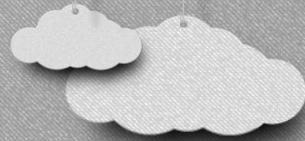


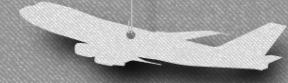


# PYTHON - 데이터분석



# 02

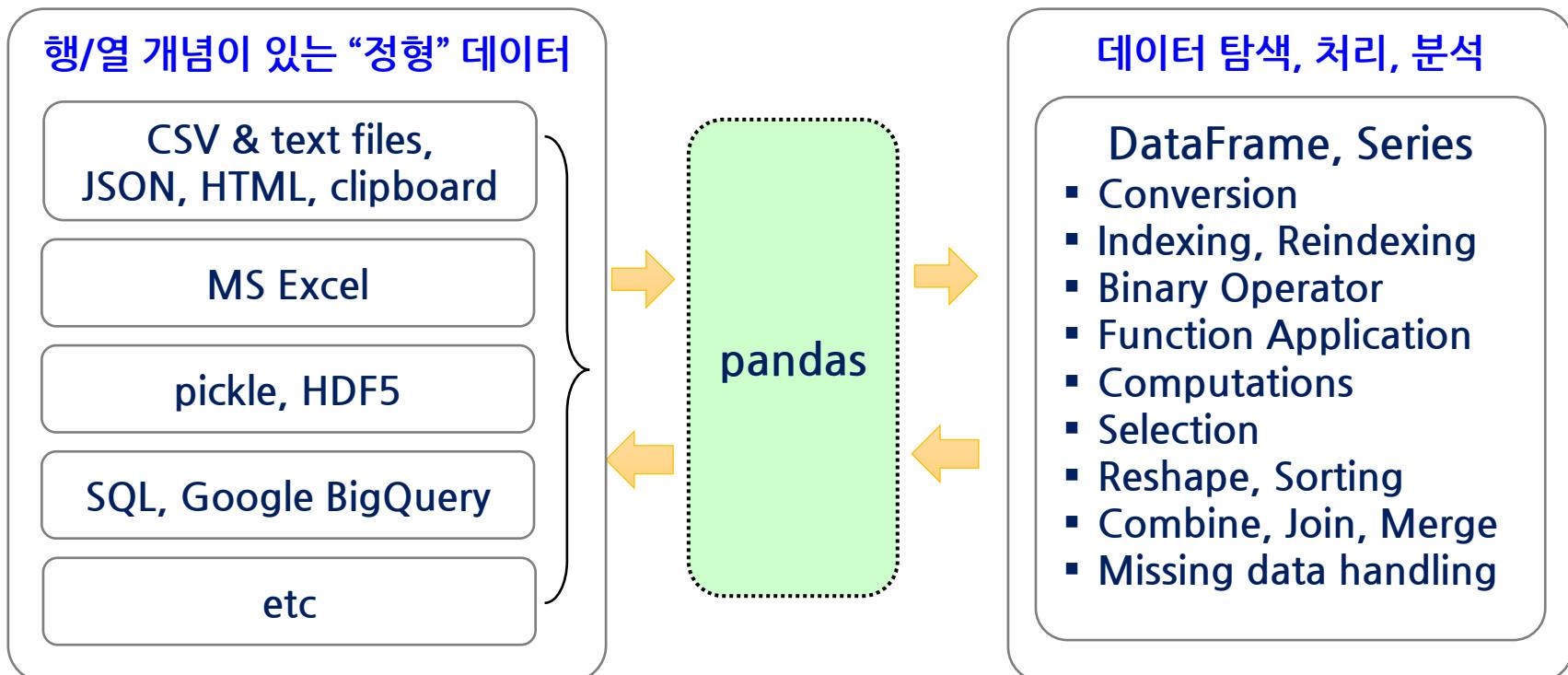
## pandas module



# pandas 특징



- 多样化한 형태의 file, 생성자를 사용하여 DataFrame을 만든다
- DataFrame, Series, Index 등의 객체를 사용하여 데이터 처리 및 분석을 수행 한다
  - 위의 객체들은 ndarray를 기반으로 데이터의 처리, 분석을 효율적으로 할 수 있다



# pandas 설치



- ☞ pandas(powerful Python data analysis toolkit)는 외부 라이브러리이다
- ☞ 표준 파이썬에 포함되지 않으므로 pandas는 import 하여 사용한다
  - pd라는 이름으로 pandas 라이브러리를 가져오라는 명령

```
import pandas as pd
```

- ☞ pandas 모듈이 설치 되지 않았다면 pip 명령으로 설치한다
  - pip install pandas xlrd xlwt openpyxl
- ☞ 설치 시 주의 사항은 다음과 같다
  - Python 3 (3.5이상의 버전) 에서만 사용할 수 있다
  - Numpy1.12.0 이상을 필요로 한다
  - numexpr, bottleneck 은 pandas 연산 성능에 도움 되는 라이브러리이다
    - numexpr 는 2.6.1 이상의 버전, bottleneck 은 1.2.0 이상의 버전을 설치하도록 한다
- ☞ 보다 자세한 설치 관련 안내는 아래의 링크를 참조한다
  - <http://pandas.pydata.org/pandas-docs/stable/install.html>

# pandas API reference



- ▶ pandas API 에 대한 설명은 다음 링크 참조한다
- ▶ <http://pandas.pydata.org/pandas-docs/stable/>

pandas 0.25.3 documentation » [next](#) | [modules](#) | [index](#)

Table Of Contents

- What's New in 0.25.3
- Installation
- Getting started
- User Guide
- Pandas ecosystem
- API reference
- Development
- Release Notes

Search

Enter search terms or a module, class or function name.

**pandas**: powerful Python data analysis toolkit

**Date:** Nov 01, 2019 **Version:** 0.25.3

**Download documentation:** [PDF Version](#) | [Zipped HTML](#)

**Useful links:** [Binary Installers](#) | [Source Repository](#) | [Issues & Ideas](#) | [Q&A](#) [Support](#) | [Mailing List](#)

**pandas** is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the **Python** programming language.

See the [Package overview](#) for more detail about what's in the library.

# 데이터 분석 내용



▶ 데이터를 분석하는 과정은 목적에 따라 다를 수 있지만 다음 작업을 수행한다

상태 분석	데이터의 상태를 파악한다 - df.dtypes, df.columns, df.info, df.describe
index 변경, 정렬	set_index, reset_index, sort_index, sort_value
필요 데이터 추출	indexing 사용 (Basic Indexing, Multi axis Indexing)
Data Cleaning	NA value 데이터 처리
dtype 확인 및 변경	필요시 dtype 변경
데이터 병합	여러 데이터를 한 개로 만들기, pd.concat, pd.merge
그룹별 통계	df.groupby 를 사용한 그룹별 통계 처리
피벗테이블	df.pivot_table을 사용한 행/열 통계 처리
시각화	다양한 그래프를 통한 시각화 처리

# pandas IO tools



## ▣ 다음은 pandas I/O API의 일부이다

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql

[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/io.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) 참조

- Reader 함수 : `pandas.read_XXX()` 와 같이 사용하며, `DataFrame` 객체 반환
- Writer 함수 : `DataFrame.to_XXX()` 와 같이 사용함

# Data Structures



## ☞ pandas의 Data Structure에는 Series와 DataFrame 이 있다

Series

- 1차원의 동일 타입 데이터로 구성된 배열

DataFrame

- 2차원의 테이블 형식의 배열
- Row Index(=index) 와 Column Index(=columns)가 있음
- 여러 개의 Series 로 구성된 Data Table
- 하나의 열은 동일 타입(Homogeneous data)로 구성 함
- 서로 다른 열은 다른 타입의 데이터(Heterogeneous)로 구성 가능 함

## ☞ Series와 DataFrame 의 특징은 다음과 같다

- **iterable** 컨테이너이다 (반복문 사용이 가능함)
- Series는 스칼라 데이터를 위한, DataFrame은 Series를 위한 컨테이너이다
- 데이터 삽입, 삭제, 변경 등 dictionary 와 같은 연산들을 할 수 있다 (dictionarylike)
- **axis=0 대신 'index', axis=1 대신 'columns'** 로 사용 할 수 있다
- index와 columns는 Index 타입이다

# pandas function/method 학습



▣ pandas의 function/method는 매우 다양한 옵션을 제공한다

▣ pandas.read\_csv function의 prototype은 다음과 같다

```
pandas.read_csv(filepath_or_buffer: Union[str, pathlib.Path, IO[~AnyStr]],  
sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None,  
squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None,  
converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None,  
skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False,  
skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False,  
date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None,  
compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar="'",  
quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None,  
dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False,  
low_memory=True, memory_map=False, float_precision=None)
```

1. 처음 학습 시 default 값이 없는 parameter를 사용하는 방법을 익힌다  
- 이를 통해 해당 함수/메서드에 적합하게 구성된 형식의 데이터 처리 작업이 가능하다
2. 예외적인 조건, 확장사용, 정렬, 동작 방법의 세부 설정이 필요한 경우  
다른 parameter 의 사용법을 예제를 통해 익힌다 (확실한 이해 한 뒤 사용하도록 한다)

# 파일 읽어 DataFrame 객체 생성



## 파일을 읽어 DataFrame 객체를 생성하는 메서드를 간단히 살펴보자

메서드	대상
<code>pd.read_csv('XXX.csv')</code>	<ul style="list-style-type: none"><li>텍스트 파일 (열이 콤마(,)로 분리된 텍스트 파일 형식)</li><li>구분자가 콤마가 아닌 경우 <code>sep='구분자'</code> 를 사용함 (공백: '\s+')</li></ul>
<code>pd.read_excel('XXX.xlsx')</code>	<ul style="list-style-type: none"><li>일반 엑셀 파일</li><li><code>xlrd</code>, <code>openpyxl</code> 등의 설치를 통해 엑셀 형식 지원을 해야 함</li></ul>
<code>pd.read_table('XXX.txt')</code>	<ul style="list-style-type: none"><li>텍스트 파일 (열이 탭('\t')으로 분리된 텍스트 파일 형식)</li><li>구분자가 탭이 아닌 경우 <code>sep='구분자'</code>를 사용함</li></ul>

### 주요 parameters

- `header` : 열 이름이 표시된 ‘행 번호’
- 파일을 읽어 DataFrame 객체를 생성할 경우 첫 행을 `header`로 사용함 (`header=0`)
- 파일 내부에 `header`가 없는 경우 :  
`header = None` 지정 후, `names = [목록]`으로 각 열의 이름을 부여함
- `index_col` : `index` 역할을 할 ‘열 번호/열 이름’ 지정,  
지정하지 않으면 일련번호가 사용됨

# CSV 파일 살펴보기



## easySample.csv 를 메모장으로 열어 구조를 살펴 보자

```
ID,pname,birth,dept,english,japanese,chinese  
18030201,James Kim,1990-01-23,Education,1,1,  
18030202,Rose Hwang,1992-10-11,Marketing,,2,  
19030401,Sam Park,1995-07-02,Education,1,,  
19070101,Chris Jang,1990-11-23,Education,,,3  
19070102,Grace Lee,1993-02-01,Marketing,,,  
19070103,Juile Yoon,1992-07-16,Education,,,1
```

## easySample\_woHeader.csv 를 메모장으로 열어 구조를 살펴 보자

```
18030201,James Kim,1990-01-23,Education,1,1, header 없음  
18030202,Rose Hwang,1992-10-11,Marketing,,2,  
19030401,Sam Park,1995-07-02,Education,1,,  
19070101,Chris Jang,1990-11-23,Education,,,3  
19070102,Grace Lee,1993-02-01,Marketing,,,  
19070103,Juile Yoon,1992-07-16,Education,,,1
```

# 파일 읽어 DataFrame 생성하기



- ▶ [예제1] 코드를 실행하여 DataFrame 및 Series의 구조를 확인하라
- ▶ data는 DataFrame, eng는 Series 객체 이룸이다

```
import pandas as pd
if 1:
    data = pd.read_csv('easySample.csv', index_col = 0)
if 0:
    data = pd.read_csv('easySample_woHeader.csv',
                        header = None,
                        names = ['ID', 'name', 'birth', 'dept',
                                 'english', 'japanese', 'chinese'],
                        index_col = 'ID')
eng = data['english']
```

index 지정 (열 번호 사용)

↑

DataFrame

Series

header가 없는 경우

index 지정 (열 이름 사용)

- pd.read\_csv() : csv 파일을 읽어 DataFrame 반환
- data['english'] : DataFrame['열이름']은 Series 반환

# DataFrame의 구조



## ◆ DataFrame의 구조를 살펴보자

<class 'pandas.core.frame.DataFrame'>

	index.name	columns (axis=1)		column name	columns.values		
ID	index.values	pname	birth	dept	english	japanese	chinese
18030201	James Kim	1990-01-23	Education	1.0	1.0	NaN	
18030202	Rose Hwang	1992-10-11	Marketing	NaN	2.0	NaN	
19030401	Sam Park	1995-07-02	Education	1.0	NaN	NaN	
19070101	Chris Jang	1990-11-23	Education	NaN	NaN	3.0	
19070102	Grace Lee	1993-02-01	Marketing	NaN	NaN	NaN	
19070103	Juile Yoon	1992-07-16	Education	NaN	NaN	1.0	

index  
(axis=0)      index label

values

NA values

### CSV 파일

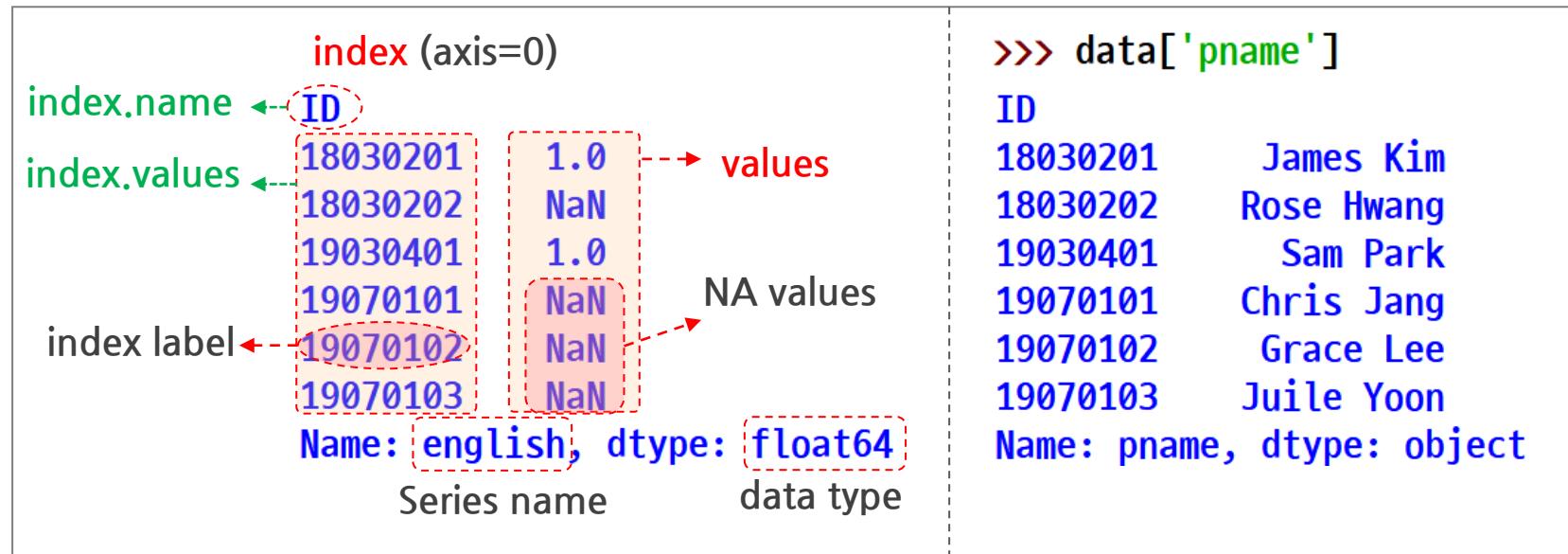
- CSV 파일에는 별도 index 열 없음, 임의로 지정해야함
- index열을 지정하지 않으면, DataFrame에서 일련번호를 갖는 index 열을 추가함
- CSV 파일의 빈 셀은 DataFrame에서 'NaN'(not a number)이 됨

