이터**

- 스프레드시트 형식의 자료구조
- 각 열(column)마다 다른 형태를 가질 수 있음
  - ✓ 서로 다른 종류의 값(숫자, 문자열)등을 가질 수 있음
- Series가 복수개가 합쳐진 것

```
In [1]: import numpy as np
import pandas as pd
from pandas import Series, DataFrame

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year' : [2000, 2001, 2002, 2001, 2002],
        'pop'  : [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

```
In [2]: frame
```

Out [2]:

	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 [4]: DataFrame(data, columns=['year', 'state', 'pop'])
```

Out [4]:

	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

```
In [6]: frame2 = DataFrame(data, columns = ['year', 'state', 'pop', 'debt'],
                             index=['one', 'two', 'three', 'four', 'five'])
frame2
```

Out [6]:

	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 [7]: frame2.columns
```

Out [7]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

```
In [8]: frame2['state']
```

Out [8]:

one	Ohio
two	Ohio
three	Ohio
four	Nevada
five	Nevada

Name: state, dtype: object

```
In [9]: frame2.year
```

Out [9]:

one	2000
two	2001
three	2002
four	2001
five	2002

Name: year, dtype: int64

```
In [11]: frame2.loc['three']
```

Out [11]:

year	2002
state	Ohio
pop	3.6
debt	NaN

Name: three, dtype: object

```
In [12]: frame2
```

```
Out[12]:
```

	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 [13]: frame2['debt'] = 16.5
frame2
```

```
Out[13]:
```

	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 [14]: frame2['debt'] = np.arange(5.)
frame2
```

```
Out[14]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0



Out[14]:

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

```
In [15]: val = Series([-1.2, -1.5, -1.7], index = ['two', 'four', 'five'])  
frame2['debt'] = val  
frame2
```

Out[15]:

	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

```
In [16]: frame2['eastern'] = frame2.state == 'Ohio'
         frame2
```

Out[16]:

	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 [17]: del frame2['eastern']
         frame2.columns
```

Out[17]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

```
In [18]: pop = {'Nevada':{2001:2.4, 2002:2.9}, 'Ohio':{2000:1.5, 2001:1.7, 2002:3.6}}
         frame3 = DataFrame(pop)
         frame3
```

Out[18]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [19]: frame3.T
```

Out[19]:

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

```
In [20]: frame3
```

```
Out[20]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [21]: DataFrame(pop, index=[2001,2002,2003])
```

```
Out[21]:
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

```
In [25]: frame3.index.name='year'; frame3.columns.name = 'state'  
frame3
```

```
Out[25]:
```

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

- Series와 유사하게 values 속성은 DataFrame에 저장된 데이터를 2차원 배열로 반환

```
In [26]: frame2.values
```

```
Out[26]: 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)
```

```
In [27]: frame3.values
```

```
Out[27]: array([[nan, 1.5],  
               [2.4, 1.7],  
               [2.9, 3.6]])
```



# 인덱싱(indexing)

- Pandas의 인덱싱은 표 형식의 데이터에서 각 행과 열에 대한 이름과 다른 메타 데이터(축의 이름 등)을 저장하는 객체
  - Series나 DataFrame 객체를 생성할때 사용되는 배열이나 다른 순차적인 이름은 내부적으로 색인으로 변환됨

```
In [29]: obj = Series(range(3), index=['a', 'b', 'c'])
         index = obj.index
         index
```

```
Out[29]: Index(['a', 'b', 'c'], dtype='object')
```

```
In [30]: index[1:]
```

```
Out[30]: Index(['b', 'c'], dtype='object')
```

```
Index[1] = 'd' #error
```

- 색인 객체는 변경할 수 없음. 색인 객체는 변경될 수 없기에 자료 구조 사이에서 안전하게 공유될 수 있음
- 배열과 유사하게 index 객체도 고정 크기로 동작
- 각각의 색인은 담고 있는 데이터의 정보를 취급하는 여러 가지 메서드와 속성을 가지고 있음

```
In [33]: labels = pd.Index(np.arange(3))
         labels
```

```
Out[33]: Int64Index([0, 1, 2], dtype='int64')
```

```
In [35]: obj2 = pd.Series([1.5, -2.5, 0], index = labels)
         obj2
```

```
Out[35]: 0    1.5
         1   -2.5
         2    0.0
         dtype: float64
```

```
In [36]: obj2.index is labels
```

```
Out[36]: True
```

is 와 ==의 차이  
is는 reference를 비교, ==은 값을 비교

```
In [37]: frame3
```

```
Out[37]:
```

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

```
In [38]: 'Ohio' in frame3.columns
```

```
Out[38]: True
```

```
In [39]: 2003 in frame3.index
```

```
Out[39]: False
```

## • Reindexing

- 데이터를 새로운 색인에 맞게 재배열, 없는 색인 값이 있다면 비어있는 값을 새로 추가

```
In [40]: obj = Series([4.5, 7.2, -5.3, 3.6], index = ['d', 'b', 'a', 'c'])  
obj
```

```
Out[40]: d    4.5  
         b    7.2  
         a   -5.3  
         c    3.6  
         dtype: float64
```

```
In [41]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])  
obj2
```

Series 객체에 대해 reindex를 호출하면 데이터를 새로운 색인에 맞게 재배열 하고, 없는 값이 있다면 비어있는 값을 새로 추가

```
Out[41]: a   -5.3  
         b    7.2  
         c    3.6  
         d    4.5  
         e    NaN  
         dtype: float64
```

```
In [42]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value = 0)
```

```
Out[42]: a   -5.3  
         b    7.2  
         c    3.6  
         d    4.5  
         e    0.0  
         dtype: float64
```

NaN으로 채우는 대신 0으로 채워짐  
\*Series나 DataFrame을 재색인할때 fill\_value사용가능

```
In [43]: obj3 = Series(['blue', 'purple', 'yellow'], index = [0,2,4])  
obj3
```

```
Out[43]: 0      blue  
         2     purple  
         4     yellow  
dtype: object
```

```
In [45]: obj3.reindex(range(6), method = 'ffill')
```

```
Out[45]: 0      blue  
         1      blue  
         2     purple  
         3     purple  
         4     yellow  
         5     yellow  
dtype: object
```

시계열과 같은 순차적인 데이터를 재색인할 때 값을 보간하거나 채워 넣어야 하는 경우 method 옵션을 이용

- ffill : 앞의 값으로 채우기
- bfill : 뒤의 값으로 채우기

```
In [55]: obj3.reindex(range(6), method = 'bfill')
```

```
Out[55]: 0      blue  
         1     purple  
         2     purple  
         3     yellow  
         4     yellow  
         5         NaN  
dtype: object
```

- DataFrame에 대한 reindex는 row(index), column 변경 가능

```
In [47]: frame = DataFrame(np.arange(9).reshape((3,3)), index = ['a', 'c', 'd'],  
                           columns = ['Ohio', 'Texas', 'California'])  
frame
```

Out[47]:

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

```
In [49]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])  
frame2
```

Out[49]:

	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 [51]: states = ['Texas', 'Utah', 'California']  
frame.reindex(columns = states)
```

Out[51]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

```
In [57]: frame.loc[['a', 'b', 'c', 'd'], states]
```

Out[57]:

	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

- drop 메소드를 사용하면 선택한 값이 삭제된 새로운 개체를 얻어 낼 수 있음
  - 색인 배열 또는 삭제하려는 행이나 열이 제외된 리스트를 이미 가지고 있다면 행과 열을 쉽게 삭제 가능

```
In [65]: obj = Series(np.arange(5.), index=['a','b','c','d','e'])  
         new_obj = obj.drop('c')  
         new_obj
```

```
Out[65]: a    0.0  
         b    1.0  
         d    3.0  
         e    4.0  
         dtype: float64
```

```
In [66]: obj.drop(['d','c'])
```

```
Out[66]: a    0.0  
         b    1.0  
         e    4.0  
         dtype: float64
```

```
In [68]: data = DataFrame(np.arange(16).reshape((4,4)),  
                        index = ['Ohio', 'Colorado', 'Utah', 'New York'],  
                        columns = ['one', 'two', 'three', 'four'])  
data
```

axis = 0 : 행 index (default)  
axis = 1 : 열 index

Out [68]:

	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 [70]: data.drop(['Colorado', 'Ohio'])
```

Out [70]:

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

```
In [71]: data.drop('two', axis = 1)
```

Out [71]:

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

```
In [72]: data.drop(['two', 'four'], axis = 1)
```

Out [72]:

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



```
In [73]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])  
obj
```

```
Out[73]: a    0.0  
         b    1.0  
         c    2.0  
         d    3.0  
         dtype: float64
```

```
In [74]: obj['b']
```

```
Out[74]: 1.0
```

```
In [75]: obj[1]
```

```
Out[75]: 1.0
```

```
In [76]: obj[2:4]
```

```
Out[76]: c    2.0  
         d    3.0  
         dtype: float64
```

```
In [77]: obj[['b', 'a', 'd']]
```

```
Out[77]: b    1.0  
         a    0.0  
         d    3.0  
         dtype: float64
```