# Series의 구조



## ▶ Series의 구조를 살펴보자

```
<class 'pandas.core.series.Series'>
```



- DataFrame의 `df.columns`, `df.index` 와 Series의 `s.index`는 Index 객체이다
- Index 객체는 반드시 존재하는 구성 요소로 지정하지 않으면 자동으로 생성한다

# DataFrame의 3가지 주요 속성



▣ 다음은 DataFrame의 3가지 주요 속성이다

속성	설명
df.index	<ul style="list-style-type: none"><li>pandas Index 타입</li><li>DataFrame 의 index로 각 row 를 대표하는 값의 목록</li><li>별도 지정 없으면 0부터 1씩 증가하는 값을 갖는 pandas.RangeIndex 사용됨</li><li>생성자의 index parameter 또는 df.index 를 사용하여 index를 지정 가능</li><li>index에 포함된 값에 따라 그 타입이 달라짐</li><li>df.index.name = 'index이름' 으로 이름을 부여할 수 있음</li><li>df.index.values 는 numpy.ndarray 타입의 'index label'의 목록</li></ul>
df.columns	<ul style="list-style-type: none"><li>pandas Index 타입</li><li>DataFrame 의 column label 들의 목록</li><li>df.columns.values 는 numpy.ndarray 타입의 'column name'의 목록</li></ul>
df.values	<ul style="list-style-type: none"><li><b>numpy.ndarray 타입</b></li><li>DataFrame의 index와 columns를 제외한 행, 열의 목록</li></ul>

- columns 또는 index의 이름을 변경할 때는 df.rename 메서드를 사용한다
- <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html>

# DataFrame의 주요 속성



▶ index, columns, values를 구분하여 보자

ID	pname	birth	dept	english	japanese	chinese
18030201	James Kim	1990-01-23	Education	1.0	1.0	NaN
18030202	Rose Hwang	1992-10-11	Marketing	NaN	2.0	NaN
19030401	Sam Park	1995-07-02	Education	1.0	NaN	NaN

속성	결과
df.index	Int64Index([18030201, 18030202, 19030401, 19070101, 19070102, 19070103], dtype='int64', name='ID')
df.columns	Index(['pname', 'birth', 'dept', 'english', 'japanese', 'chinese'], dtype='object') <b>numpy에서 NaN 표기 법</b>
df.values	[['James Kim' '1990-01-23' 'Education' 1.0 1.0 nan] ... ]
df.index.values	[18030201 18030202 19030401 19070101 19070102 19070103]
df.columns.values	['pname' 'birth' 'dept' 'english' 'japanese' 'chinese']

- values 속성은 numpy의 ndarray 타입이며, Read Only 이다 **df.index.name 의 타입은?**
- df.index, df.columns 속성은 pandas의 Index 타입이며, Read/Write 이다

# Series의 주요 속성



▶ Series의 2가지 주요 속성은 index와 values이다

▶ 특성은 DataFrame의 index 및 values와 같다

- 다만, values의 경우 1개의 열에 대한 값들의 목록이다

ID	
18030201	1.0
18030202	NaN
19030401	1.0
19070101	NaN
19070102	NaN
19070103	NaN

Name: english, dtype: float64

각 항의 타입을 적고 R0, RW 속성으로 나누세요

- s.index
- s.values
- s.index.values
- s.index.name

속성	결과
s.index	Int64Index([18030201, 18030202, 19030401, 19070101, 19070102, 19070103], dtype='int64', name='ID')
s.values	[ 1. nan 1. nan nan nan]
s.index.values	[18030201 18030202 19030401 19070101 19070102 19070103]

# DataFrame 및 Series의 주요 속성 확인



- ▶ [예제2] 코드를 실행하여 DataFrame 및 Series의 주요 속성을 확인하라
- ▶ 결과를 예측해 보고 실행하도록 한다

```
def printAttr(w):
    print(type(w), w, sep='\n')
    print("-" * 75)

data = pd.read_csv('easySample.csv', index_col=0)
printAttr(data.head(3))
printAttr(data.index)
printAttr(data.columns)
printAttr(data.values[:3])
printAttr(data.index.values)
printAttr(data.columns.values)

eng = data['english']
printAttr(eng)
printAttr(eng.index)
printAttr(eng.values)
printAttr(eng.index.values)
```

type 함수는 객체의 “완전 클래스 이름”을 반환  
DataFrame의 경우 pandas.core.frame.DataFrame

- df.head(숫자) : DataFrame의 내용 중 처음 부터 숫자 만큼의 행 출력
- df.tail(숫자) : DataFrame의 내용 중 마지막 부터 숫자 만큼의 행 출력
- 숫자를 생략하면 5개의 행이 출력 됨 (숫자의 기본값 = 5)

# Index 객체



## Index 객체의 특징, 연산, 구성 및 종류를 살펴보자

### 특징

- DataFrame과 Series의 index와 columns로 사용될 수 있음
- pandas.Index의 subclass 임
- 빠른 선택과 정렬을 위해 해시 테이블을 사용해 구현됨

### 연산, 구성

- 교집합, 합집합, 대칭 차집합 등의 연산 지원 (set과 같은 연산 제공)
- hashable 데이터로 구성되어야 함
- 데이터에 순서가 있어 정렬 가능
- 데이터 중복이 허용되지만, 데이터가 중복되면 객체 접근이 느려짐

### 종류

- Numeric Index  
→ Int64Index, UInt64Index, Float64Index, RangeIndex
- CategoricalIndex, IntervalIndex, MultiIndex
- DatetimeIndex, TimedeltaIndex, PeriodIndex

DataFrame의 Index가 정렬 되어 있는 것이 selection, indexing 등의 작업 시 편리하다

# RangelIndex 객체



## [예제3] RangelIndex 객체를 생성하여 보자

• `pd.RangelIndex( data [, dtype=np.object, name, ... ] )`

- Int64Index의 **메모리 절약**을 위한 특수 Index 타입
- start부터 stop 까지 step 씩 변하는 값으로 RangelIndex 객체 생성
- Series, DataFrame 에 index가 지정되지 않는 경우의 기본 설정되는 Index 객체

```
v = [80, 90, 95]
a = pd.RangeIndex(3)
b = pd.RangeIndex(5, 8)
c = pd.RangeIndex(10, 31, 10)
s1 = pd.Series(v, a)
s2 = pd.Series(v, b)
s3 = pd.Series(v, c)
```

RangeIndex(start=0, stop=3, step=1)  
RangeIndex(start=5, stop=8, step=1)  
RangeIndex(start=10, stop=31, step=10)

<code>0</code>	80	<code>s1</code>
<code>1</code>	90	
<code>2</code>	95	
		<code>dtype: int64</code>
<code>5</code>	80	<code>s2</code>
<code>6</code>	90	
<code>7</code>	95	
		<code>dtype: int64</code>
<code>10</code>	80	<code>s3</code>
<code>20</code>	90	
<code>30</code>	95	
		<code>dtype: int64</code>

```
d = pd.RangeIndex(10000)
print(a.memory_usage(deep=True), d.memory_usage(deep=True))
```

“실행 결과”를 확인하도록 한다

a, d: 동일한  
메모리양 사용

# Index 객체 확인



▶ [예제4] 다음 코드를 실행하여 다양한 종류의 Index 객체를 확인한다

```
def printObj(*a):
    for df in a:
        print(df.index)
        print(type(df.index))
        print('-'*75)

df1 = pd.read_csv('easySample.csv')
df2 = pd.read_csv('easySample.csv', index_col='ID')
df3 = pd.read_csv('easySample.csv', index_col='pname')
df4 = pd.read_csv('easySample.csv', index_col='birth')
df4.index = pd.to_datetime(df4.index)      --> 타입을 DatetimeIndex 으로 변경
                                                dtype=datetime64[ns]
                                                (numpy의 dtype 타입)
printObj(df1, df2, df3, df4, df1.T)
```

다음 코드의 실행 결과 표시되는 Series의 데이터 타입(dtype)은 무엇일까?

```
birth = df1['birth']
print(birth)
```

# DataFrame, Series, Index의 속성들



## ▶ [예제5] 다음 각 속성을 DataFrame, Series, Index에서 확인한다

속성/기본데이터	적용객체	설명
ndim	D, S, I	▪ 차원을 int로 반환
shape	D, S, I	▪ 차원에 대한 정보를 tuple 형태로 반환
size	D, S, I	▪ 객체의 요소 개수를 int로 반환
memory_usage( index=True, <b>deep=False</b> ) <b>dtype=object</b> 는 deep=True 사용 필요	D, S, I	▪ 각 열의 메모리 사용 정보를 byte 단위로 반환 ▪ 1개 열인 경우 해당 객체의 메모리 사용 정보 반환 ▪ index=True가 기본, index 열 포함 여부를 나타냄
T	D	▪ 행/열 전환
dtypes	D, S	▪ 각 열 별 데이터 타입 정보 (데이터 사용 전 확인 필수)
name	S, I	▪ 이름 정보, MultiIndex의 경우 None 이 반환됨
names	I	▪ 이름 정보를 list 형태로 반환
dtype	S, I	▪ values에 대한 데이터 타입
nbytes	S, I	▪ 기본데이터의 메모리 사용 정보를 byte 단위로 반환

D : DataFrame, S : Series, I : Index 객체를 의미함

# pandas의 데이터 형식



▣ 다음은 dtype 지정에 사용되는 pandas의 데이터 형식이다

문자열 이름	numpy/pandas 객체	설명
bool	np.bool	▪ 단일 바이트 저장
int	np.int32, np.int64, np.uint	▪ 기본 64비트, unsigned 도 사용 가능
float	np.float, np.float32, np.float64	▪ 기본 64비트
complex	np.complex	▪ 복소수는 거의 사용되지 않음
0, object	np.object	▪ str, tuple, list, dict 등 ▪ 여러 데이터를 포함한 객체
datetime64	np.datetime64, pd.Timestamp	▪ 나노초 단위의 정밀도를 가진 특정 시각
timedelta64	np.timedelta64, pd.Timedelta	▪ 1부터 나노초 단위의 시간 간격
category	pd.Categorical	▪ pandas에만 있음 ▪ 고유 값이 별로 없는 열 객체에 유용

# 데이터 타입(dtype)의 메모리 사용



## [예제6] 데이터 타입(dtype) 변경 전, 후의 메모리 사용을 비교하여 보자

```
data = pd.read_csv('easySample.csv', index_col='ID')
print(data.dtypes, end='\n\n')
print(data.memory_usage(index=False, deep=True), end='\n\n')
```

1

```
data['dept'] = data['dept'].astype('category')
data['birth'] = pd.to_datetime(data['birth'])
print(data.dtypes, end='\n\n')
print(data.memory_usage(index=False, deep=True))
```

→ dtype 변경 메서드/함수

2

easySample.csv에 포함된  
데이터 개수는 6개!!

pname	object	pname	230	1
birth	object	birth	234	
dept	object	dept	228	

pname	object	pname	230	2
birth	datetime64[ns]	birth	48	
dept	category	dept	130	

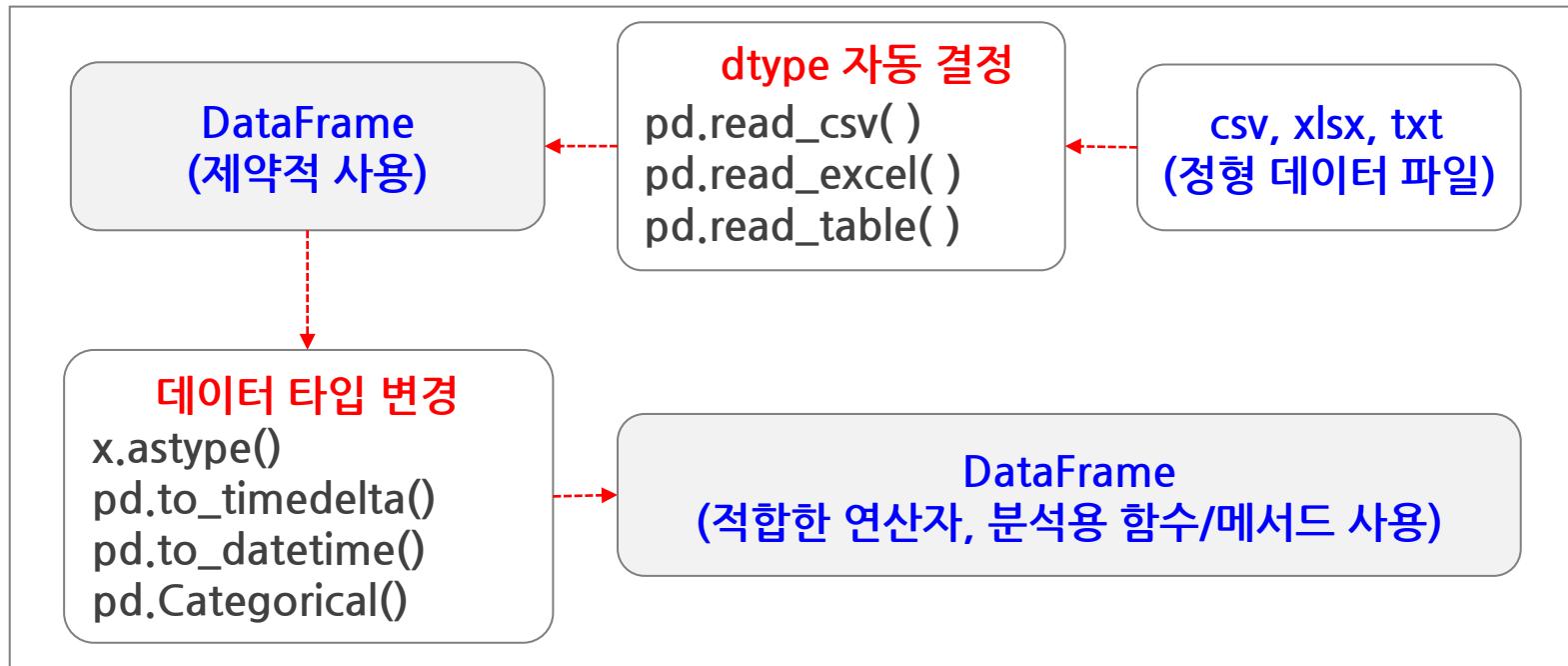
데이터가 많은 경우의 차이는??  
다음 파일로 차이를 확인하여 보자  
easySampleLong.csv → 1000개

- Series.astype(), pandas.to\_datetime() : dtype을 변경하는 메서드/함수
- 메모리 사용을 비교하기 위해 사용한 것으로 학습은 별도 진행 됨

# 데이터 타입(dtype) 변경의 필요성



- 적합한 기능 사용을 위해 잘못 지정된 dtype을 바르게 변경해야 한다
- 파일을 읽어 DataFrame 생성시 column 별로 dtype이 자동 결정된다
  - 대부분의 경우 숫자는 int 또는 float, 나머지는 object로 결정된다



데이터 타입 변경 전에 데이터 조작이 필요할 수 있음 (예) 공백 제거, 콤마 제거, 단위 변환

# 데이터 타입 변경 메서드

## ▶ [예제7] 데이터 타입 변경 메서드 및 함수를 알아보자

• `x.astype( dtype [, copy=True, ...])`

• x의 데이터 타입을 dtype으로 주어진 타입으로 변경하여 반환

x ▪ DataFrame, Series, Index 객체

dtype ▪ data type, or dict of column name & data type  
▪ **numpy.dtype, Python type 사용**  
▪ 전체를 같은 타입으로 변경 : 1개의 data type 지정  
▪ DataFrame의 열에 대해 별도 타입 변경 : {column-label : dtype, ...}로 지정

```
s = pd.Series([1, 0])
a = s.astype(np.bool)
```

0 1	0	True
1 0	1	False
dtype: int64		

```
df = pd.DataFrame([[1, 2, 3]] * 2)
b = df.astype(np.float32)
c = df.astype({0 : np.float32, 2 : np.bool})
```

0 1 2	0 1 2	0 1 2
0 1 2 3	0 1.0 2.0 3.0	0 1.0 2
1 1 2 3	1 1.0 2.0 3.0	1 1.0 2 True

# datetime 타입



## ▶ [예제8] datetime 타입 변경 (dtype=datetime64[ns])

- pd.to\_datetime( arg [... , unit=None, ... ] ) # parameters - 하단 링크 참조
  - argument에 따라 datetime 관련 객체를 생성하거나 데이터 타입을 전환하여 반환

arg

- integer, float, string, datetime, list, tuple, 1D array, Series, DataFrame
- list-like → DatetimeIndex 객체, scalar → Timestamp 객체
- Series, DataFrame → Series 객체 (dtype : np.datetime64)

```
s = pd.Series(['2019-11-01', '2019-11-02'])
df = pd.DataFrame({'year' : [2020, 2020],
                   'month': [1, 1],
                   'day'  : [1, 2]})

a = pd.to_datetime(s)
b = pd.to_datetime(df)
```

Series

```
0 2019-11-01
1 2019-11-02
dtype: datetime64[ns]

0 2020-01-01
1 2020-01-02
dtype: datetime64[ns]
```

[https://pandas.pydata.org/pandas-docs/version/0.25/reference/api/pandas.to\\_datetime.html](https://pandas.pydata.org/pandas-docs/version/0.25/reference/api/pandas.to_datetime.html)

# timedelta 타입



## ▶ [예제9] timedelta 타입 변경 (dtype=timedelta64[ns])

- `pd.to_timedelta( arg [, unit='ns', ... ] ) # parameters - 하단 링크 참조`
  - argument에 따라 timedelta 관련 객체를 생성하거나 데이터 타입을 변환하여 반환

arg

- str, timedelta, list-like, Series
- Timedelta 객체 또는 TimedeltaIndex 객체 (dtype : np.timedelta64)

```
a = pd.to_timedelta('12:30:05.0123')
b = pd.to_timedelta(['1 days 1:2:3.0123',
                     '15.5us', 'nan'])
s = pd.Series(np.arange(1, 3))
c = pd.to_timedelta(s, unit='h')
```

Timedelta

```
0 days 12:30:05.012300
```

Series

```
0    01:00:00
1    02:00:00
```

```
dtype: timedelta64[ns]
```

TimedeltaIndex

```
TimedeltaIndex(['1 days 01:02:03.012300', '0 days 00:00:00.000015', NaT],
                dtype='timedelta64[ns]', freq=None)
```

[https://pandas.pydata.org/pandas-docs/version/0.25/reference/api/pandas.to\\_timedelta.html](https://pandas.pydata.org/pandas-docs/version/0.25/reference/api/pandas.to_timedelta.html)

# Series 객체 생성



## ▶ [예제10] 다음은 Series 객체를 생성하는 방법이다

```
pd.Series(data=None, index=None, dtype=None, name=None, copy=False, ... )
```

data	<ul style="list-style-type: none"><li>▪ array-like, iterable, dict or scalar value 등을 사용할 수 있음</li></ul>
index	<ul style="list-style-type: none"><li>▪ array-like or Index (1D)</li><li>▪ values 는 hashable 객체이어야 하며, data와 같은 길이어야 함</li><li>▪ 생략 시 RangeIndex가 사용됨 (0부터 1씩 증가하는 숫자 values)</li><li>▪ data가 dict 객체일 때 index가 생략되면, dict 객체의 key를 index로 사용</li><li>▪ dict 객체의 key와 index가 중복되면 index를 따름 (dict의 key가 갱신 됨)</li></ul>
dtype	<ul style="list-style-type: none"><li>▪ Series.values 의 데이터 타입</li><li>▪ str, numpy.dtype, ExtensionDtype 등을 사용할 수 있음</li></ul>
name	<ul style="list-style-type: none"><li>▪ Series.name으로 사용할 것으로 문자열로 지정함</li></ul>
copy	<ul style="list-style-type: none"><li>▪ 입력 데이터를 복사하는지에 대한 여부 (ndarray에만 적용)</li><li>▪ True인 경우 복사(다른 메모리 사용), False라도 dtype이 다르면 복사함</li></ul>

# Series 객체 생성 예



[예제11]

```
score = (80, 90, 100, 95)
name = ["Kim", "Yoon", "Choi", "Park"]
mydata = { k : v for k, v in zip(name, score) }
name_p = ["Song", "Kim", "Lee", "Choi", "Park"]
```

pd.Series(score)	pd.Series(name)	pd.Series(mydata, dtype=np.int32)
0 80	0 Kim	dict 의 key
1 90	1 Yoon	Kim 80
2 100	2 Choi	Yoon 90
3 95	3 Park	Choi 100
dtype: int64	dtype: object	Park 95
		dtype: int32

pd.Series(score, index=name, name="score")	pd.Series(mydata, index=name_p)
Kim 80 Yoon 90 Choi 100 Park 95 Name: score, dtype: int64	index 기준 Song NaN Kim 80.0 Lee NaN Choi 100.0 Park 95.0 dtype: float64

# Series의 메서드



## ▶ [예제12] 다음은 Series의 주요 메서드이다

메서드	설명
s.sum()	▪ <b>Nan</b> 을 제외한 데이터 <b>합계</b> 구하기, float 반환
s.count()	▪ <b>Nan</b> 을 제외한 데이터 <b>개수</b> 구하기, int 반환
s.mean()	▪ <b>Nan</b> 을 제외한 데이터 <b>평균</b> 구하기, float 반환
s.unique()	▪ <b>중복 데이터를 제외한</b> 데이터의 ndarray로 반환 (NaN 포함)
s.value_counts()	▪ <b>Nan</b> 을 제외한 각 데이터의 개수(정수)의 Series 반환
s.head(숫자), s.tail(숫자)	▪ 데이터를 상위/하위 숫자 개 만큼의 Series 반환
s.to_list()	▪ s.values를 list 객체로 반환
s.to_numpy([dtype, copy])	▪ s.values를 ndarray 객체로 반환

pandas 의 함수/메서드 들은 일반적으로 **Nan**에 대해 제외하고 처리한다 (numpy 와 다름)

```
<class 'numpy.ndarray'>
```

```
[ 4.  2.  2. nan nan  6.  7.  6.  7.]
```

```
np.unique() [ 2.  4.  6.  7. nan nan]
```

```
s.unique() [ 4.  2. nan  6.  7.]
```

# Series의 메서드 사용 예



## [예제12]

```
arr = np.array([4, 2, 2, np.nan, np.nan, 6, 7, 6, 7])
s = pd.Series(arr)
```

s.sum()

```
<class 'numpy.float64'>
34.0
```

s.count()

```
<class 'numpy.int32'>
7
```

s.unique()

```
<class 'numpy.ndarray'>
[ 4.  2. nan  6.  7.]
```

s.mean()

```
<class 'numpy.float64'>
4.857142857142857
```

s.value\_counts()

```
<class 'pandas.core.series.Series'>
7.0    2
6.0    2
2.0    2
4.0    1
dtype: int64
```

```
>>> 34 / 7
4.857142857142857
```

s.sum()/s.count()

```
<class 'float'>
4.857142857142857
```

arr의 dtype을 np.float32로 지정하여 결과의 차이를 확인하도록 한다

```
<class 'float'>
4.857142925262451
```

# Series의 연산



▶ [예제13] 다음은 Series의 연산의 특징이다

▶ “Series 와 Series” 의 연산은 같은 index의 value 끼리 연산 된다

- 같은 index가 없는 경우, 추가되며 결과는 NaN이다

▶ “Series 와 스칼라”의 연산은 각 원소별로 스칼라와 연산 된다

- 스칼라가 broadcasting 되어 사용된다

Series와 Series 연산

A	1
B	2
C	3
D	4
E	5

+

A	7.0
B	NaN
C	NaN
D	13.0
E	NaN
X	NaN
Y	NaN

Series와 스칼라 연산

A	1
B	2
C	3
D	4
E	5

+

A	3
B	4
C	5
D	6
E	7

s1.add(s2, fill\_value=0) 을 실행하여 결과를 비교하자

# DataFrame 객체 생성



## ▶ [예제14] 다음은 DataFrame 객체를 생성하는 방법이다

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

data	<ul style="list-style-type: none"><li>▪ ndarray(structured or homogeneous), Iterable, dict, DataFrame</li><li>▪ dict 사용시 columns가 지정되지 않으면 dict의 key가 columns로 사용됨</li><li>▪ DataFrame 사용시 data, index, columns 가 복사 됨</li></ul>
index columns	<ul style="list-style-type: none"><li>▪ array-like or Index (1D)</li><li>▪ 생략 시 RangeIndex가 사용됨 (0 부터 1씩 증가하는 숫자 values)</li><li>▪ index.values 는 hashable 객체이어야 함</li><li>▪ data로 dict 사용시, dict 의 key 와 columns가 중복될 경우 columns를 따름</li><li>▪ data로 ndarray 사용시 index.values 개수 == data 행 개수</li><li>▪ data로 ndarray 사용시 columns.values 개수 == data 열 개수</li></ul>
dtype	<ul style="list-style-type: none"><li>▪ 직접 데이터 타입을 지정하며, 1개의 타입 지정만 가능함</li></ul>
copy	<ul style="list-style-type: none"><li>▪ ndarray(2D), DataFrame을 data로 사용시 copy=False는 연결을 의미함</li><li>▪ ndarray의 dtype이나, DataFrame의 index, columns 등이 변경이 되면 copy=True로 동작함</li></ul>

# DataFrame 객체 생성



## [예제14]

```
ID = [1900101, 1900102, 1900103, 1900104]
name = ["Kim", "Yoon", "Choi", "Park"]
data = {'name' : name,
        'english' : [80, 90, 100, 95],
        'chinese' : [100, 80, 70, 85],
        'korean' : [95, 100, 80, 60]}
```

pd.DataFrame(data)

	name	english	chinese	korean
0	Kim	80	100	95
1	Yoon	90	80	100
2	Choi	100	70	80
3	Park	95	85	60

```
df = pd.DataFrame(data, index = ID,
                   columns = ['name', 'english'])
df.index.name = 'ID'
```

name english

ID	name	english
1900101	Kim	80
1900102	Yoon	90
1900103	Choi	100
1900104	Park	95

# DataFrame 연산 1/2



## ▶ [예제15] DataFrame과 DataFrame 사이의 연산을 알아보자

◀ DataFrame 끼리의 연산은 index와 column을 모두 대상으로 한다

- `df + df`의 경우 match 되는 index나 column이 없는 경우 NaN으로 결과가 표시된다

<table border="1"><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>A</td><td>1</td><td>1</td></tr><tr><td>B</td><td>1</td><td>1</td></tr></table>	a	b	c	A	1	1	B	1	1	+	<table border="1"><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>A</td><td>2</td><td>2</td></tr><tr><td>B</td><td>2</td><td>2</td></tr></table>	a	b	c	A	2	2	B	2	2	=	<table border="1"><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>A</td><td>3</td><td>3</td></tr><tr><td>B</td><td>3</td><td>3</td></tr></table>	a	b	c	A	3	3	B	3	3
a	b	c																													
A	1	1																													
B	1	1																													
a	b	c																													
A	2	2																													
B	2	2																													
a	b	c																													
A	3	3																													
B	3	3																													

<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr><tr><td>C</td><td>1</td><td>1</td><td>1</td></tr><tr><td>B</td><td>1</td><td>1</td><td>1</td></tr></table>	a	b	c	d	C	1	1	1	B	1	1	1	+	<table border="1"><tr><td>a</td><td>c</td><td>d</td></tr><tr><td>A</td><td>2</td><td>2</td></tr><tr><td>B</td><td>2</td><td>2</td></tr></table>	a	c	d	A	2	2	B	2	2	=	<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr><tr><td>A</td><td>NaN</td><td>NaN</td><td>NaN</td></tr><tr><td>B</td><td>3.0</td><td>NaN</td><td>3.0</td></tr><tr><td>C</td><td>NaN</td><td>NaN</td><td>NaN</td></tr></table>	a	b	c	d	A	NaN	NaN	NaN	B	3.0	NaN	3.0	C	NaN	NaN	NaN
a	b	c	d																																						
C	1	1	1																																						
B	1	1	1																																						
a	c	d																																							
A	2	2																																							
B	2	2																																							
a	b	c	d																																						
A	NaN	NaN	NaN																																						
B	3.0	NaN	3.0																																						
C	NaN	NaN	NaN																																						

A, C 행, b 열 : NaN

- `df.add(df, fill_value=0)`을 사용했을 때의 결과는 다음과 같다

<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr><tr><td>C</td><td>1</td><td>1</td><td>1</td></tr><tr><td>B</td><td>1</td><td>1</td><td>1</td></tr></table>	a	b	c	d	C	1	1	1	B	1	1	1	+	<table border="1"><tr><td>a</td><td>c</td><td>d</td></tr><tr><td>A</td><td>2</td><td>2</td></tr><tr><td>B</td><td>2</td><td>2</td></tr></table>	a	c	d	A	2	2	B	2	2	=	<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr><tr><td>A</td><td>2.0</td><td>NaN</td><td>2.0</td></tr><tr><td>B</td><td>3.0</td><td>1.0</td><td>3.0</td></tr><tr><td>C</td><td>1.0</td><td>1.0</td><td>1.0</td></tr></table>	a	b	c	d	A	2.0	NaN	2.0	B	3.0	1.0	3.0	C	1.0	1.0	1.0
a	b	c	d																																						
C	1	1	1																																						
B	1	1	1																																						
a	c	d																																							
A	2	2																																							
B	2	2																																							
a	b	c	d																																						
A	2.0	NaN	2.0																																						
B	3.0	1.0	3.0																																						
C	1.0	1.0	1.0																																						

A 행, b 열 둘 다 없음

# DataFrame 연산 2/2



## ▶ [예제15] DataFrame과 Scalar, Series의 사이의 연산을 알아보자

◀ DataFrame + Scalar는 각 원소별로 스칼라와 연산 된다

- 스칼라가 **broadcasting** 되어 사용된다

$$\begin{array}{c|ccc} & a & b & c \\ \hline A & 2 & 2 & 2 \\ B & 2 & 2 & 2 \end{array} + \textcolor{red}{(2)} = \begin{array}{c|ccc} & a & b & c \\ \hline A & 4 & 4 & 4 \\ B & 4 & 4 & 4 \end{array}$$