```
In [79]: obj[[1, 3]]
```

```
Out[79]: b    1.0  
         d    3.0  
         dtype: float64
```

```
In [82]: obj[obj < 2]
```

```
Out[82]: a    0.0  
         b    1.0  
         dtype: float64
```

```
In [83]: obj['b':'c']
```

```
Out[83]: b    1.0  
         c    2.0  
         dtype: float64
```

Index 이름으로  
슬라이싱하는 것은 시작과  
끝점을 포함한다는 점이  
일반 파이썬에서의  
슬라이싱과 다른점

```
In [84]: obj['b':'c'] = 5
```

```
In [85]: obj
```

```
Out[85]: a    0.0  
         b    5.0  
         c    5.0  
         d    3.0  
         dtype: float64
```



```
In [88]: data = DataFrame(np.arange(16).reshape((4,4)),
                        index=['Ohio', 'Colorado', 'Utah', 'New York'],
                        columns=['one', 'two', 'three', 'four'])
data
```

Out [88]:

	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 [89]: data['two']
```

Out [89]:

Ohio	1
Colorado	5
Utah	9
New York	13

Name: two, dtype: int32

```
In [90]: data[['three', 'one']]
```

Out [90]:

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

슬라이싱으로 행을 선택하거나 Boolean 배열로 열을 선택할 수 있음

```
In [91]: data[:2]
```

Out [91]:

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

```
In [92]: data[data['three'] > 5]
```

Out [92]:

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

```
In [88]: data = DataFrame(np.arange(16).reshape((4,4)),  
                          index=['Ohio', 'Colorado', 'Utah', 'New York'],  
                          columns=['one', 'two', 'three', 'four'])  
data
```

Out[88]:

	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 [94]: data < 5
```

Out[94]:

	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 [95]: data[data < 5] = 0
```

```
In [96]: data
```

Out[96]:

	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

Out [88]:

	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

[행, 열]

In [98]: `data.loc['Colorado', ['two', 'three']]`

Out [98]: two 5  
three 6  
Name: Colorado, dtype: int32

In [118]: `data.iloc[[1,3],[3,0,1]]`

Out [118]:

	four	one	two
Colorado	7	0	5
New York	15	12	13

In [120]: `data.iloc[2]`

Out [120]: one 8  
two 9  
three 10  
four 11  
Name: Utah, dtype: int32

In [121]: `data.loc[:, 'two']`

Out [121]: Ohio 0  
Colorado 5  
Utah 9  
Name: two, dtype: int32

- .iloc : integer position를 통해 값을 찾을 수 있음. label로는 찾을 수 없음
- .loc : label을 통해 값을 찾을 수 있음. Integer position으로 찾을 수 없음.



```
In [2]: import pandas as pd
import numpy as np

df = pd.DataFrame({'AAA': [4, 5, 6, 7], 'BBB': [10, 20, 30, 40], 'CCC': [100, 50, -30, -50]})
```

- ① df에서 AAA가 5보다 크거나 같은 경우의 BBB를 -1로 바꾸시오.  
결과 화면>>

	AAA	BBB	CCC
0	4	10	100
1	5	-1	50
2	6	-1	-30
3	7	-1	-50

- ② Df에서 AAA가 5보다 크거나 같은 경우의 BBB와 CCC를 555로 바꾸시오.

결과 화면>>

	AAA	BBB	CCC
0	4	10	100
1	5	555	555
2	6	555	555
3	7	555	555

- ③ Df에서 AAA가 5보다 작은 BBB와 CCC를 2000으로 바꾸시오.

결과 화면>>

	AAA	BBB	CCC
0	4	2000	2000
1	5	555	555
2	6	555	555
3	7	555	555



- Index가 다른 객체간의 산술 연산
- 객체를 더할 때 짝이 맞지 않은 index가 있다면...

```
In [136]: s1 = Series([7.3, -2.5, 3.4, 1.5], index = ['a','c','d','e'])  
          s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index = ['a','c','e','f','g'])  
          print(s1)  
          print(s2)
```

```
a    7.3  
c   -2.5  
d    3.4  
e    1.5  
dtype: float64  
a   -2.1  
c    3.6  
e   -1.5  
f    4.0  
g    3.1  
dtype: float64
```

```
In [137]: s1+s2
```

서로 겹치는 index가 없으면 데이터는 NaN이 됨

```
Out [137]: a    5.2  
          c    1.1  
          d   NaN  
          e    0.0  
          f   NaN  
          g   NaN  
          dtype: float64
```

```
In [141]: df1 = DataFrame(np.arange(9.).reshape((3,3)), columns = list('bcd'),
                        index=['Ohio', 'Texas', 'Colorado'])
df2 = DataFrame(np.arange(12.).reshape((4,3)), columns = list('bde'),
                index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df1
```

Out [141]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [142]: df2

Out [142]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [143]: df1+df2

Out [143]:

	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



- DataFrame을 사용하여 다음 표를 만드시오.

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

df1

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

df2

```
In [144]: df1 = DataFrame(np.arange(12.).reshape((3,4)), columns=list('abcd'))  
          df2 = DataFrame(np.arange(20.).reshape((4,5)), columns=list('abcde'))  
          df1
```

Out [144]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

In [145]: df2

Out [145]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [146]: df1+df2

Out [146]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

서로 다른 index를 가지는 객체 간에 산술 연산에서 존재하지 않는 값을 특수한 값(0)으로 지정(add 메소드와 fill\_value 활용)

```
In [147]: df1.add(df2, fill_value = 0)
```

Out [147]:

	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

```
In [148]: df1.reindex(columns = df2.columns, fill_value = 0)
```

Out [148]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0



# DataFrame과 시리즈 간의 작업

```
In [8]: arr = np.arange(12.).reshape((3,4))  
arr
```

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

```
In [9]: arr[0]
```

```
Out [9]: array([0., 1., 2., 3.])
```

```
In [10]: arr - arr[0]
```

```
Out [10]: array([[0., 0., 0., 0.],  
                [4., 4., 4., 4.],  
                [8., 8., 8., 8.]])
```

```
In [12]: frame = DataFrame(np.arange(12.).reshape(4,3), columns = list('bde'),  
                           index = ['Utah', 'Ohio', 'Texas', 'Oregon'])  
frame
```

Out[12]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [13]: series = frame.iloc[0]  
series
```

Out[13]:

b	0.0
d	1.0
e	2.0

Name: Utah, dtype: float64

```
In [14]: frame - series
```

Out[14]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

```
In [16]: series1 = frame.loc['Utah']  
series1
```

Out[16]:

b	0.0
d	1.0
e	2.0

Name: Utah, dtype: float64

```
In [17]: frame - series1
```

Out[17]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Out [12]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

DataFrame과 Series인 두 객체를 연산하면 행과 열을 통합하고, 짝이 안 맞으면 NaN값이 된다. (서로 겹치는 index가 없다면 데이터는 NaN이 된다.)

```
In [21]: series2 = Series(range(3), index=['b', 'e', 'f'])
series2
```

Out [21]:

b	0
e	1
f	2

dtype: int64

```
In [22]: frame + series2
```

Out [22]:

	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 [24]: series3 = frame['d']
series3
```

Out [24]:

Utah	1.0
Ohio	4.0
Texas	7.0
Oregon	10.0

Name: d, dtype: float64

```
In [25]: frame.sub(series3, axis = 0)
```

Out [25]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

각 행에 대해 연산을 수행하고 싶다면 산술 연산 메서드(add, sub)를 사용

- Pandas 객체에서도 Numpy의 유니버설 함수(배열의 각 원소에 적용되는 메서드)를 적용가능

```
In [26]: frame = DataFrame(np.random.randn(4,3), columns = list('bde'),  
                           index = ['Utah', 'Ohio', 'Texas', 'Oregon'])  
frame
```

Out [26]:

	b	d	e
Utah	-0.334644	-1.401216	-0.145720
Ohio	-0.844773	-0.258680	-0.623687
Texas	-1.105680	0.322099	-1.533654
Oregon	0.096238	0.157943	-1.536680

Randn은 기대값이 0이고, 표준편차가 1인  
가우시안 표준 정규 분포를 따르는 난수를 생성

```
In [27]: np.abs(frame) # 절대값
```

Out [27]:

	b	d	e
Utah	0.334644	1.401216	0.145720
Ohio	0.844773	0.258680	0.623687
Texas	1.105680	0.322099	1.533654
Oregon	0.096238	0.157943	1.536680

	b	d	e
Utah	<a href="#">-0.334644</a>	-1.401216	-0.145720
Ohio	-0.844773	<a href="#">-0.258680</a>	<a href="#">-0.623687</a>
Texas	<a href="#">-1.105680</a>	<a href="#">0.322099</a>	-1.533654
Oregon	0.096238	0.157943	-1.536680

```
In [28]: f = lambda x: x.max() - x.min()
```

lambda 함수 : 함수의 이름 없이 함수처럼 사용할 수 있는 익명의 함수

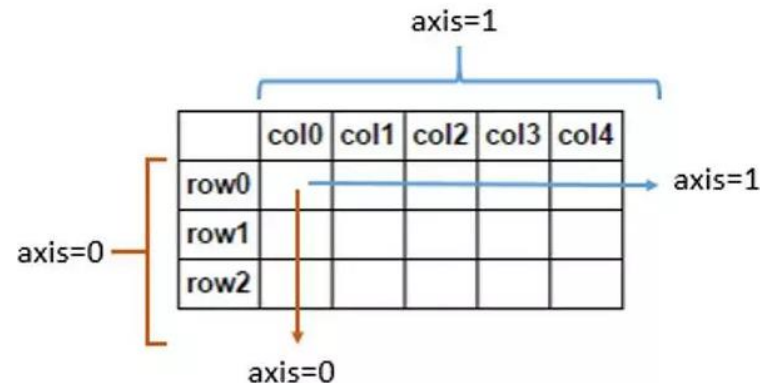
```
In [30]: frame.apply(f)
```

```
Out[30]: b    1.201919
         d    1.723315
         e    1.390960
         dtype: float64
```