◀ DataFrame + Series은 DataFrame의 column, Series의 index에 맞춰 연산 된다

- Series가 DataFrame의 row 개수에 맞춰 broadcasting 되어 사용된다
- match 되는 index 가 없는 경우 결과는 NaN 이다

$$\begin{array}{c|ccc} & a & b & c \\ \hline A & 2 & 2 & 2 \\ B & 2 & 2 & 2 \end{array} + \begin{array}{c|c} a & 1 \\ b & 1 \\ c & 1 \end{array} = \begin{array}{c|ccc} & a & b & c \\ \hline A & 3 & 3 & 3 \\ B & 3 & 3 & 3 \end{array}$$
  
$$\begin{array}{c|ccc} & a & b & \textcolor{red}{c} \\ \hline A & 2 & 2 & 2 \\ B & 2 & 2 & 2 \end{array} + \begin{array}{c|c} a & 3 \\ b & 3 \\ \textcolor{red}{x} & 3 \end{array} = \begin{array}{c|cccc} & a & b & c & x \\ \hline A & 5.0 & 5.0 & \text{NaN} & \text{NaN} \\ B & 5.0 & 5.0 & \text{NaN} & \text{NaN} \end{array}$$

c, x 열 : NaN

# DataFrame의 열, Series의 행 - 갱신, 추가, 제거



▣ 다음은 DataFrame 의 columns, Series의 index 조작 방법이다

조작 방법	설명
<code>df['column_label']=1D array</code> <code>df.column_label = 1D array</code>	<ul style="list-style-type: none"><li>'column_label'에 해당하는 column 갱신 또는 추가</li><li>항의 개수가 같아야 하며 df.column_label 은 갱신만 가능</li></ul>
<code>del df['column_label']</code>	<ul style="list-style-type: none"><li>'column_label'에 해당하는 column 삭제</li></ul>
<code>columns = 1D array</code>	<ul style="list-style-type: none"><li>단순히 이름만 변경하는 것으로 <b>values가 바뀌는 것이 아님</b></li><li>'column_label' 목록을 열의 개수에 맞춰 1D array로 지정함</li></ul>
<code>s['index_label'] = value</code> <code>s.index_label = value</code>	<ul style="list-style-type: none"><li>'index_label'에 해당하는 요소 갱신 또는 추가</li><li>s.index_label 는 갱신만 가능</li></ul>
<code>del s['index_label']</code>	<ul style="list-style-type: none"><li>'index_label'에 해당하는 요소 삭제</li></ul>

- column\_label, index\_label 과 객체의 메서드 이름이 겹치지 않는 것이 좋음 (예 : sum)
  - df.column\_label, s.index\_label 의 표현은 좌변에 사용하지 않는 것이 좋음
- 
- df.column\_label 을 추가 용도로 사용하면 UserWarning 이 발생함
  - UserWarning은 프로그램 실행이 중단되는 것은 아님

# Series, DataFrame의 label 사용 (indexing)



## ▶ [예제16] label을 사용한 코드의 결과를 예측하라

```
ID = [1900101, 1900102, 1900103, 1900104]
name = ["Kim", "Yoon", "Choi", "Park"]
data = {'name' : name,
        'english' : [80, 90, 100, 95],
        'korean' : [95, 100, 80, 60]}
```

### 사용 label

- Series : index\_label,
- DataFrame : column\_label

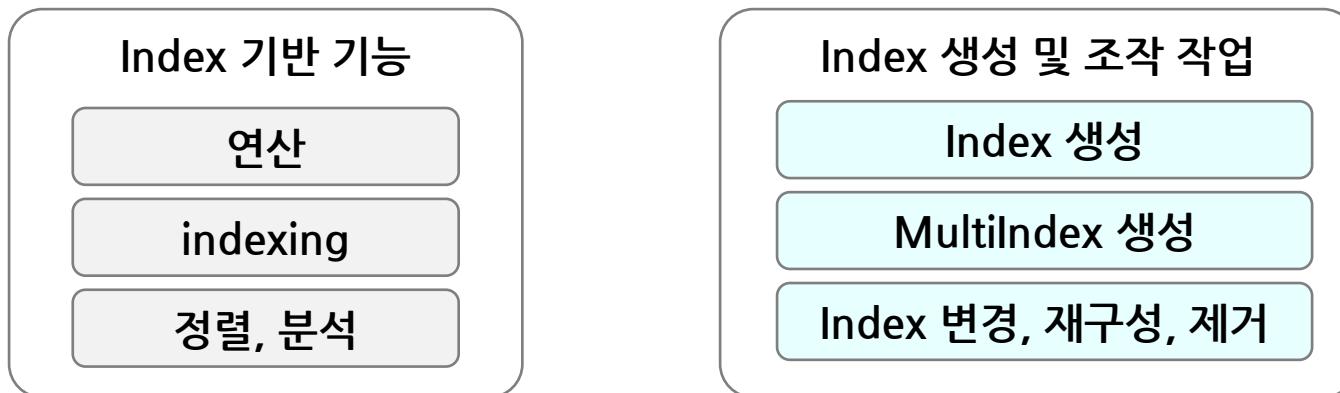
```
s = pd.Series(ID, index=name)
s['Yoon'] = 1900100
s['Lee'] = 1900200
printobj(s, s['Park'])
```

```
df = pd.DataFrame(data, index=ID)
eng1 = df['english']
eng2 = df.english
df['korean'] = [100] * 4
#df.chinese = [100, 80, 70, 85]
df['chinese'] = [100, 80, 70, 85]
printobj(id(eng1) == id(eng2), df)
```

# Index의 구조 및 용도



- ◆ pandas의 연산, indexing, 정렬 및 분석 시 Index를 사용할 수 있다
- ◆ index를 생성, 변경, 제거, 재구성 하는 작업이 필요할 수 있다



The diagram illustrates the structure of a Pandas DataFrame with annotations:

index.name	columns (axis=1)	column name	columns.values
index.values	pname birth	(dept) english	japanese chinese
ID	James Kim 1990-01-23	Education	1.0 1.0 NaN
18030201	Rose Hwang 1992-10-11	Marketing	NaN 2.0 NaN
18030202	Sam Park 1995-07-02	Education	1.0 NaN NaN
19030401			

Annotations:

- index (axis=0)**: Points to the row index column.
- index label**: Points to the value "19030401" in the index column.
- index.name**: Points to the column "index.name".
- index.values**: Points to the column "index.values".
- columns (axis=1)**: Points to the column "columns (axis=1)".
- pname**: Points to the column "pname".
- birth**: Points to the column "birth".
- column name**: Points to the column "column name".
- (dept)**: Points to the column "(dept)".
- english**: Points to the column "english".
- columns.values**: Points to the column "columns.values".
- japanese**: Points to the column "japanese".
- chinese**: Points to the column "chinese".

# index 변경 -1/3



## [예제17] set\_index 메서드를 사용한 index 변경을 알아보자

df.set\_index(keys, drop=True, append=False, inplace=False)

선택된 columns의 일부로 새로 구성한 index 를 갖는 DataFrame 반환

keys ▪ 새로운 index 생성에 사용할 column/column 목록(columns labels로 작성)

drop ▪ keys를 columns 로 부터 제거 여부 (default True → 제거함)

append ▪ 기존 index를 유지하면서 keys를 추가 여부 (default False → 유지하지 않음)

Index! ← dept gender age salary				
pname	dept	gender	age	salary
James Kim	Education	Female	36	4700
Rose Hwang	Marketing	Male	35	4320

df1 = df.set\_index('age')

dept	gender	age	salary
36	Education	Female	36
35	Marketing	Male	35

df2 = df.set\_index('age', drop=False)

dept	gender	salary
James Kim	Female	4700
Rose Hwang	Male	4320

df1 = df.set\_index('age', drop=True, append=False)

pname	age
James Kim	36
Rose Hwang	35

df3 = df.set\_index('age', append=True)

dept	gender	salary
Education	Female	4700
Marketing	Male	4320

# index 변경 -2/3



## ▶ [예제17] reset\_index 메서드를 사용한 index 변경을 알아보자

- `X.reset_index( level=None, drop=False, ... ) : # X : DataFrame, Series`
- 현 index를 columns에 포함하여 새로 구성한 index를 갖는 DataFrame 또는 Series 반환

level	▪ int, str, tuple or list로 작성 (None이면 모든 index를 대상으로 함) ▪ 현 index 중 columns로 포함 할 대상을 번호 또는 name으로 지정함 ▪ 모든 index가 columns로 포함되면 RangeIndex가 새로운 Index로 생성됨
drop	▪ 현 index를 columns로 포함하지 않을 지의 여부 (drop=True : 포함 안함)

index	pname	dept	gender	age	salary	columns
	James	Kim	Education	Female	36	4700
	Rose	Hwang	Marketing	Male	35	4320

columns

```
df1 = df.reset_index(0)
df2 = df.reset_index('pname')
```

dept	gender	pname	age	salary
Education	Female	James	Kim	36
Marketing	Male	Rose	Hwang	35

# index 변경 -3/3



▶ [예제17] reset\_index 메서드를 사용한 index 변경을 알아보자

			age	salary
pname	dept	gender		
James Kim	Education	Female	36	4700
Rose Hwang	Marketing	Male	35	4320

```
df3 = df.reset_index()
```

	pname	dept	gender	age	salary
0	James Kim	Education	Female	36	4700
1	Rose Hwang	Marketing	Male	35	4320

모든 index가 columns로 이동

```
df4 = df.reset_index(['pname', 'dept'])
```

```
df5 = df.reset_index([1,2], drop=True) dept, gender 를 제거함
```

gender	pname	dept	age	salary	age	salary
Female	James Kim	Education	36	4700	James Kim	36
Male	Rose Hwang	Marketing	35	4320	Rose Hwang	35

pname, dept가 columns로 이동

# DataFrame, Series 정렬 - index 기준



▶ [예제18] 축에 따른 index labels를 기준으로 대상을 정렬하는 메서드이다

```
X.sort_index(axis=0, level=None, ascending=True, inplace=False,  
kind='quicksort', na_position='last', ...)
```

axis	▪ 정렬 축 지정, DataFrame은 0, 1 Series은 0을 사용할 수 있음
level	▪ int, level-name, list of int or level-names ▪ MultiIndex에서 정렬 기준으로 사용할 level을 지정 함
ascending	▪ True: 오름차순 정렬, False: 내림차순 정렬, list of boolean
inplace	▪ True인 경우 객체를 직접 수정하고 None 반환
kind	▪ 정렬 알고리즘 지정 {'quicksort', 'mergesort', 'heapsort'}
na_position	▪ NA value의 위치, {'first', 'last'}

# DataFrame, Series 정렬 - index 기준



▶ [예제18] 축에 따른 index labels를 기준으로 대상을 정렬하는 메서드이다

ID	pname	birth	dept
18030201	James Kim	1990-01-23	Education
18030202	Rose Hwang	1992-10-11	Marketing
19030401	Sam Park	1995-07-02	Education
19070101	Chris Jang	1990-11-23	Education
19070102	Grace Lee	1993-02-01	Marketing

```
df1 = df.sort_index(ascending=False)
```

```
df2 = df.sort_index(axis=1)
```

ID	pname	birth	dept	birth	dept	pname
19070102	Grace Lee	1993-02-01	Marketing	18030201	1990-01-23	Education James Kim
19070101	Chris Jang	1990-11-23	Education	18030202	1992-10-11	Marketing Rose Hwang
19030401	Sam Park	1995-07-02	Education	19030401	1995-07-02	Education Sam Park
18030202	Rose Hwang	1992-10-11	Marketing	19070101	1990-11-23	Education Chris Jang
18030201	James Kim	1990-01-23	Education	19070102	1993-02-01	Marketing Grace Lee

# DataFrame, Series 정렬 - values 기준



▶ [예제19] 축에 따른 by 목록의 values를 기준으로 대상을 정렬하여 반환한다

```
X.sort_values(by, axis=0, ascending=True, inplace=False,  
kind='quicksort', na_position='last')
```

by

- str 또는 list of str을 사용하여 정렬 기준이 되는 이름 또는 이름 목록 지정
- axis=0 : column labels, axis=1 : index labels를 사용하여 이름 목록 작성
- axis=1 사용을 위해서는 모든 columns의 dtype이 동일해야 함(거의 사용 안함)

```
df1 = dfna.sort_values('pname')
```

ID	pname	english
19070101	Chris Jang	NaN
19070102	Grace Lee	NaN
18030202	Rose Hwang	
18030201	James Kim	1
19030401	Sam Park	1

```
df2 = dfna.sort_values(['english', 'pname'],  
ascending=[True, False])
```

ID	pname	english	chinese
18030202	Rose Hwang		NaN
19030401	Sam Park	1	NaN
18030201	James Kim	1	
19070102	Grace Lee	NaN	NaN
19070101	Chris Jang	NaN	3

# Indexing and selection data



- ◆ DataFrame, Series 등에서 원하는 데이터를 추출/갱신하기 위해 사용한다
- ◆ indexing의 종류

Basic indexing

Multi-axis indexing

Selection by Label

Selection by Position

- a single label/integer
- a list or array of labels/integers
- a slice object with labels/integers
- a boolean array
- a callable function with one argument

	pname		
0	A	James	Kim
1	B	Rose	Hwang
2	C	Sam	Park
3	D	Chris	Jang
4	E	Grace	Lee
5	F	Juile	Yoon
6	G	Chirle	Song
7	H	Bob	Kim
8	I	John	Park
9	J	Anne	Lee

	dept	birth	overtime
Education	1990-01-23	23:10:10	
Marketing	1992-10-11	10:15:17	
Education	1995-07-02	16:21:10	
Education	1990-11-23	15:00:20	
Marketing	1993-02-01	21:19:50	
Education	1990-01-23	23:10:10	
Accounting	1992-10-11	10:15:17	
Sales	1995-07-02	16:21:10	
Sales	1990-11-23	15:00:20	
Sales	1993-02-01	21:19:50	

df1 : Basic indexing  
df2 : Selection by Label  
df3 : Selection by Position

```
df2 = df.loc['A':'E', ['dept', 'birth']]  
df3 = df.iloc[0:5, [1,2]]
```

```
df1 = df['pname']
```

# Basic indexing



마지막 축(axis=-1)을 기준으로 1개 indexer 만 사용하는 indexing 이다

indexer 종류	Series	DataFrame
a single <code>label</code> (unique일 것)	<code>s[ index_label ]</code>	<code>df[ column_label ]</code>
a list or array of <code>labels</code>	<code>s[ list of index_labels ]</code>	<code>df[ list of column_labels ]</code>
a slice object with <code>integers</code> or <code>index_labels</code>	<ul style="list-style-type: none"><li>▪ X : s 및 df 에 공통 적용됨을 의미함</li><li>▪ <code>index_label</code> 사용시 index는 정렬된 상태여야 함</li><li>▪ X[ slice with <code>index_labels</code> ], X[ slice with <code>row position</code> ]</li></ul>	
a boolean array	X[ 객체의 row 길이와 같은 길이의 Boolean array ]	
a callable function with one argument	<code>X[ callable function ]</code> 한 개 parameter를 갖고, 위의 4가지 indexer 중 한 가지를 return 하는 함수를 indexer로 사용	

- 특정 조건/상태에서만 적용되는 basic indexing을 사용하면 모호한 표현이 되어 좋지 않다
  - index의 dtype, 정렬 상태, unique 여부에 따라 다른 방법으로도 indexer가 사용 될 수 있다
  - slice objec의 경우 dtype에 따라 integers 및 labels 가 사용 되어 명료하지 않다

[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html)

# Basic indexing 의 예 -1



## [예제20] Basic indexing 에 사용되는 indexer를 살펴 보자

	pname	dept	birth	overtime
A	James Kim	Education	1990-01-23	23:10:10
B	Rose Hwang	Marketing	1992-10-11	10:15:17
C	Sam Park	Education	1995-07-02	16:21:10
D	Chris Jang	Education	1990-11-23	15:00:20
E	Grace Lee	Marketing	1993-02-01	21:19:50
F	Juile Yoon	Education	1992-07-16	14:10:40
G	Chirle Song	Accounting	1993-04-11	09:50:30
H	Bob Kim	Sales	1991-12-07	08:40:40
I	John Park	Sales	1992-06-16	17:30:20
J	Anne Lee	Education	1993-05-05	19:50:20

```
df1 = df['dept'][3]
```

Education → Series

```
df2 = df[:2][['pname', 'birth']]
```

DataFrame → DataFrame

	pname	birth
A	James Kim	1990-01-23
C	Sam Park	1995-07-02
E	Grace Lee	1993-02-01
G	Chirle Song	1993-04-11
I	John Park	1992-06-16

```
df3 = df[(df.dept=='Accounting') | (df.dept=='Sales')]
```

	pname	dept	birth	overtime
G	Chirle Song	Accounting	1993-04-11	09:50:30
H	Bob Kim	Sales	1991-12-07	08:40:40
I	John Park	Sales	1992-06-16	17:30:20

# Basic indexing - Boolean array



## ▶ [예제21] indexing에 사용되는 Boolean array의 특징을 알아보자

- DataFrame, Series, Index 객체의 indexing에 사용할 수 있음
- indexing 할 객체의 행(row)과 동일한 개수이여야 함
- dtype이 bool인 ndarray, list, Series 객체로 작성할 수 있음
- 주로 결과가 True/False인 비교 연산을 사용하여 작성
- 조건이 복잡한 경우 &, |, ~ 등의 비트연산자를 함께 사용하며, 조건을 괄호로 묶어야 함
- np.logical\_and(), np.logical\_or(), np.logical\_not() 활용 가능

a	b	a[b]	a%2==0	a[a%2==0]
0 1	0 True	0 1	0 False	1 2
1 2	1 False	2 3	1 True	3 4
2 3	2 True	3 4	2 False	5 6
3 4	3 True	5 6	3 True	dtype: int64
4 5	4 False	dtype: int64	4 False	
5 6	5 True		5 True	
dtype: int64				

# Basic indexing - Boolean array 의 예



## ▶ [예제21] Boolean array 를 사용한 basic indexing을 살펴보자

df = DataFrame 객체

se = df.english

ss = df['salary']

sd = df['dept']

연산에는 se.values 가 사용됨 (se 및 ss, sd 모두 values가 사용되는 것임)

boolean indexing	결과
se[se>0]	se 의 값이 0 보다 큰 것
se[se>se.mean()]	se 값이 se.mean() 보다 큰 것
df[ (ss>4500) & (ss<5000) ] df[ np.logical_and(ss>4500, ss<5000) ]	ss 값이 4500 보다 크고 5000 보다 작은 것
df[ (sd=='Education')   (sd=='Accounting') ] df[np.logical_or(sd=='Education', sd=='Accounting')]	sd 가 'Education' 이거나 'Accounting' 인 것
df[~(sd=='Education')] df[np.logical_not(sd=='Education')]	sd가 'Education' 이 아닌 것

# isin 메서드로 Boolean array 작성



## [예제22] isin 메서드를 사용하면 Boolean array를 반환한다

x.isin(values)

- 각 element가 values 내의 값인지에 대한 True/False로 구성된 객체 반환

values

- df의 values : iterable, Series, DataFrame or dict
- s의 values : set, list-like
- dict의 키는 column labels 를 사용하고, DataFrame 은 구조가 같아야 함
- Series 이면 index임

	korean	english
Kim	A	B
Yoon	A	A
Choi	A	C
Park	B	B

df1 = df.isin(['B', 'C'])

	korean	english
Kim	True	True
Yoon	True	True
Choi	True	False
Park	False	True

w = {'korean' : ['A'],  
'english' : ['A', 'B']}  
df2 = df.isin(w1)

	korean	english
Kim	False	True
Yoon	False	False
Choi	False	True
Park	True	True

x.isna(values) 또한 활용 할 수 있다

# Series - Accessors



▶ Series는 몇 가지 dtype 내부 객체 접근을 위한 accessor가 제공된다

데이터 타입	Accessor
Datetime, Timedelta, Period	dt
String	str

데이터 타입	Accessor
Categorical	cat
Sparse	sparse

▶ pandas에는 4가지 시계열 concept이 있다

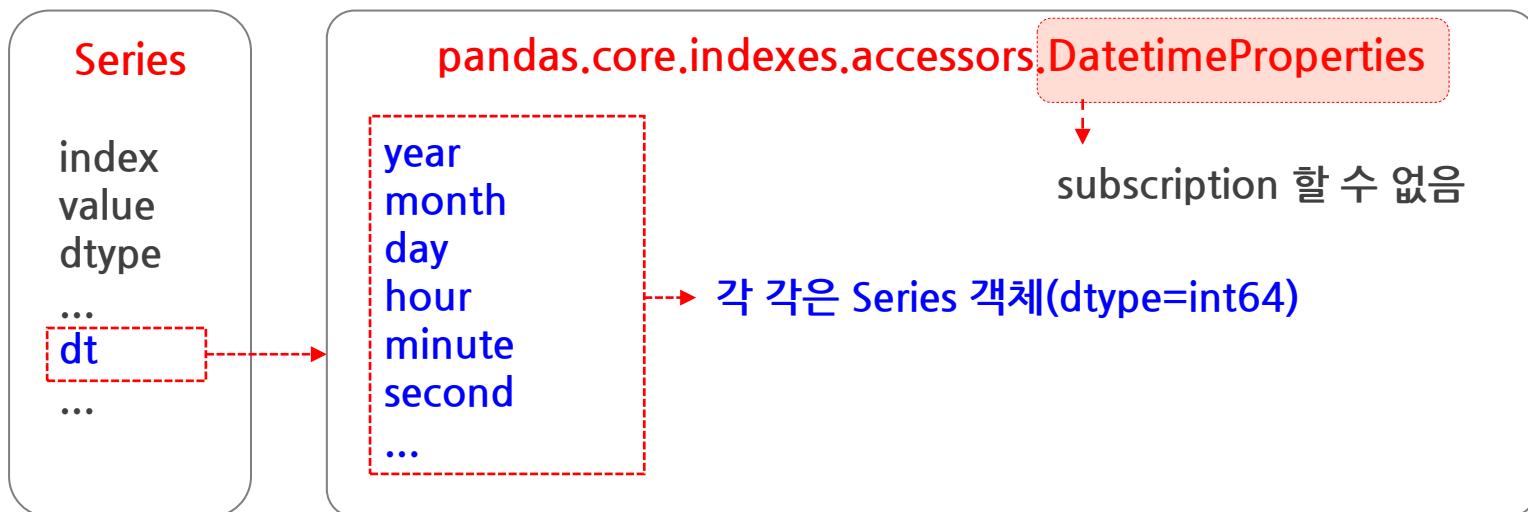
Concept	Scalar Class	Array Class	pandas DataType	Primary Creation Method
Date times	Timestamp	DatetimeIndex	datetime64[ns] datetime64[ns,tz]	▪ to_datetime ▪ date_range
Time deltas	Timedelta	TimedeltaIndex	timedelta64[ns]	▪ to_timedelta ▪ timedelta_range
Time spans	Period	PeriodIndex	period[freq]	▪ Period ▪ period_range
Date offsets	DateOffset	None	None	▪ DateOffset

<https://pandas.pydata.org/pandas-docs/stable/reference/series.html> (Accessors 참조)  
[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html) (시계열)

# datetime 타입의 Series 다루기



## datetime 타입의 Series에서 Datetime properties의 사용



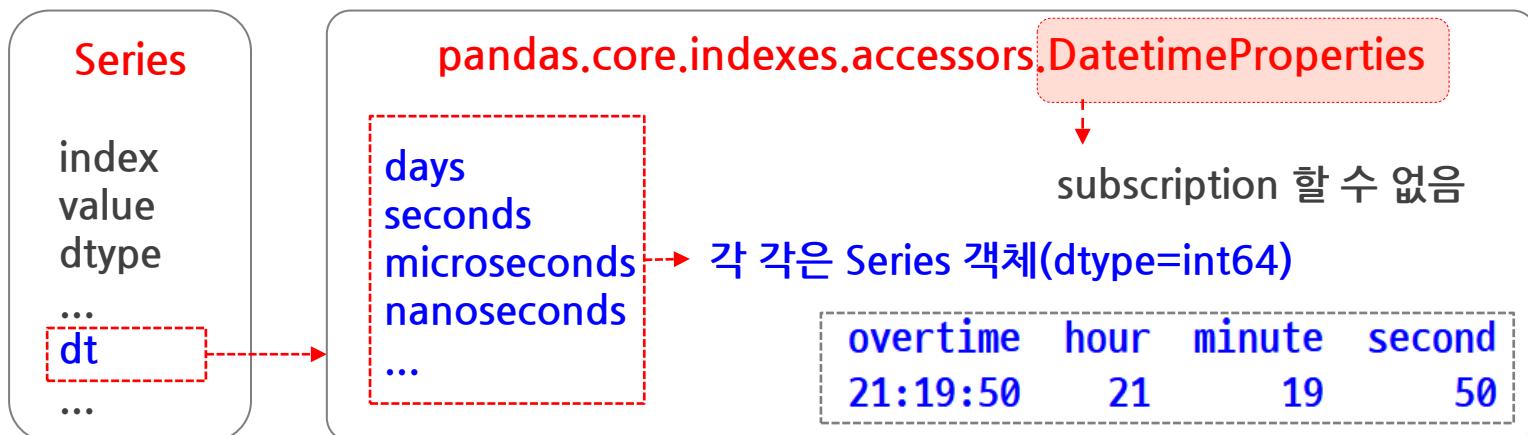
datetime 타입의 활용	결과
<code>sb[sb.dt.year == 1992]</code>	sb의 년도가 1992 인 것, Series (dtype=int64)
<code>df['year'] = sb.dt.year</code>	df에 'year' 열을 sb.dt.year로 갱신 또는 추가

weekday : 요일 (0~6, 월~일), dayofyear : 일 년 중 몇 번째 날

# timedelta 타입의 Series 다루기



## [예제23] timedelta 타입의 Series에서 Datetime properties의 사용



timedelta 타입의 활용	결과
seconds = df.overtime.dt.seconds	df에 'hour' 열 추가
df['hour'] = seconds //3600	df에 'hour' 라는 열 추가
df['minute'] = (seconds - df.hour*3600)//60	df에 'minute' 라는 열 추가
df['second'] = seconds%60	df에 'second' 라는 열 추가
df=df[df.hour >18]	overtime 18시간 초과 근무자

# Multi-axis indexing의 indexer 종류



- Multi-axis indexing에 사용되는 indexer 종류는 다음과 같다

Selection by Label	Selection by Position
a single <b>label</b>	an <b>integer</b>
a list or array of <b>labels</b>	a list or array of <b>integers</b>
a slice object with <b>labels</b>	a slice object with <b>integers</b>
a boolean array	a boolean array <b>차이점은 별도로 다름</b>
a callable <b>function</b> with one argument	a callable <b>function</b> with one argument

- 존재하지 않는 Label, Position 사용시 indexer 종류에 따라 다른 동작한다  
예) KeyError, IndexError 또는 NaN 값을 갖는 항목으로 추가
- boolean array는 True/False로 이루어진 배열이다
- callable function은 1개 parameter를 갖고, 가능한 형태의 indexer를 반환하는 함수이다

Callable은 본 학습에서 다루지 않음

# Multi-axis indexing



Multi-axis indexing 은 by Label과 by Position이 제공된다

Selection by Label		Selection by Position	
▪ <code>s.loc[ indexer ]</code>		▪ <code>s.iloc [ indexer ]</code>	
▪ <code>df.loc[row_indexer, column_indexer]</code>		▪ <code>df.iloc[row_indexer, column_indexer]</code>	
▪ index 및 columns 의 ‘label’ 사용		▪ index 및 columns의 ‘position’ 사용	

- Series는 row\_indexer 만 존재한다
- Series, DataFrame의 indexer 로 “.” 사용시 모든 대상 선택을 의미한다  
예) `df.loc[ :, column_indexer ]` → 모든 행에 대해서 특정 열 선택
- 예) `df.loc[row_indexer]` 또는 `df.loc[row_indexer, : ]` → 특정 행에 대해서 모든 열 선택

# Selection by Label



## ▶ [예제24] .loc 을 사용한 Selection 동작을 살펴본다

indexer	설명
a single label	<ul style="list-style-type: none"><li>존재하는 label을 사용해야 함</li><li>존재하지 않는 label 사용시 <b>KeyError 발생</b></li><li>label이 unique 하지 않으면 label과 같은 모든 행 또는 열 추출</li></ul>
a list or array of labels	<ul style="list-style-type: none"><li>하나의 label을 여러 번 중복 사용 가능</li><li>존재하지 않는 label 사용시 <b>KeyError 발생</b></li></ul>
a slice object with labels	<ul style="list-style-type: none"><li>label을 정렬하여 사용하는 것이 좋음</li><li>정렬되지 않은 경우 “<b>존재하며, 중복 없는 index 만</b>” 사용해야 함</li></ul>

## ▶ slice object에서 S, E label과 step 사용

.loc[ S label : E label : step [, S label : E label : step] ]

[index, columns] S label 부터 E label **까지**의 데이터를 step 씩 건너뛰며 추출

E label과 같은 것이 있으면 포함

- label의 type은 index의 dtype에 따라 달라지며, step은 정수이다

# Selection by Label 예 - 1/2



▶ [예제24] Selection by label을 사용 예를 분석하여 보자

<table border="1"><thead><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr><tr><th>a</th><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><th>b</th><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><th>c</th><td>8</td><td>9</td><td>10</td><td>11</td></tr></thead></table>		A	B	C	D	a	0	1	2	3	b	4	5	6	7	c	8	9	10	11	<code>df1 = df.loc['a']</code>	<code>df2 = df.loc[:, 'A']</code>	<code>df3 = df.loc['a', 'B']</code>							
	A	B	C	D																										
a	0	1	2	3																										
b	4	5	6	7																										
c	8	9	10	11																										
		<table border="1"><thead><tr><th></th><th>A</th><th>0</th></tr></thead><tbody><tr><th>B</th><td></td><td>1</td></tr><tr><th>C</th><td></td><td>2</td></tr><tr><th>D</th><td></td><td>3</td></tr></tbody></table>		A	0	B		1	C		2	D		3	<table border="1"><thead><tr><th></th><th>a</th><th>0</th></tr></thead><tbody><tr><th>b</th><td></td><td>4</td></tr><tr><th>c</th><td></td><td>8</td></tr></tbody></table>		a	0	b		4	c		8						
	A	0																												
B		1																												
C		2																												
D		3																												
	a	0																												
b		4																												
c		8																												
			<table border="1"><thead><tr><th></th><th>1</th></tr></thead></table>		1																									
	1																													
			<code>df4 = df.loc['d']</code>																											
			KeyError: 'd'																											
<table border="1"><thead><tr><th></th><th>C</th><th>B</th><th>C</th><th>D</th></tr><tr><th>a</th><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><th>b</th><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><th>a</th><td>8</td><td>9</td><td>10</td><td>11</td></tr></thead></table>		C	B	C	D	a	0	1	2	3	b	4	5	6	7	a	8	9	10	11	<code>df1 = df.loc['a']</code>	<code>df2 = df.loc['a', 'C']</code>								
	C	B	C	D																										
a	0	1	2	3																										
b	4	5	6	7																										
a	8	9	10	11																										
		<table border="1"><thead><tr><th></th><th>C</th><th>B</th><th>C</th><th>D</th></tr><tr><th>a</th><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><th>a</th><td>8</td><td>9</td><td>10</td><td>11</td></tr></thead></table>		C	B	C	D	a	0	1	2	3	a	8	9	10	11	<table border="1"><thead><tr><th></th><th>a</th><th>C</th><th>C</th></tr></thead><tbody><tr><th>a</th><td></td><td>0</td><td>2</td></tr><tr><th>a</th><td></td><td>8</td><td>10</td></tr></tbody></table>		a	C	C	a		0	2	a		8	10
	C	B	C	D																										
a	0	1	2	3																										
a	8	9	10	11																										
	a	C	C																											
a		0	2																											
a		8	10																											
<table border="1"><thead><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr><tr><th>a</th><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><th>b</th><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><th>c</th><td>8</td><td>9</td><td>10</td><td>11</td></tr></thead></table>		A	B	C	D	a	0	1	2	3	b	4	5	6	7	c	8	9	10	11	<code>df1 = df.loc[['a', 'b']]</code>	<code>df2 = df.loc[:, ['A', 'D']]</code>								
	A	B	C	D																										
a	0	1	2	3																										
b	4	5	6	7																										
c	8	9	10	11																										
		<table border="1"><thead><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr><tr><th>a</th><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><th>b</th><td>4</td><td>5</td><td>6</td><td>7</td></tr></thead></table>		A	B	C	D	a	0	1	2	3	b	4	5	6	7	<table border="1"><thead><tr><th></th><th>A</th><th>D</th></tr></thead><tbody><tr><th>a</th><td>0</td><td>3</td></tr><tr><th>b</th><td>4</td><td>7</td></tr><tr><th>c</th><td>8</td><td>11</td></tr></tbody></table>		A	D	a	0	3	b	4	7	c	8	11
	A	B	C	D																										
a	0	1	2	3																										
b	4	5	6	7																										
	A	D																												
a	0	3																												
b	4	7																												
c	8	11																												