각 열 또는 행에 1차원 배열에 함수를 적용할때 apply 메소드를 사용

```
In [31]: frame.apply(f, axis = 1)
```

```
Out[31]: Utah    1.255496
         Ohio    0.586093
         Texas   1.855753
         Oregon  1.694623
         dtype: float64
```



- 람다(lambda,  $\lambda$ )

- 수리논리학에서의 함수 정의를 추상화한 형식 체계
  - ✓ 이름이 없는 함수
- 수학에서의 람다대수의 정의와 비슷하게 파이썬에서 정의
  - ✓ `lambda { 파라미터,...} : 표현식`
- 람다식은 그 자체로 표현식이며 다음 구성요소로 작성
  1. 키워드 `lambda`
  2. 파라미터 : 콤마로 구분되는 1개 이상의 파라미터. 파라미터는 반드시 1개 이상
  3. 콜론 :
  4. 표현식 : 파라미터와 그 외 값으로 이루어지는 일련의 표현식
- 어떤 파라미터 `x`에 대해 1을 증가시킨 값을 구하는 함수는 `lambda x : x+1`로 표현
- 두수를 더하는 함수를 람다식으로 표현 `lambda x,y : x + y`
- 두수를 곱하는 함수를 람다식으로 표현 `lambda x,y : x * y`

람다식은 파라미터와 표현식을 결합한 것이며, 그 자체로는 파라미터를 표현식에 대입하여 값을 평가하는 객체를 뜻하므로 함수로 평가. 즉, 람다식 = 익명 함수

- 람다 함수는 그 자체로도 실제 이름이 없는 함수이며, 표현식
  - 대입 구문에서 우변에 사용될 수 있고, 변수에 바인딩하여 이름을 부여할 수 있음
  - 일반 함수와 사실상 동일하며, 람다식을 대입문에 사용하고, 변수명을 사용해서 함수 호출과 같은 방식으로 실행 가능

```
add = lambda x, y : x+y  
add(3, 4)
```

# map(), filter(), reduce()

## • 람다표현식이 많이 쓰이는 곳이 map(), filter(), reduce()

- 이 세가지 연산은 리스트(혹은 연속열)의 원소를 변환하거나, 특정 조건으로 필터링하거나, 혹은 여러 개의 값을 차곡차곡 접어서 하나의 값으로 압축하는 작업으로 사실상 리스트와 관련된 대부분의 연산작업
- 파이썬3에서 map(), filter()의 결과는 리스트가 아니라 각각 map, filter 객체이다. 이들은 제너레이터로 연속열로 만약 map(), filter()를 사용하여 리스트를 얻고 싶으면 그 결과를 다시 list()에 인자로 집어넣어 리스트로 변환해야 함

```
In [1]: xs = list(range(10))
xs
```

```
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

각 수를 2배

```
In [2]: doubled = map(lambda x: x*2, xs)
doubled
```

```
Out[2]: <map at 0x22db2192ac8>
```

```
In [3]: for x in doubled:
        print(x)
```

```
0
2
4
6
8
10
12
14
16
18
```

각 수에 1을 더한 후 제곱

```
In [4]: squared = map(lambda x: (x+1)**2, xs)
list(squared)
```

```
Out[4]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
doubled = [x*2 for x in xs]
```

```
squared = [(x+1)**2 for x in xs]
```



# map(), filter()

```
In [1]: xs = list(range(10))
        xs
```

Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

각 수를 2배

```
In [2]: doubled = map(lambda x: x*2, xs)
        doubled
```

Out[2]: <map at 0x22db2192ac8> ???

```
In [3]: for x in doubled:
        print(x)
```

0  
2  
4  
6  
8  
10  
12  
14  
16  
18

각 수에 1을 더한 후 제곱

```
In [4]: squared = map(lambda x : (x+1) **2, xs)
        list(squared)
```

Out[4]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

리스트 축약 : 리스트 축약은 람다식의 본체가 될 표현식을 그대로 사용하므로 따로 람다 함수를 정의할 필요가 없음

[{표현식} for {변수} in {반복자/연속열} if {조건 표현식}]

```
doubled = [x*2 for x in xs]
```

```
squared = [(x+1)**2 for x in xs]
```

짝수로 필터링

```
In [5]: evens = filter(lambda x:x%2 is 0, xs)
        list(evens)
```

Out[5]: [0, 2, 4, 6, 8]

```
evens = [x for x in xs if x % 2 is 0]
```

range()는 일종의 제너레이터로 느긋하게 평가되는 연속열 → 필요한 시점에 필요한 원소가 하나씩 생성되고 평가됨 (map(), filter())의 결과가 리스트가 아닌 제너레이터로 느긋하게 평가되는 연속열이므로 한번에 하나의 원소가 필요한 시점에 만들어지므로 메모리를 많이 사용안함)  
리스트 축약을 사용하게 되면 리스트 축약 자체가 리스트로 평가되는 표현식이므로 제너레이터 내의 모든 원소가 한번에 평가되어 리스트가 생성됨(편리한 대신 전체 연속열을 모두 평가해서 리스트로 만들기때문에 메모리 낭비가 심함)

## • 동작원리

1. 단계를 거듭할 수록 값의 상태의 개수를 점점 줄여나가야 하므로 연산(함수)은 2인자 함수(2개의 값을 받아서 하나의 결과값을 리턴)여야 함
2. 연산을 계속해나가면서 중간 값을 누적시켜 나갈 변수가 필요하다. 이 변수는 최초의 초기값을 필요로 함
3. 중간 값과 리스트의 맨 앞에 있는 값. 이렇게 두 값을 1의 함수에 넣어서 평가하고 그 결과로 중간값을 업데이트
4. 다시 중간 값과 그 다음 리스트 원소 값을 함수에 넣어 평가하고 그 결과로 중간 값을 업데이트
5. 리스트의 끝에 다다를 때까지 4의 과정을 반복
6. 리스트에 더 이상의 원소가 없으면 중간 값이 최종 값

# reduce() 예제

- reduce() 메소드를 이용해서 [1,2,3,4]를 +를 사용해서 축약하는 과정

중간값	리스트	평가식
0	[1, 2, 3, 4]	-> 0 + 1 #1
1	[2, 3, 4]	-> 1 + 2 #2
3	[3, 4]	-> 3 + 3
6	[4]	-> 6 + 4
10	[]	:: 완료

- 최초 중간 값은 0이며, 리스트의 맨 첫 값은 1이다. 이 둘을 더한다. 그 결과는 1이며 중간값은 1이 된다.
- 다시 중간 값과 리스트의 첫 값을 더한다.
- 이 과정을 반복하여 리스트가 빌 때까지 진행한다.
- 완료된 후의 중간 값은 10이며 이 값은 1+2+3+4이다. 즉, 리스트의 합계이다.

```
In [9]: from functools import reduce
xs = range(5)
total = reduce(lambda x,y : x+y, xs)
total
```

Out [9]: 10

```
from functools import reduce
```

```
li = [-3, -2, 0, 6, 8]
```

**1. li의 최대값을 구하여 max\_xs에 저장하여라.**

출력화면 >> 8

**2. li의 숫자들이 음수면 음수, 0이면 0, 양수면 양수를 출력하여라.**

출력화면 > ['음수', '음수', 0, '양수', '양수']

**3. map()을 이용하여 리스트 a와 리스트 b의 합을 구하여 출력하라.**

```
a = [1,2,3,4]
```

```
b = [5,6,7,8]
```

출력화면 > [6, 8, 10, 12]

	b	d	e
Utah	-0.334644	-1.401216	-0.145720
Ohio	-0.844773	-0.258680	-0.623687
Texas	-1.105680	0.322099	-1.533654
Oregon	0.096238	0.157943	-1.536680

```
In [32]: def f(x):
          return Series([x.min(), x.max()], index = ['min', 'max'])

          frame.apply(f)
```

Apply 메서드에 전달된 함수는 스칼라 값, Series 또는 DataFrame을 반환해도 됨

Out [32]:

	b	d	e
min	-1.105680	-1.401216	-1.536680
max	0.096238	0.322099	-0.145720

applymap을 이용하여 배열의 각 원소에 적용되는 함수를 사용할 수 있음.

```
In [33]: format = lambda x: '%.2f' % x
          frame.applymap(format)
```

applymap은 Series에 요소별 함수를 적용하기 위한 map 메소드

Out [33]:

	b	d	e
Utah	-0.33	-1.40	-0.15
Ohio	-0.84	-0.26	-0.62
Texas	-1.11	0.32	-1.53
Oregon	0.10	0.16	-1.54

```
In [34]: frame['e'].map(format)
```

```
Out [34]: Utah      -0.15
           Ohio      -0.62
           Texas     -1.53
           Oregon    -1.54
           Name: e, dtype: object
```

## - map 함수

- ✓ DataFrame 타입이 아니라 반드시 **Series 타입에서만 사용**
  - Series는 Numpy에서 제공하는 1차원 배열과 비슷하지만 각 데이터의 의미를 표시하는 인덱스(index)를 붙일 수 있다.
  - 하지만, 데이터 자체는 그냥 값(value)의 1차원 배열

## - apply 함수

- ✓ 사용자 함수를 사용하기 위해 DataFrame에서 복수 개의 열이 필요할 경우 사용
- ✓ **Series, DataFrame 모두에서 사용가능**

## - applymap 함수

- ✓ 각 원소에 적용되는 함수
- ✓ **DataFrame에서만 적용**

- Sort\_index 메서드를 이용하면 행이나 열의 색인을 알파벳순으로 정렬된 새로운 객체를 반환
- DataFrame은 행이나 열 중 하나의 축을 기준으로 정렬
  - ✓ 기본적으로 행이 기준이 되고, axis=1 옵션으로 열 기준

```
In [10]: obj = pd.Series(range(4), index = ['d','a','b','c'])  
obj.sort_index()
```

```
Out[10]: a    1  
        b    2  
        c    3  
        d    0  
        dtype: int64
```

```
In [13]: frame = pd.DataFrame(np.arange(8).reshape(2,4), index = ['three','one'], columns=['d','a','b','c'])  
frame.sort_index()
```

```
Out[13]:
```

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

```
In [14]: frame.sort_index(axis = 1)
```

```
Out[14]:
```

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

- ascending = False 옵션을 이용하여 내림차순으로 정렬 가능

```
In [16]: frame
```

```
Out[16]:
```

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

```
In [17]: frame.sort_index(axis = 1, ascending = False)
```

```
Out[17]:
```

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

- Series의 값으로 정렬 가능(NaN은 가장 마지막에 위치)

```
In [19]: obj = pd.Series([4,7,-3,2])  
obj.sort_values()
```

```
Out[19]: 2    -3  
         3     2  
         0     4  
         1     7  
         dtype: int64
```



- DataFrame은 by옵션으로 하나 이상의 열을 정렬할 수 있음

```
In [21]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})  
frame
```

Out[21]:

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

```
In [22]: frame.sort_values(by = 'b')
```

Out[22]:

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

```
In [24]: frame.sort_values(by=['a', 'b'])
```

Out[24]:

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

## -rank() 메서드로 순위 매기기

- ✓ 동점인 항목에 대해서는 평균 순위를 매김

```
In [27]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])  
obj.rank()
```

```
Out[27]: 0    6.5  
         1    1.0  
         2    6.5  
         3    4.5  
         4    3.0  
         5    2.0  
         6    4.5  
         dtype: float64
```

- ✓ method로 같은 값이 있을 경우 다르게 순위를 매길 수 있음

- 'average' : 같은 값을 가지는 항목의 평균 값을 순위로 삼는다.(기본값)
- 'min' : 같은 값을 가지는 그룹을 낮은 순위로 매긴다.
- 'max' : 같은 값을 가지는 그룹을 높은 순위로 매긴다.
- 'first' : 데이터 내에서 위치에 따라 순위를 매긴다.

```
In [36]: obj.rank(method = 'first')
```

```
Out[36]: 0    6.0  
         1    1.0  
         2    7.0  
         3    4.0  
         4    3.0  
         5    2.0  
         6    5.0  
         dtype: float64
```

```
In [37]: obj.rank(method = 'min')
```

```
Out[37]: 0    6.0  
         1    1.0  
         2    6.0  
         3    4.0  
         4    3.0  
         5    2.0  
         6    4.0  
         dtype: float64
```

```
In [38]: obj.rank(method = 'max')
```

```
Out[38]: 0    7.0  
         1    1.0  
         2    7.0  
         3    5.0  
         4    3.0  
         5    2.0  
         6    5.0  
         dtype: float64
```

```
In [40]: obj
```

```
Out[40]: 0    7  
         1   -5  
         2    7  
         3    4  
         4    2  
         5    0  
         6    4  
         dtype: int64
```

- ascending 옵션으로 내림차순으로 순위를 매길 수 있음

```
In [39]: obj.rank(ascending=False)
```

```
Out[39]: 0    1.5  
         1    7.0  
         2    1.5  
         3    3.5  
         4    5.0  
         5    6.0  
         6    3.5  
         dtype: float64
```

```
In [41]: obj.rank(ascending=False, method = 'max')
```

```
Out[41]: 0    2.0  
         1    7.0  
         2    2.0  
         3    4.0  
         4    5.0  
         5    6.0  
         6    4.0  
         dtype: float64
```

- DataFrame에서는 axis 옵션으로 행에 대한 순위를 정할 수 있음

```
In [43]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2],  
                                'a': [0, 1, 0, 1],  
                                'c': [-2, 5, 8, -2.5]})  
frame
```

Out[43]:

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

```
In [44]: frame.rank()
```

Out[44]:

	b	a	c
0	3.0	1.5	2.0
1	4.0	3.5	3.0
2	1.0	1.5	4.0
3	2.0	3.5	1.0

```
In [45]: frame.rank(axis = 1)
```

Out[45]:

	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



- Pandas의 많은 함수(reindex 같은)에서 인덱스 값은 유일해야 하지만 강제사항은 아님

```
In [47]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])  
obj
```

```
Out[47]: a    0  
         a    1  
         b    2  
         b    3  
         c    4  
dtype: int64
```

- is\_unique 속성으로 인덱스가 유일한지 확인 가능

```
In [48]: obj.index.is_unique
```

```
Out[48]: False
```

- 데이터 선택 시 중복되는 인덱스가 있으면 Series 객체를 반환하고, 중복되는 색인이 없으면 스칼라 값을 반환

```
In [49]: obj['a']
```

```
Out[49]: a    0  
         a    1  
dtype: int64
```

```
In [50]: obj['c']
```

```
Out[50]: 4
```

## - DataFrame도 동일

```
In [52]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])  
df
```

Out [52]:

	0	1	2
a	0.998046	-0.624470	-0.251459
a	1.469155	-0.601471	0.859829
b	1.328036	0.308077	0.148977
b	2.227829	1.761991	1.580657

```
In [53]: df.loc['b']
```

Out [53]:

	0	1	2
b	1.328036	0.308077	0.148977
b	2.227829	1.761991	1.580657

```
In [55]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                             [np.nan, np.nan], [0.75, -1.3]],
                             index = ['a', 'b', 'c', 'd'],
                             columns = ['one', 'two'])
df
```

Out [55]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

- sum()메소드를 호출하면 각 열의 합을 담은 Series를 반환

✓ axis = 1 옵션을 넘기면 각 행의 합을 반환

```
In [56]: df.sum()
```

```
Out [56]: one    9.25
          two   -5.80
          dtype: float64
```

```
In [62]: df.sum(axis = 1)
```

```
Out [62]: a    1.40
          b    2.60
          c    0.00
          d   -0.55
          dtype: float64
```

- mean()메소드를 호출시키면 각 열의 평균을 담은 Series를 반환

✓ axis = 1 옵션은 각 행의 평균을 반환

✓ skipna 옵션이 False일 경우 NaN이 포함된 행은 평균값이 제대로 계산되지 않고, NaN이라고 표시

```
In [63]: df.mean()
```

```
Out [63]: one    3.083333
          two   -2.900000
          dtype: float64
```

```
In [65]: df.mean(axis = 1)
```

```
Out [65]: a    1.400
          b    1.300
          c    NaN
          d   -0.275
          dtype: float64
```

```
In [64]: df.mean(axis = 1, skipna = False)
```

```
Out [64]: a    NaN
          b    1.300
          c    NaN
          d   -0.275
          dtype: float64
```

- idxmin(), idxmax() 메소드는 전체 인덱스 중 최소값, 최대값을 반환

```
In [67]: df.idxmin()
```

```
Out [67]: one    d
          two    b
          dtype: object
```

```
In [68]: df.idxmax()
```

```
Out [68]: one    b
          two    d
          dtype: object
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

- cumsum() 메서드는 맨 첫 번째 성분 부터 각 성분까지의 누적합을 계산

```
In [69]: df.cumsum()
```

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

- describe() 메소드는 생성했던 DataFrame의 간단한 통계정보를 보여줌

```
In [71]: df.describe()
```

	one	two
count	3.000000	<a href="#">2.000000</a>
mean	3.083333	<a href="#">-2.900000</a>
std	3.493685	<a href="#">2.262742</a>
min	0.750000	-4.500000
25%	<a href="#">1.075000</a>	-3.700000
50%	1.400000	<a href="#">-2.900000</a>
75%	<a href="#">4.250000</a>	<a href="#">-2.100000</a>
max	7.100000	-1.300000

열별 데이터의 개수,  
데이터 평균값,  
표준변차,  
최소값,  
4분위수(25%,  
50%,  
75%),  
최대값의 정보를 알 수 있음



```
In [76]: obj = pd.Series(['a','a','b','c'] * 4)
obj
```

```
Out[76]: 0    a
         1    a
         2    b
         3    c
         4    a
         5    a
         6    b
         7    c
         8    a
         9    a
        10    b
        11    c
        12    a
        13    a
        14    b
        15    c
dtype: object
```

```
In [77]: obj.describe()
```

```
Out[77]: count      16    데이터의 개수,
         unique      3    중복값을 제외한 유니크한 값의 수,
         top         a    가장 많이 반복되는 데이터,
         freq        8    반복 빈도 수,
         dtype: object    데이터 타입의 정보를 알 수 있음
```