# Selection by Label 예 - 2/2



## ▶ [예제24] Selection by label을 사용 예를 분석하여 보자

정렬한 경우

	A	B	C	D
a	4	5	6	7
b	0	1	2	3
b	8	9	10	11

```
df.sort_index(axis='index', inplace=True)
```

```
df1 = df.loc[:'a']
```

	A	B	C	D
a	4	5	6	7

```
df2 = df.loc['a':'c', 'B':'E']
```

	B	C	D
a	5	6	7
b	1	2	3
b	9	10	11

	C	D	A	B
b	0	1	2	3
a	4	5	6	7
b	8	9	10	11

```
df1 = df.loc[:'a', 'C':'A']
```

	C	D	A
b	0	1	2
a	4	5	6

```
df2 = df.loc[:'c']  
df3 = df.loc[:, 'B':'E']
```

KeyError: 'c'

KeyError: 'E'

정렬하지 않은 경우

# Label의 dtype에 따른 사용법 -2



## ▶ [예제25] Label의 dtype에 따른 사용법을 알아보자

Label의 dtype	설명
category	<ul style="list-style-type: none"><li>존재하는 label 만 사용, 아니면 KeyError</li></ul>
datetime	<ul style="list-style-type: none"><li>'-' 없이 '20191122'과 같이 문자열로 사용함</li><li>slice 사용시 범위 밖 이어도 상관 없음, 정렬되어 있지 않아도 됨</li></ul>
timedelta	<ul style="list-style-type: none"><li>'10:15:12' 또는 '1days 11:25:33' 과 같이 문자열로 사용함</li><li>slice 사용시 범위 밖 이어도 상관 없음, 정렬되어 있어야 함</li></ul>

	A	B	C
2020-01-07	0	1	2
2020-01-06	3	4	5
2020-01-04	6	7	8
2020-01-05	9	10	11
2020-01-08	12	13	14

정렬되지 않음

df1 = df.loc['20200104':'20200106', ['B', 'C']]	df2 = df.loc['20200105':'20200110'] 범위 밖																																
<table border="1"><thead><tr><th></th><th>B</th><th>C</th></tr></thead><tbody><tr><td>2020-01-06</td><td>4</td><td>5</td></tr><tr><td>2020-01-04</td><td>7</td><td>8</td></tr><tr><td>2020-01-05</td><td>10</td><td>11</td></tr></tbody></table>		B	C	2020-01-06	4	5	2020-01-04	7	8	2020-01-05	10	11	<table border="1"><thead><tr><th></th><th>A</th><th>B</th><th>C</th></tr></thead><tbody><tr><td>2020-01-07</td><td>0</td><td>1</td><td>2</td></tr><tr><td>2020-01-06</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2020-01-05</td><td>9</td><td>10</td><td>11</td></tr><tr><td>2020-01-08</td><td>12</td><td>13</td><td>14</td></tr></tbody></table>		A	B	C	2020-01-07	0	1	2	2020-01-06	3	4	5	2020-01-05	9	10	11	2020-01-08	12	13	14
	B	C																															
2020-01-06	4	5																															
2020-01-04	7	8																															
2020-01-05	10	11																															
	A	B	C																														
2020-01-07	0	1	2																														
2020-01-06	3	4	5																														
2020-01-05	9	10	11																														
2020-01-08	12	13	14																														

# Label의 dtype에 따른 사용법 - 2



## ▶ [예제25] Label의 dtype에 따른 사용법을 알아보자

	A	B	C
00:16:40	0	1	2
00:33:20	3	4	5
00:50:00	6	7	8
01:06:40	9	10	11
01:23:20	12	13	14
01:40:00	15	16	17

```
df1 = df.loc['01:00:00':'02:00:00'] 범위 밖  
df2 = df.loc['00:30:00':'01:23:20']
```

	A	B	C		A	B	C
01:06:40	9	10	11	00:33:20	3	4	5
01:23:20	12	13	14	00:50:00	6	7	8
01:40:00	15	16	17	01:06:40	9	10	11
				01:23:20	12	13	14

정렬 됨

# Selection by Position



## ▶ [예제26] .iloc 을 사용한 Selection 동작을 살펴본다

indexer	설명
an integer	<ul style="list-style-type: none"><li>▪ index 범위 내의 정수만 사용 가능</li><li>▪ 범위 내가 아닐 경우 <b>IndexError</b> 발생</li></ul>
a list or array of integers	<ul style="list-style-type: none"><li>▪ 정수(index position number)는 중복 사용 가능</li><li>▪ index 범위 내의 정수만 사용 가능</li></ul>
a slice object with integers	<ul style="list-style-type: none"><li>▪ index 범위내의 정수가 아니어도 됨</li></ul>

## ▶ slice object에서 S, E position과 step 사용

.iloc[ S position : E position : step, [S position : E position : step] ]

- [index, columns] S position 부터 E position 전 까지의 데이터를 step 씩 건너뛰며 추출

E position 포함하지 않음

Selection by Label 보다 규칙이 단순하여, 사용이 편리하다

# Selection by Position 예 - 1/2



▶ [예제26] Selection by Position의 사용 예를 분석하여 보자

	A	B	C	D
a	0	1	2	3
b	4	5	6	7
c	8	9	10	11

`df1 = df.iloc[2]`

	A	B	C	D
A	8			
B	9			
C	10			
D	11			

`Series`

`df2 = df.iloc[1, 2]`

6
<code>df4 = df.iloc[4]</code>
<code>IndexError:</code>

`df3 = df.iloc[:, 1]`

	a	b	c
a	1		
b	5		
c	9		

`Series`

	A	B	C	D
a	0	1	2	3
b	4	5	6	7
c	8	9	10	11

`df1 = df.iloc[[1, 2]]`

	A	B	C	D
b	4	5	6	7
c	8	9	10	11

`df2 = df.iloc[[2], [0, 2, 1]]`

	A	C	B
c	8	10	9

`df3 = df.iloc[2, [0, 2, 1]]`

A	8
C	10
B	9

`Series`

# Multi-axis indexing - Boolean array



▶ [예제27] Boolean array를 .loc과 .iloc 의 indexer로 사용해 보자

x.loc[r\_indexer, c\_indexer], x.iloc[r\_indexer, c\_indexer]

x.loc

- basic indexing과 동일 방법으로 행/열의 indexer로 Boolean array를 사용 함
- dtype이 bool 인 list, ndarray, Series 객체로 작성할 수 있음

x.iloc

- dtype=bool 인 1D array (list, ndarray 등) 사용 (Series 객체 안됨)

	a	b	c
A	49	97	12
B	34	95	5
C	71	71	23
D	6	3	96
E	72	69	79

```
df1 = df[df.a > 50]
df2 = df.loc[df.a>50,:]
```

	a	b	c
C	71	71	23
E	72	69	79

```
ridx = (df.a > 50) & (df.b > 50)
cidx = df.loc['A'] > df.loc['C']
df3 = df.loc[ridx, cidx]
df4 = df.iloc[ridx.to_list(),
              cidx.to_list()]
```

	b
C	71
E	69

# MultilIndex 객체



## [예제28] MultilIndex 객체 생성을 위한 메서드 사용 방법을 알아본다

• pd.MultiIndex.from\_tuples( tuples, names )

- tuples : MultilIndex 각 row를 항으로 하는 1D array
- names : index의 각 column 이름, 1D array

• pd.MultiIndex.from\_product( iterables, names )

- iterables : MultilIndex 각 column을 항으로 하는 1D array
- iterable 들의 각 항 개수를 곱한 것 만큼의 항이 생성됨

```
idx_t = [ ['A', 1], ['A', 2],  
         ['B', 1], ['B', 2],  
         ['C', 1], ['C', 2] ]
```

comprehension으로  
변경하여 보자

```
midx1 = pd.MultiIndex.from_tuples(idx_t,  
                                   names=['L0', 'L1'])
```

```
idx_p = [['A', 'B', 'C'], [1, 2]]
```

```
midx2 = pd.MultiIndex.from_product(idx_p,  
                                   names=['L0', 'L1'])
```

```
MultiIndex([('A', 1),  
            ('A', 2),  
            ('B', 1),  
            ('B', 2),  
            ('C', 1),  
            ('C', 2)],  
           names=['L0', 'L1'])
```

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.MultiIndex.html>

# Multi-index indexing



▶ [예제29] Multi-Index는 tuple을 이용하여 각 level의 대상을 지정한다

	A		B	
	a	b	a	b
a	58	74	53	89
y	73	65	60	80
b	x	84	52	84
y	84	74	65	86

df1 = df['A']['a']		
a	x	58
	y	73
b	x	84
	y	84

Name: a, dtype: int32

df2 = df.loc[:, ('A', 'a')]		
a	x	58
	y	73
b	x	84
	y	84

Name: (A, a), dtype: int32

```
df3 = df.copy()  
df3[('A', 'a')] = [100]*4  
df3.loc[('a', 'y'), ('B', 'b')] = 1234
```

```
df3['A']['a'] = [100]*4
```

SettingWithCopyWarning:



	A		B	
	a	b	a	b
a	x	100	74	53
y		100	65	60
b	x	100	52	84
y		100	74	65

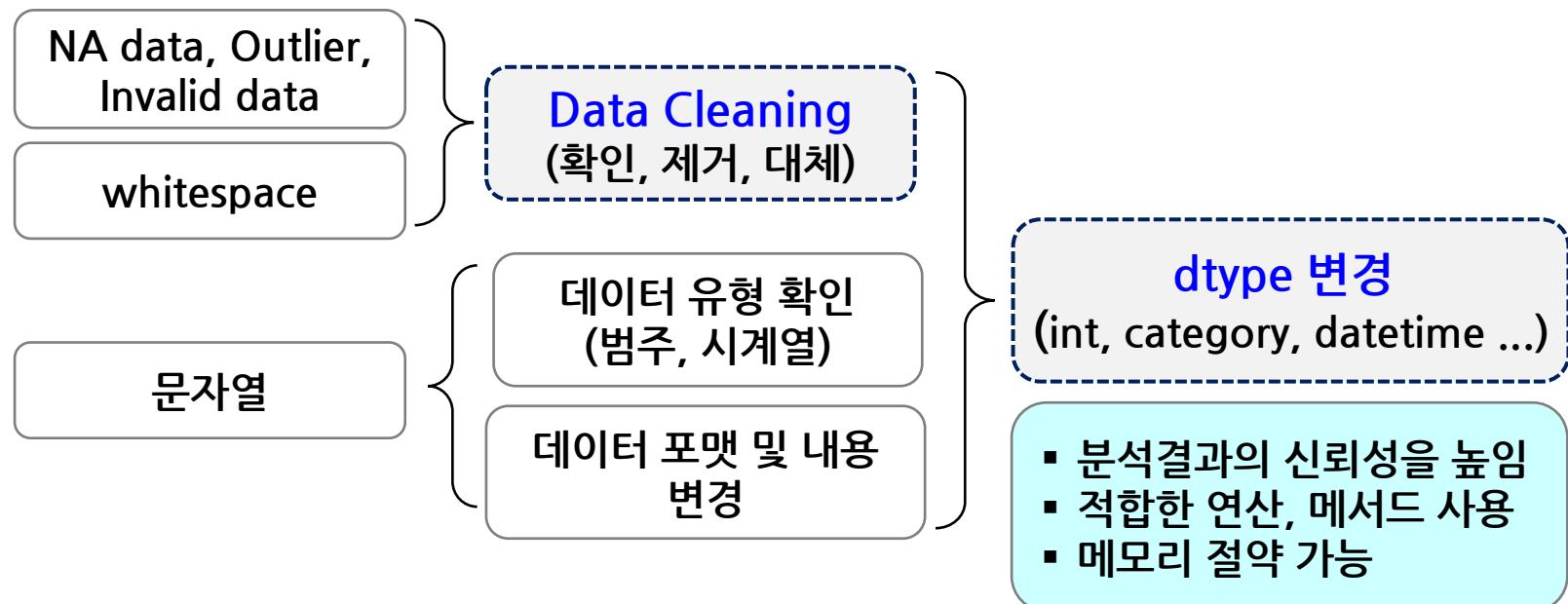
1234

- indexing의 대부분은 view로 취급되고 대입이 가능하다
- df['A'][1] 은 chained indexing 이라 하며, indexing이 두 번 실행 된 것이고 copy 이다
- df[('A', 1)]과 df.loc[:, ('A', 1)] 은 indexing이 한 번 실행 된 것이다

# Data Cleaning, dtype 변경의 필요성



- 파일에서 데이터를 읽어 올 경우 자동으로 dtype이 정해진다
- 숫자 데이터만으로 이루어진 경우가 아니라면 잘못된 dtype으로 정해질 수 있다
  - NA data, Outlier, Invalid data, whitespace 데이터는 적절한 처리가 우선되어야 한다
  - dtype에 따라 사용 가능한 연산, 메서드, 메모리 사용량 등이 다르다



# 누락 데이터(Missing data, NA data) 종류



## ▶ [예제30] pandas의 missing data 종류

missing data - **None, np.nan, pd.NaT** (각각을 missing value로 부름)

np.nan	▪ float 타입으로 숫자의 missing 을 의미
pd.NaT	▪ np.datetime64 타입으로 missing 날짜를 의미
np.inf	▪ pd.options.mode.use_inf_as_na=True → np.inf를 missing data 취급

[https://pandas.pydata.org/pandas-docs/version/0.25/user\\_guide/missing\\_data.html](https://pandas.pydata.org/pandas-docs/version/0.25/user_guide/missing_data.html)

## ▶ 연산 시 missing data는 제외하고 처리함

```
pd.options.mode.use_inf_as_na = False  
s = pd.Series([np.nan, pd.NaT, None, np.inf])  
print(s.count(), s.sum())
```

결과  
1 inf → options = False  
0 0 → options = True

## ▶ missing data의 상등 비교

NA, not available, missing 은 같은 의미

가능

None == None, np.inf == np.inf → True

불가능

np.nan == np.nan, pd.NaT == pd.NaT → False

# NA data 확인 및 처리 함수



## ▶ [예제31] 누락 데이터 확인 및 처리를 위한 함수/메서드

### NA data 확인

- |           |  |
|-----------|--|
| pandas    | ▪ <code>isna(obj)</code> , <code>isnull(obj)</code> , <code>notna(obj)</code> , <code>notnull(obj)</code> array-like 객체            |
| DataFrame | ▪ <code>df.info()</code> , <code>df.isna()</code> , <code>df.isnull()</code> , <code>df.notna()</code> , <code>df.notnull()</code> |
| Series    | ▪ <code>s.isna()</code> , <code>s.isnull()</code> , <code>s.notna()</code> , <code>s.notnull()</code>                              |

### NA data 처리

- |           |   |
|-----------|---|
| DataFrame | ▪ <code>df.dropna()</code> , <code>df.fillna()</code> , <code>df.replace()</code> , <code>df.interpolate()</code> |
| Series    | ▪ <code>s.dropna()</code> , <code>s.fillna()</code> , <code>s.replace()</code> , <code>s.interpolate()</code>     |

- NA data replace에 사용할 값 생성 방법
  - 기본 통계 값 활용 (mean, max, min, median...)
  - indexing, selection 기술, 연산자 활용

# DataFrame 정보 출력



df.info(...)

- index, columns, dtypes, memory usage 정보 출력
- 출력 정보의 정도를 조절할 수 있는 parameters가 있음
- memory\_usage='deep' → deep memory introspection 설정

ID	pname	birth	dept	english	japanese	chinese
18030201	James Kim	1990-01-23	Education	1	1.0	
18030202	Rose Hwang	1992-10-11	Marketing		2.0	Nan
19030401	Sam Park	1995-07-02	Education	1	Nan	Nan

Int64Index: 10 entries, 18030201 to 19090202  
Data columns (total 6 columns): → 열의 개수

#	Column	Non-Null Count	Dtype
0	pname	10 non-null	object
1	birth	10 non-null	object
2	dept	10 non-null	object
3	english	6 non-null	object
4	japanese	5 non-null	float64
5	chinese	6 non-null	object

dtypes: float64(1), object(5)  
memory usage: 560.0+ bytes

각 열에 대한 label, non-null 개수 및 dtype 출력

df.info(memory\_usage='deep')  
memory usage: 3.1 KB

어떤 데이터 일까?

# NA data 확인



df.isna()  
df.isnull()

- Boolean 데이터로 작성된 DataFrame 객체 반환 (NA value → True)
- isna, isnull은 동일 동작 (isnull 는 isna 의 alias)
- any(), all() 등으로 정보를 요약 할 수 있음

df.isna()

```
<class 'pandas.core.frame.DataFrame'>
      pname  birth  dept  english  japanese  chinese
ID
18030201  False  False  False    False    False  False
18030202  False  False  False    False    False  True
19030401  False  False  False    False   True  True
```

df.isna().any()

any() 와 all() 의 차이는 무엇일까?

df.isna().any().any()

```
<class 'pandas.core.series.Series'>
pname      False
birth      False
dept       False
english    False
japanese   True
chinese    True
```

df.isna().all()

```
pname      False
birth      False
dept       False
english   False
japanese  False
chinese   False
```

```
<class 'numpy.bool_'>
True
```

# NA data 제거



- `df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)`

• NA value 를 포함한 행 또는 열이 제거 된 DataFrame 객체 반환

axis	<ul style="list-style-type: none"><li>▪ axis=0 이거나 'index' : NA value 포함 행(row) 제거</li><li>▪ axis=1 이거나 'columns' : NA value 포함 열(column) 제거</li></ul>
how	<ul style="list-style-type: none"><li>▪ how='any' : NA value 가 하나라도 포함된 경우 True</li><li>▪ how='all' : 모든 값이 NA value 인 경우 True</li></ul>
thresh	<ul style="list-style-type: none"><li>▪ int, non-NA value(유효 데이터) 개수가 설정 값 이상 일 때 제거 안함</li></ul>
subset	<ul style="list-style-type: none"><li>▪ array-like, NA value를 살펴 볼 label 목록</li><li>▪ axis=0 : axis=1에 대한 label 을 목록으로 작성함</li></ul>
inplace	<p><code>df = df.dropna()</code></p> <p><code>df.dropna(inplace=True)</code></p>

# df.dropna 동작 이해

▶ [예제32] a ~ e 의 결과를 예측하여 적어본 뒤, 실행하여 결과를 확인 한다

	dept	english	japanese	chinese
0	Education	1	1.0	NaN
1	Marketing	Nan	2.0	NaN
2	Education	1	NaN	NaN

a = df.dropna()

Empty DataFrame

Columns: [dept, english, japanese, chinese]

Index: []

b = df.dropna(axis=1)

	dept
0	Education
1	Marketing
2	Education

c = df.dropna(axis='columns', how='all')

	dept	english	japanese
0	Education	1	1.0
1	Marketing	Nan	2.0
2	Education	1	NaN

d = df.dropna(thresh=3)

e = df.dropna(subset=['english', 'japanese'])

	dept	english	japanese	chinese
0	Education	1	1.0	NaN

df.dropna(axis=1, inplace=True)

	dept
0	Education
1	Marketing
2	Education

# df.fillna 메서드



- df.fillna(value=None, method=None, axis=None, inplace=False, limit=None, ... )
  - NA value 를 value 또는 method를 사용하여 변경한 DataFrame 객체 반환

value	<ul style="list-style-type: none"><li>▪ scalar, dict, Series, or DataFrame</li><li>▪ NA data를 대신 할 값을 지정함</li><li>▪ dict, Series, DataFrame 을 사용해 행/열 별 채우기 값 별도 지정 가능</li></ul>
method	<ul style="list-style-type: none"><li>▪ {'backfill', 'bfill', 'pad', 'ffill', None}</li><li>▪ value=None 일 때, NA data를 대신 할 값 선정 방법을 지정함</li><li>▪ 'backfill', 'bfill' : 다음 발견되는 valid observation으로 채움</li><li>▪ 'pad', 'ffill' : 이전에 발견된 valid observation으로 채움</li></ul>
axis	<ul style="list-style-type: none"><li>▪ axis=0 이거나 'index' : 행 방향으로 채우기 진행</li><li>▪ axis=1 이거나 'columns' : 열 방향으로 채우기 진행</li></ul>
inplace	<ul style="list-style-type: none"><li>▪ bool, True인 경우 대상에 직접 반영 하고, None을 반환</li></ul>
limit	<ul style="list-style-type: none"><li>▪ method를 지정했을 때, axis에 따라 선정 값을 몇 번 사용하는가</li><li>▪ value를 지정했다면, 그 값의 사용 횟수 (axis는 항상 0으로 동작)</li></ul>

# df.fillna 동작 이해



▶ [예제33] a ~ d 의 결과를 예측하여 적어본 뒤, 실행하여 결과를 확인 한다

	english	chinese
0	1.0	2.0
1	Nan	3.0
2	1.0	Nan
3	Nan	3.0
4	Nan	Nan
5	Nan	1.0

a = df.fillna(0)

	english	chinese
0	1.0	2.0
1	0.0	3.0
2	1.0	0.0
3	0.0	3.0
4	0.0	0.0
5	0.0	1.0

b = df.fillna({'english': -1, 'chinese': -2})

	english	chinese
0	1.0	2.0
1	-1.0	3.0
2	1.0	-2.0
3	-1.0	3.0
4	-1.0	-2.0
5	-1.0	1.0

c = df.fillna(method='ffill', limit=2)

	english	chinese
0	1.0	2.0
1	1.0	3.0
2	1.0	3.0
3	1.0	3.0
4	1.0	3.0
5	Nan	1.0

d = df.fillna(method='bfill')

	english	chinese
0	1.0	2.0
1	1.0	3.0
2	1.0	3.0
3	Nan	3.0
4	Nan	1.0
5	Nan	1.0

# df.replace 메서드



- df.replace(to\_replace, value=None, inplace=False, limit=None, regex=False, ... )
  - to\_replace로 주어진 대상이 value로 주어진 값으로 변경 된 DataFrame 객체

to_replace	<ul style="list-style-type: none"><li>▪ str, regex, list, dict, Series, int, float, or None</li><li>▪ value로 대체될 값을 찾는 방법</li><li>▪ 아래 제시된 링크에서 상세 설명 참조</li></ul>
value	<ul style="list-style-type: none"><li>▪ scalar, dict, list, str, regex, default None</li><li>▪ to_replace에 매칭하는 값을 대체할 값을 지정함</li><li>▪ dict을 사용해 열 별 채우기 값을 별도 지정 가능</li></ul>
inplace	<ul style="list-style-type: none"><li>▪ bool, True인 경우 대상에 직접 반영하고, None을 반환</li></ul>
limit	<ul style="list-style-type: none"><li>▪ NA values를 다른 value로 변경하는 동작의 최대 횟수</li></ul>
regex	<ul style="list-style-type: none"><li>▪ bool or same types as to_replace</li><li>▪ True 설정 시 to_replace 및 value의 정규식 사용 가능</li><li>▪ to_replace는 str을 사용해야 함</li></ul>

<https://pandas.pydata.org/pandas-docs/version/0.25/reference/api/pandas.DataFrame.replace.html>

# df.replace 동작 이해 -1



▶ [예제34] 다음의 replace 메서드 동작을 이해하여 보자

	dept
0	Education
1	Marketing
2	Education
3	Education
4	Marketing
5	Education
6	Accounting
7	Sales

```
a = df.replace("Education", "E")
b = df.replace(["Education", "Marketing", "Sales", "Accounting"],
               ["E", "M", "S", "A"])
c = df.replace({"Education": 0,
                "Marketing": 1,
                "Sales": 2,
                "Accounting": 3})
s = pd.Series([0, 1, 2, 3],
              index = ["Education", "Marketing", "Sales", "Accounting"])
d = df.replace(s)
```

부서(dept)의 종류 개수에  
독립적인 코드로 작성하여 보자

a	dept
0	E
1	Marketing
2	E
3	E
4	Marketing
5	E
6	Accounting
7	Sales

b	dept
0	E
1	M
2	E
3	E
4	M
5	E
6	A
7	S

c d	dept
0	0
1	1
2	0
3	0
4	1
5	0
6	3
7	2

# df.replace 동작 이해 -2



## [예제34] 다음의 replace 메서드 동작을 이해하여 보자

	english	japanese	chinese	salary
0	1	1.0		3,456
1		2.0	Nan	4,320
2	1	Nan	Nan	5,600

$^{\wedge}\s+\$$  : 1개 이상의 whitespace 만으로 구성  
 $(\backslash d), (\backslash d)$  : 숫자 사이에 콤마(,)가 있음  
W1 W2

```
a = df.replace(r'^\s+$', np.nan, regex=True)
```

	english	japanese	chinese	salary
0	1	1.0	NaN	3,456
1	NaN	2.0	NaN	4,320
2	1	Nan	Nan	5,600

a, b 동작을 한 번에 실행할 수 있도록  
list, dict 을 사용할 수 있다

```
b = a.replace(r'(\d),(\\d)', r'\\1\\2', regex=True)
```

	english	japanese	chinese	salary
0	1	1.0	NaN	3456
1	NaN	2.0	NaN	4320
2	1	Nan	Nan	5600

```
c = b.replace(np.nan, 0)
```

	english	japanese	chinese	salary
0	1	1.0	0.0	3456
1	0	2.0	0.0	4320
2	1	0.0	0.0	5600

# 병합 - concat



## [예제35] concat은 index/column을 기준으로 병합한다

- pd.concat(objs, axis=0, join='outer', ignore\_index=False, verify\_integrity=False, ... )

• index를 기준으로, 행/열 방향으로 DataFrame을 병합함

axis	▪ 0 or 'index' : 행 방향, 1 or 'columns' : 열 방향
join	▪ { 'outer', 'inner' }, 매치되는 index/column 없을 때의 동작 ▪ outer : NaN 채우기, inner : 삭제하기
ignore_index	▪ index를 무시하고 RangeIndex로 변경
verify_integrity	▪ True : 중복 데이터 있으면 ValueError 발생

df1	A	B
a	1	o
b	2	p
c	3	q
d	4	r

df2	A	B
a	1	x
b	20	y
c	15	z

df3	C	D
a	10	Q
b	20	X
c	15	Y
d	40	Z

	A	B
a	1	o
b	2	p
c	3	q
d	4	r
a	1	x
b	20	y
c	15	z

A	B	C	D
a	1	o	10
b	2	p	20
c	3	q	15
d	d	r	40
			Z

[df1, df3], axis=1

[df1, df2], axis=0

# 병합 - merge

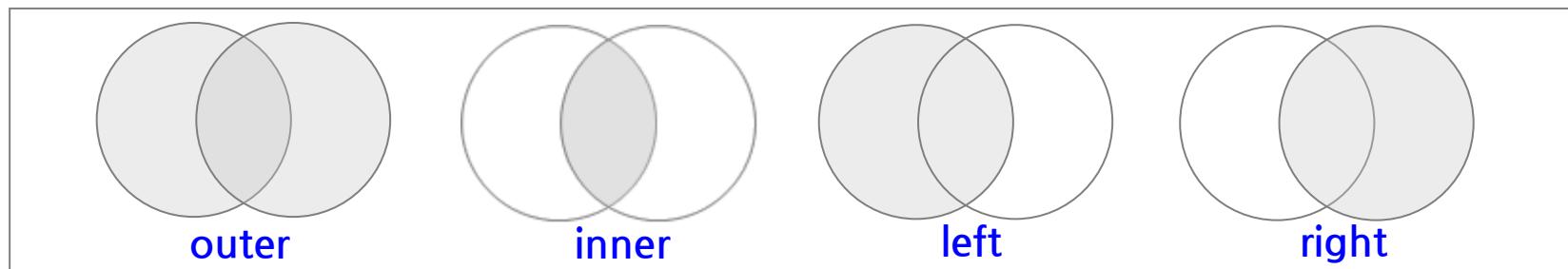


▶ [예제36] merge는 병합 기준을 직접 지정하여 두 객체를 병합한다

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,  
        left_index=False, right_index=False, ...)
```

on에 지정된 병합 기준 또는 index에 따라 left, right 병합

left, right	▪ DataFrame or named Series
how	▪ {'left', 'right', 'outer', 'inner'}, default 'inner'
on	▪ label or list, 병합 기준 지정 (columns or index level names)
left_on, right_on	▪ label or list, 왼쪽/오른쪽 병합 기준 지정
left_index, right_index	▪ True/False를 사용하여 index를 병합 기준으로 사용할지 여부 지정 ▪ columns가 다를 경우 True로 지정해야 함



# Merge 의 예



## ▶ [예제36] 다음 코드를 보고 병합 방법을 분석하라