# Unique value

```
In [80]: obj = pd.Series(['c','a','d','a','a','b','b','c','c'])
obj
```

```
Out[80]: 0    c
         1    a
         2    d
         3    a
         4    a
         5    b
         6    b
         7    c
         8    c
         dtype: object
```

- **unique() : 유일한 값 찾기**

- Series에서 고유한 값의 배열을 반환
  - ✓ NaN도 포함

```
In [81]: uniques = obj.unique()
uniques
```

```
Out[81]: array(['c', 'a', 'd', 'b'], dtype=object)
```

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

- `value_counts()` : 유일한 값 별 개수 세기

```
In [87]: obj.value_counts()
```

```
Out[87]: a    3
         c    3
         b    2
         d    1
         dtype: int64
```

- `normalize` 옵션이 `False`면 개수, `True`면 상대 비율을 구함

```
In [90]: obj.value_counts(normalize = True)
```

```
Out[90]: a    0.333333
         c    0.333333
         b    0.222222
         d    0.111111
         dtype: float64
```

```
In [91]: obj.value_counts(normalize = False)
```

```
Out[91]: a    3
         c    3
         b    2
         d    1
         dtype: int64
```

- `sort` 옵션이 `True`면 개수 기준 정렬, `False`면 유일한 값 기준 정렬

```
In [94]: obj.value_counts(sort = True, ascending = False) #default
```

```
Out[94]: a    3
         c    3
         b    2
         d    1
         dtype: int64
```

유일한 값의 개수 기준 내림차순 정렬

```
In [95]: obj.value_counts(sort = True, ascending = True)
```

```
Out[95]: d    1
         b    2
         c    3
         a    3
         dtype: int64
```

유일한 값의 개수 기준 오름차순 정렬

- **isin() 메소드 : 값들을 포함한 구성원을 확인할 때 쓰임**

- Series나 DataFrame의 열에 있는 값을 필터링할 때 유용

```
In [104]: obj.isin(['b','c'])
```

```
Out [104]: 0    True
           1   False
           2   False
           3   False
           4   False
           5    True
           6    True
           7    True
           8    True
           dtype: bool
```

```
In [105]: mask = obj.isin(['b','c'])
           obj[mask]
```

```
Out [105]: 0    c
           5    b
           6    b
           7    c
           8    c
           dtype: object
```

True인 인덱스와 그 인덱스의 값들이 출력

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



```
In [107]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],  
                               'Qu2': [2, 3, 1, 2, 3],  
                               'Qu3': [1, 5, 2, 4, 4]})  
data
```

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

## • DataFrame의 여러 행에 대해 히스토그램을 구해야 하는 경우

- 각 열에서 특정 값이 몇 번 쓰였는지 알고 싶음
- DataFrame의 apply함수에 pd.value\_counts를 넘김
- value\_counts의 결과가 DataFrame의 열의 크기보다 작을 수 있으므로 fillna(0)함수를 이용해서 비어있는 값은 0으로 채움

```
In [108]: result = data.apply(pd.value_counts).fillna(0)
```

```
In [109]: result
```

Out [109]:

	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

- **isnull() : 누락된 데이터를 찾음**

- NaN이나 None일 경우 True, 그렇지 않을 경우 False

```
In [112]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

```
In [113]: string_data
```

```
Out [113]: 0    aardvark
           1    artichoke
           2         NaN
           3     avocado
           dtype: object
```

```
In [114]: string_data.isnull()
```

```
Out [114]: 0    False
           1    False
           2     True
           3    False
           dtype: bool
```

```
In [115]: string_data[0]=None
           string_data
```

```
Out [115]: 0         None
           1    artichoke
           2         NaN
           3     avocado
           dtype: object
```

```
In [117]: string_data.isnull()
```

```
Out [117]: 0     True
           1    False
           2     True
           3    False
           dtype: bool
```

## NaN과 None의 차이?

NaN(np.nan)은 missing value(결측치, 잘못된 값)를 표현하고, pandas의 산술통계 메서드 사용시 NaN을 제외하고 연산  
None은 undefined(아직 정해지지 않은 값)를 의미하고, Python에서 None에 대한 연산은 예외를 발생

```
In [118]: data = pd.Series([1,np.nan, 3.5, np.nan, 7])
data
```

```
Out [118]: 0    1.0
           1    NaN
           2    3.5
           3    NaN
           4    7.0
           dtype: float64
```

## • dropna() : 결측값(NaN) 있는 Series, 또는 행, 열 제거

- 관측 값이 아주 많고 결측 값이 별로 없는 경우에는 결측 값이 들어있는 행 전체를 삭제하고 분석을 진행해도 무리가 없고 편리할 수 있음
- 혹은 특정 변수의 결측 값 비율이 매우 높고, 결측 값을 채워넣을 만한 마땅한 방법이 없는 경우에는 분석의 신뢰성 확보를 위해서 그 변수(행)을 삭제하고 분석을 진행할 필요도 있음

```
In [119]: data.dropna()
```

```
Out [119]: 0    1.0
           2    3.5
           4    7.0
           dtype: float64
```

## • notnull() : 관측치가 결측(NaN)이면 False, 결측이 아니면 True를 반환

```
In [120]: data.notnull()
```

```
Out [120]: 0    True
           1   False
           2    True
           3   False
           4    True
           dtype: bool
```

```
In [121]: data[data.notnull()]
```

```
Out [121]: 0    1.0
           2    3.5
           4    7.0
           dtype: float64
```

# 누락된 데이터 필터링

91

```
In [123]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],  
                                [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])  
data
```

	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

- DataFrame에서 결측 값(NaN)있는 행, 열 제거
  - ✓ axis = 0 : 결측 값 있는 행 삭제(default)
  - ✓ axis = 1 : 결측 값 있는 열 삭제
  - ✓ how = 'all' : 모든 행(axis=0), 열(axis=1)의 값이 NaN일 때만 제거

```
In [127]: cleaned = data.dropna()  
cleaned
```

Out [127]:

	0	1	2
0	1.0	6.5	3.0

```
In [126]: cleaned2 = data.dropna(axis = 1)  
cleaned2
```

Out [126]:

0
1
2
3

```
In [128]: data.dropna(how = 'all')
```

Out [128]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

```
In [129]: data.dropna(axis = 1, how = 'all')
```

Out [129]:

	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 [132]: data[4]=np.nan  
data
```

Out [132]:

	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 [133]: data.dropna(axis = 1, how = 'all')
```

Out [133]:

	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

how = 'all' : 모든 값이 누락된 행이나 열만 삭제  
how = 'any' : 누락된 값을 포함하는 행이나 열이  
삭제  
(axis값에 따라 행/열이 결정)

```
In [135]: df = pd.DataFrame(np.random.randn(7,3))
df
```

randn은 기대값이 0이고 표준편차가 1인 가우시안 표준 정규분포를 따르는 난수를 생성

Out [135]: NaN으로 채움

	0	1	2
0	-0.386413	1.426523	0.692234
1	<a href="#">0.287284</a>	-0.944787	<a href="#">0.108808</a>
2	<a href="#">-0.249313</a>	1.318024	-0.902918
3	1.589420	0.152824	-1.595507
4	<a href="#">-1.171166</a>	<a href="#">2.482867</a>	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

```
In [136]: df.loc[:4,1] = np.nan
df
```

Out [136]: NaN으로 채움

	0	1	2
0	-0.386413	NaN	0.692234
1	<a href="#">0.287284</a>	NaN	<a href="#">0.108808</a>
2	<a href="#">-0.249313</a>	NaN	-0.902918
3	1.589420	NaN	-1.595507
4	<a href="#">-1.171166</a>	NaN	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

```
In [137]: df.loc[:2,2] = np.nan
df
```

Out [137]:

	0	1	2
0	-0.386413	NaN	NaN
1	<a href="#">0.287284</a>	NaN	NaN
2	<a href="#">-0.249313</a>	NaN	NaN
3	1.589420	NaN	-1.595507
4	<a href="#">-1.171166</a>	NaN	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

```
In [137]: df.loc[:,2] = np.nan  
df
```

Out [137]:

	0	1	2
0	-0.386413	NaN	NaN
1	<a href="#">0.287284</a>	NaN	NaN
2	<a href="#">-0.249313</a>	NaN	NaN
3	1.589420	NaN	-1.595507
4	<a href="#">-1.171166</a>	NaN	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

```
In [138]: df.dropna(thresh=3)
```

Out [138]:

	0	1	2
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

기본 axis = 0이므로 각 행에서 NaN 값이 최소 3개 이상 나와야 된다는 것  
그것보다 적게 나오면 행을 제거한다. 만약 열에 적용하려면 axis = 1로 설정하고 적용

# 누락된 데이터 채우기

- **fillna()** : 결측 값을 채움

- fillna(0) : 결측 값을 0으로 대체
- fillna('missing') : 결측값을 missing이라는 문자열로 대체

```
In [139]: df.fillna(0)
```

Out [139]:

	0	1	2
0	-0.386413	0.000000	0.000000
1	<a href="#">0.287284</a>	0.000000	0.000000
2	<a href="#">-0.249313</a>	0.000000	0.000000
3	1.589420	0.000000	-1.595507
4	<a href="#">-1.171166</a>	0.000000	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

```
In [140]: df.fillna({1:0.5,3:-1})
```

Out [140]:

	0	1	2
0	-0.386413	0.500000	NaN
1	<a href="#">0.287284</a>	0.500000	NaN
2	<a href="#">-0.249313</a>	0.500000	NaN
3	1.589420	0.500000	-1.595507
4	<a href="#">-1.171166</a>	0.500000	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873

	0	1	2
0	-0.386413	NaN	NaN
1	<a href="#">0.287284</a>	NaN	NaN
2	<a href="#">-0.249313</a>	NaN	NaN
3	1.589420	NaN	-1.595507
4	<a href="#">-1.171166</a>	NaN	0.986515
5	<a href="#">1.786818</a>	<a href="#">0.633544</a>	-0.026337
6	-0.923686	<a href="#">0.208333</a>	-0.132873



# 누락된 데이터 채우기

Out [142]:

```
In [142]: df = pd.DataFrame(np.random.randn(6,3))  
df.loc[2:,1] = np.nan; df.loc[4:,2] = np.nan  
df
```

96

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	NaN	NaN
5	-0.010977	NaN	NaN

- fillna(method = 'ffill') :  
결측값을 앞방향으로 채워 나감

```
In [143]: df.fillna(method='ffill')
```

Out [143]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	-0.054114	0.099476
3	-1.444557	-0.054114	0.008660
4	<a href="#">-0.208300</a>	-0.054114	0.008660
5	-0.010977	-0.054114	0.008660

- fillna(method = 'bfill') :  
결측값을 뒷방향으로 채워 나감

```
In [144]: df.fillna(method='bfill')
```

Out [144]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	NaN	NaN
5	-0.010977	NaN	NaN



```
In [145]: df.loc[4:,1] = 3.1; df.loc[5:,2] = 5.1  
df
```

Out [145]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	NaN
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

```
In [146]: df.fillna(method='bfill')
```

Out [146]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	<a href="#">3.100000</a>	0.099476
3	-1.444557	<a href="#">3.100000</a>	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	<a href="#">5.100000</a>
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	NaN
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

- limit = 숫자 : 앞/뒤 방향으로 결측 값 채우는 회수를 제한
  - ✓ 앞 방향이나 뒤 방향으로 채워나갈 때 fillna(limit = 1)를 사용해서 결측 값 채우는 개수를 1개로 한정

```
In [148]: df.fillna(method='ffill', limit=1)
```

Out [148]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	-0.054114	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	0.008660
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

```
In [149]: df.fillna(method='bfill', limit=1)
```

Out [149]:

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	<a href="#">3.100000</a>	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	<a href="#">5.100000</a>
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

```
In [151]: data = pd.Series([1,np.nan, 3.5, np.nan, 7])  
data
```

```
Out [151]: 0    1.0  
           1    NaN  
           2    3.5  
           3    NaN  
           4    7.0  
           dtype: float64
```

- 결측 값을 변수별 평균으로 대체하기

```
In [153]: data.mean()
```

```
Out [153]: 3.8333333333333335
```

```
In [152]: data.fillna(data.mean())
```

```
Out [152]: 0    1.000000  
           1    3.833333  
           2    3.500000  
           3    3.833333  
           4    7.000000  
           dtype: float64
```

```
df
```

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	NaN	0.099476
3	-1.444557	NaN	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	NaN
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>

```
In [154]: df.mean()
```

```
Out [154]: 0    0.171806  
           1    1.414812  
           2    1.485636  
           dtype: float64
```

```
In [155]: df.fillna(df.mean())
```

```
Out [155]:
```

	0	1	2
0	<a href="#">1.084163</a>	-0.486639	1.217269
1	0.713957	-0.054114	<a href="#">1.002776</a>
2	0.896549	1.414812	0.099476
3	-1.444557	1.414812	0.008660
4	<a href="#">-0.208300</a>	<a href="#">3.100000</a>	1.485636
5	-0.010977	<a href="#">3.100000</a>	<a href="#">5.100000</a>





- 축에 대해 다중(둘 이상) 색인(multi-index) 단계를 지정할 수 있도록 해줌
  - 고차원 데이터를 낮은 차원의 형식으로 다룰 수 있게 해주는 기능
  - 인덱스의 개수, 상위 level & 하위 level의 개수가 일치 해야함

```
In [158]: data=pd.Series(np.random.randn(10),  
                        index=[['a','a','a','b','b','b','c','c','d','d'],  
                        [1,2,3,1,2,3,1,2,2,3]])  
  
data
```

```
Out [158]: a 1    0.079198  
          2   -0.614335  
          3   -0.447976  
          b 1   -2.063882  
          2    0.831806  
          3   -0.770576  
          c 1   -1.923613  
          2   -0.167216  
          d 2    0.947828  
          3    0.359486  
          dtype: float64
```

(상위계층, 하위계층)

```
In [160]: data['b']
```

```
Out [160]: 1   -2.063882  
          2    0.831806  
          3   -0.770576  
          dtype: float64
```

```
In [165]: data[:,2]
```

```
Out [165]: a   -0.614335  
          b    0.831806  
          c   -0.167216  
          d    0.947828  
          dtype: float64
```

```
In [161]: data['b':'c']
```

```
Out [161]: b 1   -2.063882  
          2    0.831806  
          3   -0.770576  
          c 1   -1.923613  
          2   -0.167216  
          dtype: float64
```

```
In [163]: data.loc[['b','c']]
```

```
Out [163]: b 1   -2.063882  
          2    0.831806  
          3   -0.770576  
          c 1   -1.923613  
          2   -0.167216  
          dtype: float64
```

# hierarchical indexing(계층적 색인)

101

- `unstack()` : 색인의 최하위계층을 열의 최하위 계층으로 올림(index의 열화)

```
In [166]: data.unstack()
```

```
Out [166]:
```

	1	2	3
a	0.079198	<a href="#">-0.614335</a>	-0.447976
b	<a href="#">-2.063882</a>	0.831806	-0.770576
c	<a href="#">-1.923613</a>	<a href="#">-0.167216</a>	NaN
d	NaN	0.947828	0.359486

면 level로 지정

```
In [167]: data.unstack(level=0)
```

```
Out [167]:
```

	a	b	c	d
1	0.079198	<a href="#">-2.063882</a>	<a href="#">-1.923613</a>	NaN
2	-0.614335	0.831806	-0.167216	0.947828
3	-0.447976	-0.770576	NaN	0.359486

```
a 1 0.079198
   2 -0.614335
   3 -0.447976
b 1 -2.063882
   2 0.831806
   3 -0.770576
c 1 -1.923613
   2 -0.167216
d 2 0.947828
   3 0.359486
dtype: float64
```



- `stack()` : DataFrame의 최하위층 열자체가 색인의 최하위 색인층으로 붙게되면서 Series가 됨(열의 index화)

```
In [166]: data.unstack()
```

```
Out [166]:
```

	1	2	3
a	0.079198	<a href="#">-0.614335</a>	-0.447976
b	<a href="#">-2.063882</a>	0.831806	-0.770576
c	<a href="#">-1.923613</a>	<a href="#">-0.167216</a>	NaN
d	NaN	0.947828	0.359486

```
In [168]: data.unstack().stack()
```

```
Out [168]:
```

```
a 1    0.079198
   2   -0.614335
   3   -0.447976
b 1   -2.063882
   2    0.831806
   3   -0.770576
c 1   -1.923613
   2   -0.167216
d 2    0.947828
   3    0.359486
dtype: float64
```

- DataFrame에서도 계층적 색인

- Series에서는 index만 2차원 리스트 형식으로 줬지만, DataFrame에서는 columns까지 2차원으로 줌

```
In [171]: frame = pd.DataFrame(np.arange(12).reshape((4,3)),  
                                index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],  
                                columns=[['Ohio', 'Ohio', 'Colorado'],  
                                           ['Green', 'Red', 'Green']])  
  
frame
```

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

- 복수 계층의 index와 columns에 이름을 붙일 때는 리스트 형식으로 줌

```
In [172]: frame.index.names = ['key1', 'key2']  
           frame.columns.names = ['state', 'color']  
           frame
```

		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	

		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	

- DataFrame의 열도 가장 상위 계층을 먼저 색인해야 함

```
In [173]: frame['Ohio']
```

		color		Green	Red
key1	key2				
a	1	0	1		
	2	3	4		
b	1	6	7		
	2	9	10		

# reordering and sorting level(계층 순서 바꾸고 정렬하기)<sup>105</sup>

- **swaplevel() : 축의 레벨(계층) 순서 변경**

- 넘겨 받은 2개의 계층 번호나 이름이 뒤바뀐 새로운 객체를 반환
- 레벨을 교환하므로 데이터 변경은 없음

		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			

```
In [174]: frame.swaplevel('key1','key2')
```

Out [174]:

		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			

## • level 옵션

- 어떤 한 축에 대해 합을 구하고 싶은 단계를 지정할 수 있는 옵션
- DataFrame과 Series의 많은 기술 통계와 요약 통계에서 사용

		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	

```
In [177]: frame.sum(level = 'key2')
```

```
Out [177]:
```

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

```
In [178]: frame.sum(level='color', axis=1)
```

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

# DataFrame 의 열 사용

107

```
frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),  
                      'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],  
                      'd': [0, 1, 2, 0, 1, 2, 3]})  
frame
```

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

- DataFrame에서 행을 선택하기 위한 색인으로 하나 이상의 열을 사용하는 것은 드물지 않다.
- 아니면 행의 색인을 DataFrame의 열로 옮기고 싶을 때 다음과 같이 한다.
- **set\_index() : 하나 이상의 컬럼을 색인으로 하는 새로운 DataFrame 생성**

```
In [182]: frame2 = frame.set_index(['c', 'd'])  
frame2
```

Out [182]:

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

```
In [183]: frame.set_index(['c', 'd'], drop=False)
```

Out [183]:

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

drop=False :  
DataFrame에서  
index로 사용한  
값들을 삭제하지 않음





- `reset_index()` : 계층적 색인 단계가 열로 이동
  - `set_index()`와 반대되는 개념

```
In [184]: frame2.reset_index()
```

Out [184]:

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

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

## • 통합방법

- 세로로 증가하는 방향으로 통합 → append()
- 가로로 증가하는 방향으로 통합 → join()
- 특정 열을 기준으로 통합 → merge()



- **append()** : DataFrame에서 cloumns가 같은 두 데이터를 세로 방향(index 증가 방향)으로 합함

**DataFrame\_data1.append(DataFrame\_data2 [,ignore\_index=True])**

- 세로방향으로 DataFrame\_data1 다음에 DataFrame\_data2가 추가 되어서 DataFrame 데이터로 반환
- ignore\_index=True를 입력하지 않으면 생성된 DataFrame 데이터에는 기존의 데이터의 index가 그대로 유지되고, 입력하면 생성된 DataFrame 데이터에는 데이터 순서대로 새로운 index가 할당

리스트

list\_data1.append(list\_data2) 실행 → list\_data1에 list\_data2가 추가되고 아무것도 반환 X  
DataFrame

DataFrame\_data1.append(DataFrame\_data2 [,ignore\_index=True])) 실행 →

DataFrame\_data1이 수정되지 않고,

DataFrame1 다음에 DataFame\_data2가 추가된 복사본이 DataFrame 데이터로 반환

```
In [26]: df1 = pd.DataFrame({'Class1': [95, 92, 98, 100],  
                             'Class2': [91, 93, 97, 99]})  
df1
```

Out [26]:

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99

```
In [28]: df1.append(df2)
```

Out [28]:

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99
0	87	85
1	89	90

```
In [27]: df2 = pd.DataFrame({'Class1': [87, 89],  
                             'Class2': [85, 90]})  
df2
```

Out [27]:

	Class1	Class2
0	87	85
1	89	90

```
In [30]: df1.append(df2, ignore_index=True)
```

Out [30]:

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99
4	87	85
5	89	90

```
In [27]: df2 = pd.DataFrame({'Class1': [87, 89],  
                             'Class2': [85, 90]})  
df2
```

Out [27]:

	Class1	Class2
0	87	85
1	89	90

```
In [31]: df3 = pd.DataFrame({'Class1': [96, 83]})  
df3
```

Out [31]:

	Class1
0	96
1	83

```
In [32]: df2.append(df3, ignore_index=True)
```

Out [32]:

	Class1	Class2
0	87	85.0
1	89	90.0
2	96	NaN
3	83	NaN

- join() : index가 같은 두 DataFrame에 대해 가로방향에 새로운 데이터를 추가

`DataFrame_data1.join(DataFrame_data2)`

- DataFrame\_data1 다음에 가로 방향으로 DataFrame\_data2가 추가되어서 DataFrame 데이터로 반환



```
In [26]: df1 = pd.DataFrame({'Class1': [95, 92, 98, 100],  
                             'Class2': [91, 93, 97, 99]})  
df1
```

Out [26]:

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99

```
In [34]: df1.join(df4)
```

Out [34]:

	Class1	Class2	Class3
0	95	91	93
1	92	93	91
2	98	97	95
3	100	99	98

```
In [33]: df4 = pd.DataFrame({'Class3': [93, 91, 95, 98]})  
df4
```

Out [33]:

	Class3
0	93
1	91
2	95
3	98

```
In [35]: df5 = pd.DataFrame({'Class4': [82, 92]})  
df5
```

Out [35]:

	Class4
0	82
1	92

```
In [36]: df1.join(df5)
```

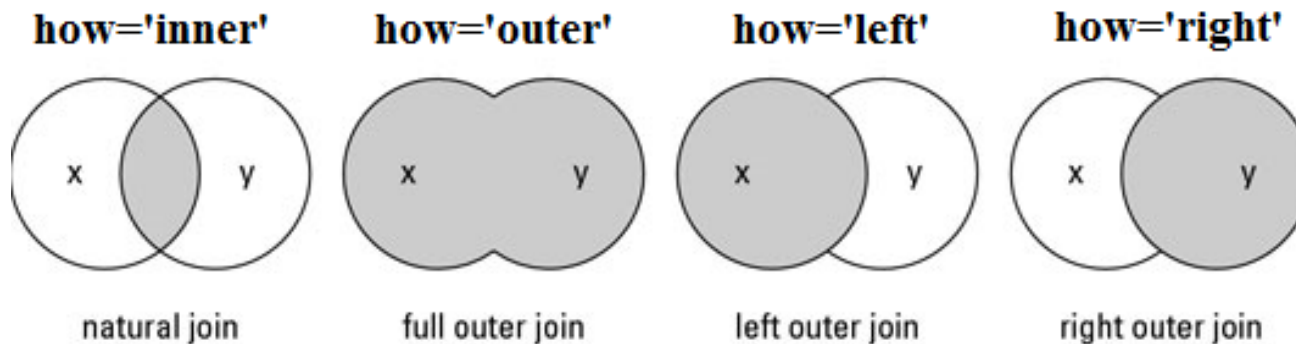
Out [36]:

	Class1	Class2	Class4
0	95	91	82.0
1	92	93	92.0
2	98	97	NaN
3	100	99	NaN



- **merge()** : 두 개의 데이터 프레임 병합

- 판다스(Pandas)에서는 데이터 프레임간에 SQL문의 테이블 간 조인 연산처럼 데이터 프레임을 합칠 수 있는 기능을 지원



- 두 DataFrame 데이터에 공통된 열이 있다면 이 열을 기준으로 두 데이터를 통합

**DataFrame\_left\_data.merge(DataFrame\_right\_data)**

- ✓ DataFrame\_left\_data(왼쪽 데이터)와 DataFrame\_right\_data(오른쪽 데이터)가 공통된 열(key) 중심으로 좌우로 통합



```
In [16]: import numpy as np
import pandas as pd

df = pd.DataFrame([{'Name': 'Chris', 'Item Purchased': 'Sponge', 'Cost': 22.50},
                   {'Name': 'Kevyn', 'Item Purchased': 'Kitty Litter', 'Cost': 2.50},
                   {'Name': 'Filip', 'Item Purchased': 'Spoon', 'Cost': 5.00}],
                  index=['Store 1', 'Store 1', 'Store 2'])
df['Date'] = ['December 1', 'January 1', 'mid-May']
df['Delivered'] = True
df['Feedback'] = ['Positive', None, 'Negative']
df
```

	Cost	Item Purchased	Name	Date	Delivered	Feedback
Store 1	22.5	Sponge	Chris	December 1	True	Positive
Store 1	2.5	Kitty Litter	Kevyn	January 1	True	None
Store 2	5.0	Spoon	Filip	mid-May	True	Negative

```
adf = df.reset_index() # 인덱스 재설정 원래 인덱스를 index column으로 빼고 0~N으로 인덱스 대체
adf['Date'] = pd.Series({0: 'December 1', 2: 'mid-May'})
adf
```

	index	Cost	Item Purchased	Name	Date	Delivered	Feedback
0	Store 1	22.5	Sponge	Chris	December 1	True	Positive
1	Store 1	2.5	Kitty Litter	Kevyn	NaN	True	None
2	Store 2	5.0	Spoon	Filip	mid-May	True	Negative

```
In [18]: staff_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR'},  
                                  {'Name': 'Sally', 'Role': 'Course liasion'},  
                                  {'Name': 'James', 'Role': 'Grader'}])  
staff_df = staff_df.set_index('Name') # Name 컬럼을 인덱스로  
staff_df                                뺐
```

Role	
Name	
Kelly	Director of HR
Sally	Course liasion
James	Grader

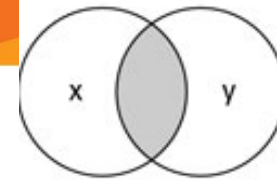
```
In [19]: student_df = pd.DataFrame([{'Name': 'James', 'School': 'Business'},  
                                     {'Name': 'Mike', 'School': 'Law'},  
                                     {'Name': 'Sally', 'School': 'Engineering'}])  
student_df = student_df.set_index('Name')  
student_df
```

School	
Name	
James	Business
Mike	Law
Sally	Engineering

# inner join (default)

how='inner'

118



staff\_df

Role	
Name	
Kelly	Director of HR
Sally	Course liasion
James	Grader

student\_df

School	
Name	
James	Business
Mike	Law
Sally	Engineering

- staff\_df 와 student\_df를 인덱스를 기준으로 inner\_join

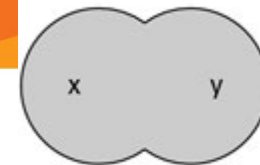
```
inner_join = pd.merge(staff_df, student_df, how='inner', left_index=True, right_index=True)
inner_join
```

Role		School
Name		
Sally	Course liasion	Engineering
James	Grader	Business

# outer join

how='outer'

119



full outer join

staff\_df

Role	
Name	
Kelly	Director of HR
Sally	Course liasion
James	Grader

student\_df

School	
Name	
James	Business
Mike	Law
Sally	Engineering

- staff\_df 와 student\_df를 인덱스를 기준으로 outer\_join

```
outer_join = pd.merge(staff_df, student_df, how='outer', left_index=True, right_index=True)
outer_join
```

Role		School
Name		
James	Grader	Business
Kelly	Director of HR	NaN
Mike	NaN	Law
Sally	Course liasion	Engineering

# left join

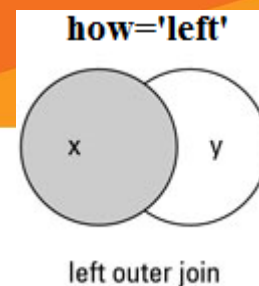
120

staff\_df

Role	
Name	
Kelly	Director of HR
Sally	Course liasion
James	Grader

student\_df

School	
Name	
James	Business
Mike	Law
Sally	Engineering



- staff\_df 와 student\_df를 인덱스를 기준으로 left\_join

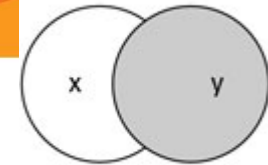
```
left_join = pd.merge(staff_df, student_df, how='left', left_index=True, right_index=True)  
left_join
```

Role		School
Name		
Kelly	Director of HR	NaN
Sally	Course liasion	Engineering
James	Grader	Business

# right join

how='right'

121



right outer join

staff\_df

Role	
Name	
Kelly	Director of HR
Sally	Course liasion
James	Grader

student\_df

School	
Name	
James	Business
Mike	Law
Sally	Engineering

- staff\_df 와 student\_df를 인덱스를 기준으로 right\_join

```
right_join = pd.merge(staff_df, student_df, how='right', left_index=True, right_index=True)
right_join
```

Role		School
Name		
James	Grader	Business
Mike	NaN	Law
Sally	Course liasion	Engineering

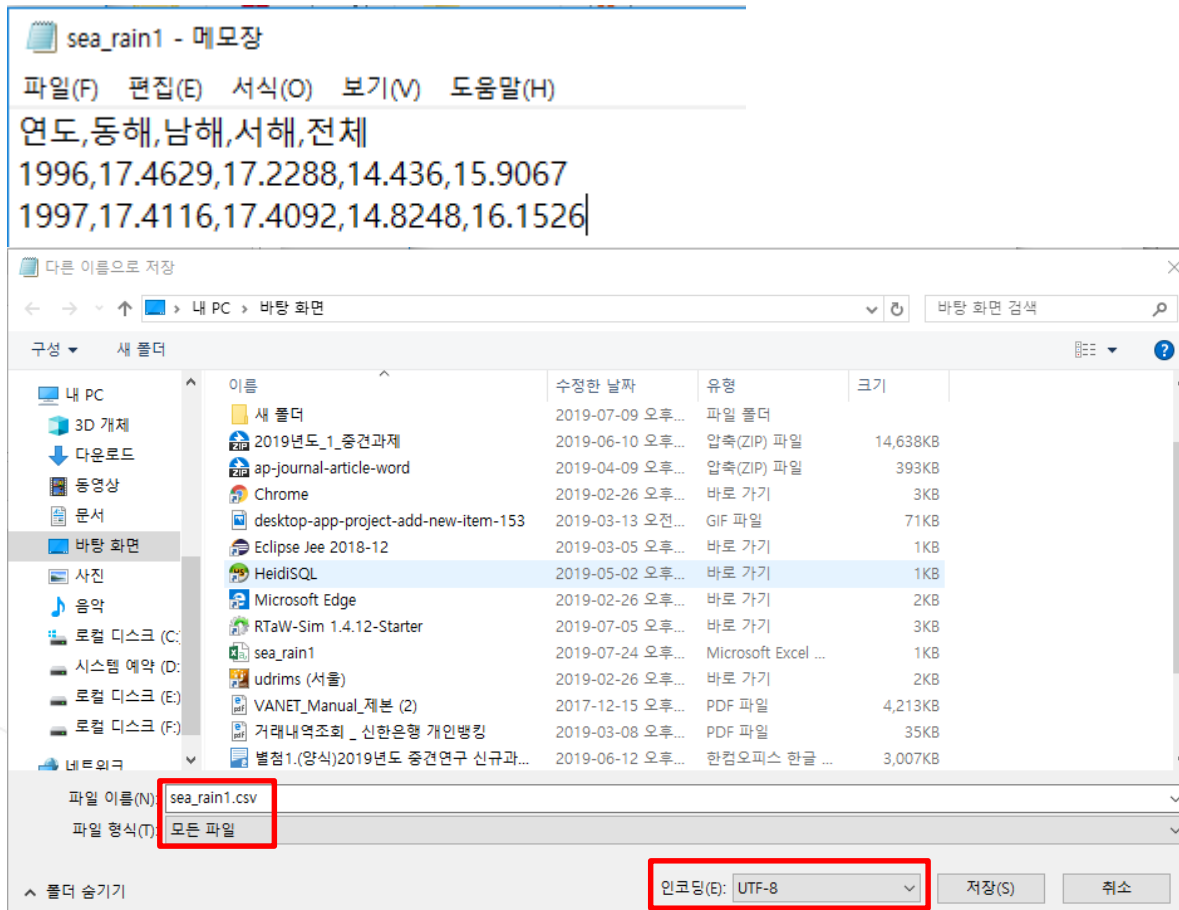
- `read_csv()`

- 기본적으로 각 데이터 필드가 콤마(,)로 구분된 CSV(comma-separated values) 파일을 읽는데 이용
- 옵션을 지정하면 각 데이터 필드가 콤마 외의 구분자로 되어 있어도 데이터를 읽어올 수 있음

`DataFrame_data = pd.read_csv(file_name [, options])`

- `file_name`은 텍스트 파일의 이름으로 경로를 포함할 수도 있음
- `options`는 선택사항





```
In [8]: import pandas as pd
pd.read_csv('C:/Users/jyeon/Desktop/sea_rain1.csv', encoding="utf-8")
```

Out [8]:

	연도	동해	남해	서해	전체
0	1996	17.4629	17.2288	14.4360	15.9067
1	1997	17.4116	17.4092	14.8248	16.1526

encoding = "cp949"  
encoding = "utf-8"



sea\_rain2 - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

연도 동해 남해 서해 전체

1996 17.4629 17.2288 14.436 15.9067

1997 17.4116 17.4092 14.8248 16.1526

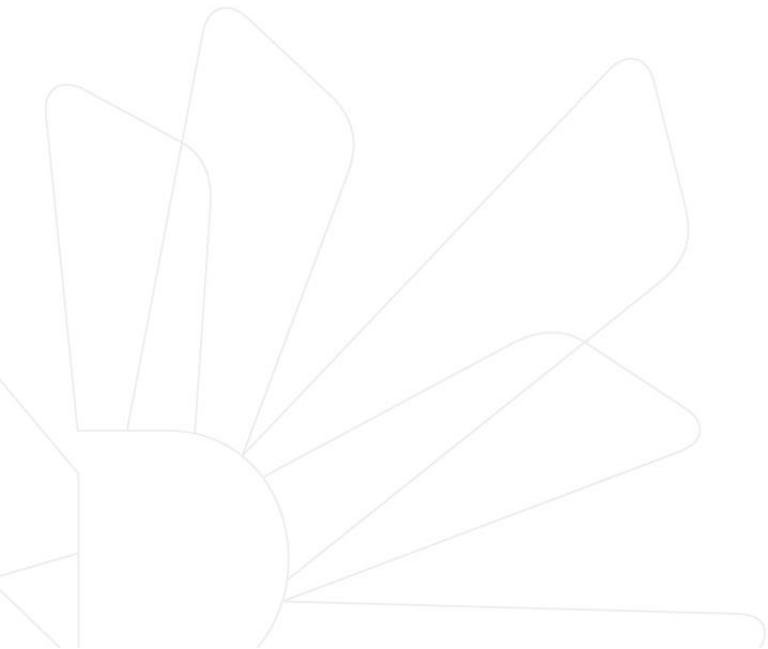
```
In [12]: pd.read_csv('C:/Users/jyeon/Desktop/sea_rain2.csv', encoding="utf-8", sep=" ")
```

Out [12]:

	연도	동해	남해	서해	전체
0	1996	17.4629	17.2288	14.4360	15.9067
1	1997	17.4116	17.4092	14.8248	16.1526



- **DataFrame\_data = pd.to\_csv(file\_name [,options])**
  - file\_name은 텍스트 파일 이름으로 경로를 포함할 수 있음
  - 선택사항인 options에는 구분자와 문자의 인코딩 방식 등을 지정할 수 있는데 지정하지 않으면 구분자는 콤마가 되고 문자의 인코딩 방식은 “utf-8”이 됨





- **panel data**

- pandas에는 Panel이라는 자료구조가 있음
- Panel은 DataFrame의 3차원 버전이라고 이해
- 형식의 데이터를 다루는데 초점을 맞추고 있고 계층적 색인을 이용하면 대개의 경우 N차원 배열은 불필요