```
df = shelve.open("easySample")['sample3']
df1 = df.loc[:, ['pname', 'birth', 'dept']]
df2 = df.loc[:, ['pname', 'english', 'chinese']]

df3 = pd.merge(df1, df2, left_index=True, right_index=True)
df4 = pd.merge(df1, df2, on='pname')
df2.columns = ['NAME', 'english', 'chinese']
df1 = df1.iloc[:6]
df2 = df2.iloc[3:]
df5 = pd.merge(df1, df2, left_on='pname', right_on='NAME')
df6 = pd.merge(df1, df2, how='left', left_on='pname', right_on='NAME')
```

- `left_index, right_index` 를 `True` 설정 : `index`를 병합 기준으로 사용을 의미함  
같은 이름의 열이 없을 때 필수 설정
- `left, right`에 같은 이름이 있을 경우 `on` 을 사용하지 않아도 자동으로 기준으로 사용함
- `left, right`의 기준으로 사용할 이름이 다른 경우 `left_on, right_on`에 이름을 각각 지정함

# 데이터 삭제



## ▶ [예제38] drop 메서드를 사용한 데이터 삭제를 알아보자

x.drop( labels, axis=0, ...)

labels에 전달된 행 또는 열이 삭제된 객체 반환 # x : Series, DataFrame

labels ▪ 한 개의 label 또는 list-like index/column labels

axis ▪ axis=0 or ‘index’ : 행 삭제, axis=1 or ‘columns’ : 열 삭제

s.drop( ‘B’ )		
‘A’ ‘B’ ‘C’ ... 10 13 15 ...	10 13 15 ...	10 15 ...

df.drop( [‘a’, ‘b’] , axis = 1 )		
‘a’ ‘b’ ‘c’ ‘A’ ‘B’ ‘C’ ... 10 13 1 5 20 5 ...	‘a’ ‘b’ ‘c’ 10 13 1 5 20 5 ...	‘c’ 20 5 ...

- s.unique() 는 값의 중복 항을 제거한 ndarray 반환
- 단, 아래의 객체를 포함하면 ExtensionArray 반환
- Categorical, Period, Datetime with Timezone, Interval, Sparse, IntegerNA

# 데이터 추가



## [예제39] append 메서드를 사용한 데이터 추가를 알아보자

• `x.append(other, ignore_index=False, verify_integrity=False, ... )`

• `other`에 전달된 데이터를 추가한 객체 반환 # `x : DataFrame, Series`

other	<ul style="list-style-type: none"><li>▪ <code>x</code> is DataFrame : DataFrame or Series/dict-like, list of these</li><li>▪ <code>x</code> is Series : Series or list/tuple of Series</li></ul>
ignore_index	<ul style="list-style-type: none"><li>▪ True : index labels 사용하지 않음, index 없는 대상 추가 시 필수</li></ul>
verify_integrity	<ul style="list-style-type: none"><li>▪ True : index 중복 시 ValueError 발생</li></ul>

`df1.append(df2), pd.concat([df1, df2])`

The diagram shows two DataFrames, df1 and df2, being concatenated horizontally. df1 has columns 'a' and 'b' with rows 'A' (10, 1) and 'B' (13, 5). df2 has columns 'a' and 'b' with rows 'C' (1, 4) and 'D' (3, 8). An arrow points from df1 to the resulting concatenated DataFrame, which has columns 'a' and 'b' with rows 'A' (10, 1), 'B' (13, 5), 'C' (1, 4), and 'D' (3, 8).

	'a'	'b'
'A'	10	1
'B'	13	5

	'a'	'b'
'C'	1	4
'D'	3	8

`df1.append({'a':1, 'b':4}, ignore_index=True)`

The diagram shows df1 being concatenated with a Series {'a':1, 'b':4} using `ignore_index=True`. df1 has columns 'a' and 'b' with rows 'A' (10, 1) and 'B' (13, 5). A red box labeled '없음' (None) indicates that the index of the Series is ignored. An arrow points from df1 to the resulting DataFrame, which has columns 'a' and 'b' with rows 0 (10, 1), 1 (13, 5), and 2 (1, 4).

	'a'	'b'
'A'	10	1
'B'	13	5

	'a'	'b'
0	10	1
1	13	5
2	1	4

# 데이터 변환 - map



## [예제40] map 메서드를 사용한 데이터 mapping 동작을 알아보자

s.map(arg, na\_action=None)

arg로 전달된 내용이 각 항에 적용된 Series 반환

arg

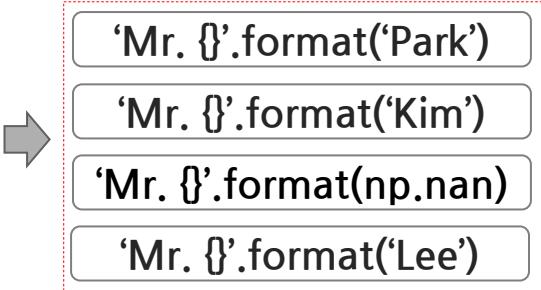
- function, dict, Series
- Series의 각 항에 적용될 내용
- dict가 사용될 경우 key에 없는 것이 Series에 포함되어 있으면 NaN이 됨

na\_action

- {None, 'ignore'} (default None)
- 'ignore' : NA Value에 대해 동작을 무시하고 NaN/None/NaT로 채움

s1=s.map( 'Mr. {}'.format )      s2=s.map( 'Mr. {}'.format, na\_action='ignore' )

0	'Park'
1	'Kim'
2	np.nan
3	'Lee'



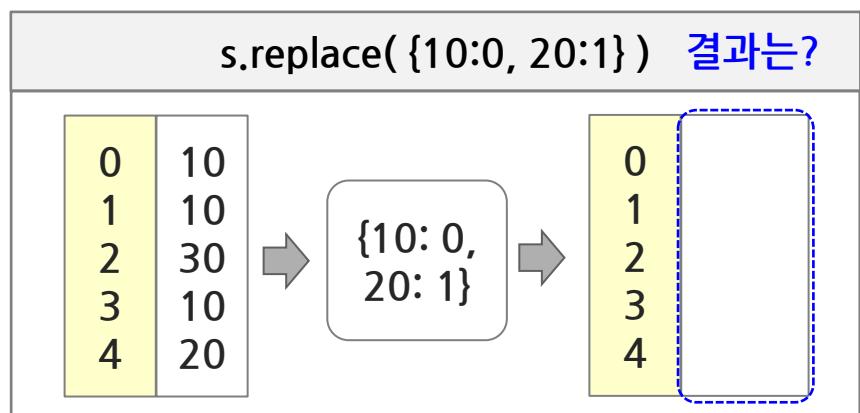
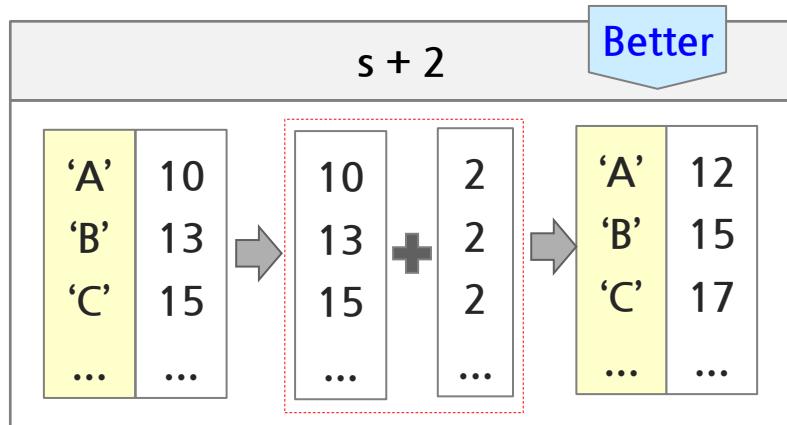
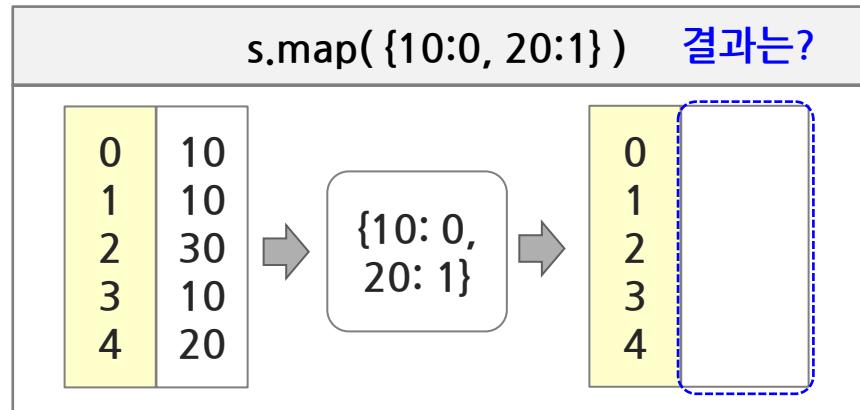
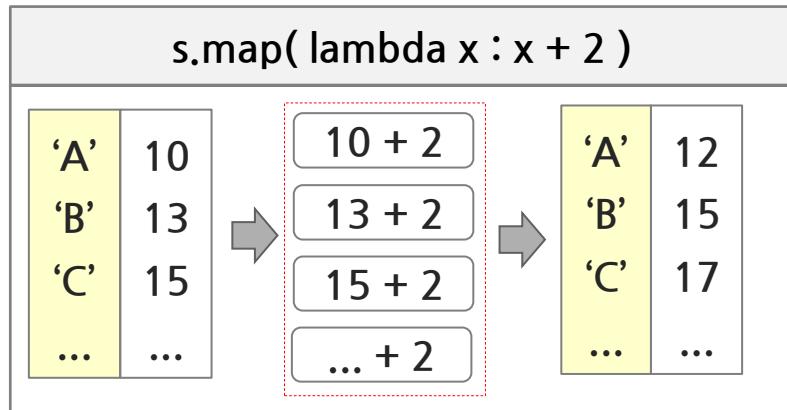
0	Mr. Park
1	Mr. Kim
2	
3	Mr. Lee

{ s1 : Mr. nan  
  s2 : NaN }

# Series.map 동작 이해



👉 그림을 사용하여 map의 동작을 이해하여 보자



s.map 은 dtype이 object, datetime 등의 객체일 때 각 객체의 메서드나 속성 사용시 적합하다

# 데이터 변환 - applymap



## ▶ [예제41] applymap 메서드를 사용한 데이터 변환 작업을 알아보자

df.applymap( func )

각 항의 메서드나 속성을 사용해야 하는 경우 적합

- DataFrame의 각 항에 함수를 적용한 뒤 반환
- 많은 반복을 하게 되므로 가능하면 피하는 것이 좋음
- df.apply() 또는 DataFrame의 연산을 사용하는 것이 성능에 좋음

df.applymap(lambda x : x + 2)

	'a'	'b'	'c'
'A'	10	1	20
'B'	13	5	5
'C'	5	2	15

→

10 + 2	1 + 2	20 + 2
13 + 2	5 + 2	5 + 2
5 + 2	2 + 2	15 + 2

→

	'a'	'b'	'c'
'A'	12	3	22
'B'	15	7	7
'C'	7	4	17

df + 2

	'a'	'b'	'c'
'A'	10	1	20
'B'	13	5	5
'C'	5	2	15

→

10 1 20	+ 2 2 2
13 5 5	2 2 2
5 2 15	2 2 2

→

	'a'	'b'	'c'
'A'	12	3	22
'B'	15	7	7
'C'	7	4	17

# 데이터 변환 - apply



## ▶ [예제42] apply 메서드를 사용한 데이터 변환 작업을 알아보자

• `x.apply( func, axis, ..., args=(), **kwds ) # x : DataFrame, Series`

• axis에 설정에 따라 행/열 별로 func에 주어진 함수를 적용한 결과 반환

func	<ul style="list-style-type: none"><li>각 행이나 열에 적용할 함수</li><li>함수는 lambda로 작성하거나 numpy, Series 등에서 제공되는 것 사용</li><li>apply는 <u>행/열에 함수를 적용</u> 함 (map은 각 항에 함수를 적용)</li></ul>
axis	<ul style="list-style-type: none"><li>0 or 'index' : 각 column에 적용, 1 or 'columns' : 각 row에 적용</li></ul>
args	<ul style="list-style-type: none"><li>array 또는 Series를 포함한 tuple로 작성</li><li>func에 전달할 Positional arguments</li></ul>
**kwargs	<ul style="list-style-type: none"><li>func에 전달할 Keyword arguments</li></ul>

	A	B	C
0	58	74	53
1	89	73	65
2	98	60	80
3	84	52	84

df.apply(sum)	
A	329
B	259
C	282

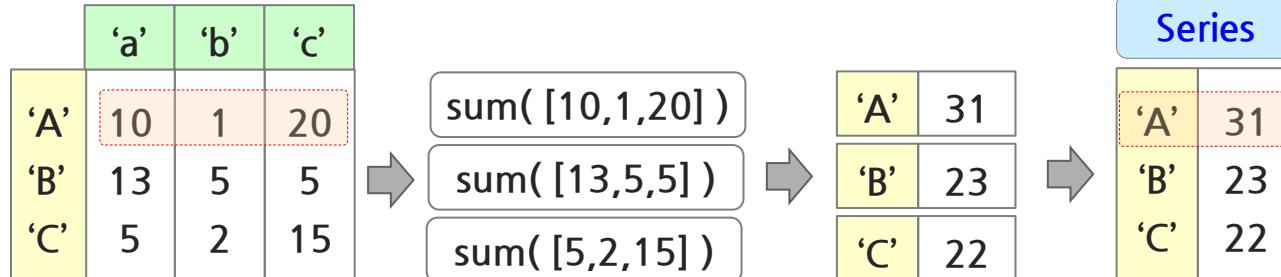
df.apply(min, axis=1)	
0	53
1	65
2	60
3	52

# apply 동작 이해

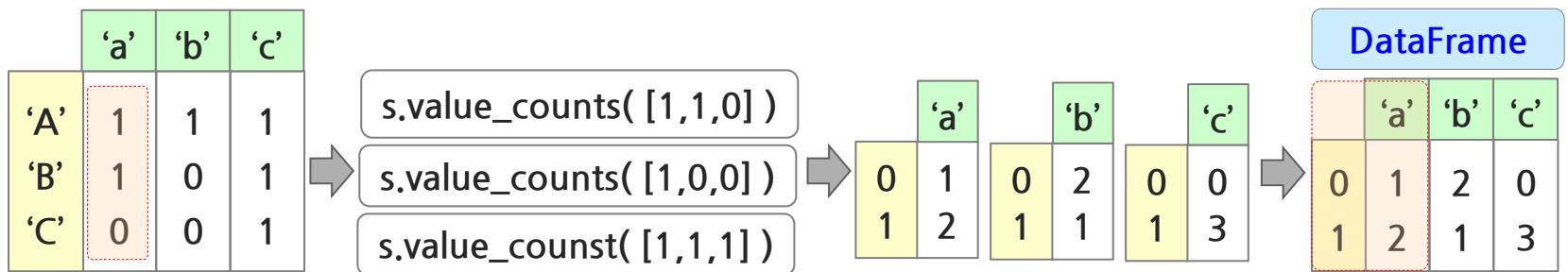


그림을 사용하여 apply 메서드의 동작을 이해하여 보자

`df.apply(sum, axis=1)`   `df.apply(np.sum, axis=1)`   `df.apply(lambda x : x.sum(), axis=1)`



`df.apply( pd.Series.value_counts ), df.apply( lambda x : x.value_counts )`



`df.apply(lambda x : np.where(x>0, 1, 0).sum(), axis=1)` 의 과정을 직접 생각해 보자!

# 데이터의 그룹별 작업



## 데이터를 그룹별 작업은 목적에 따라 세가지로 분류 할 수 있다

Aggregation	각 그룹에 함수 적용 후, 그룹별 함수결과 형태의 객체 반환 예) 그룹별 합계, 평균, 개수 구하기
Transformation	각 그룹에 함수 적용 후, index-like 객체 반환 예) 그룹 내 데이터 표준화, 각 그룹별 산출 값으로 NA Value 채우기
Filtration	각 그룹에 함수 적용 후, 그 결과가 True 인 것만 남김 (False인 것 삭제) 예) 데이터 개수가 적은 그룹 제거, 합계, 평균 등에 기반한 데이터 추출

## 그룹별 작업은 다음 단계를 거쳐 결과를 만들어 낸다

Splitting	그룹 분류 기준에 따라 데이터를 그룹으로 <b>분리</b>
Applying	각 그룹별로 연산 <b>적용</b>
Combining	applying의 결과를 하나의 데이터 구조로 <b>결합</b>

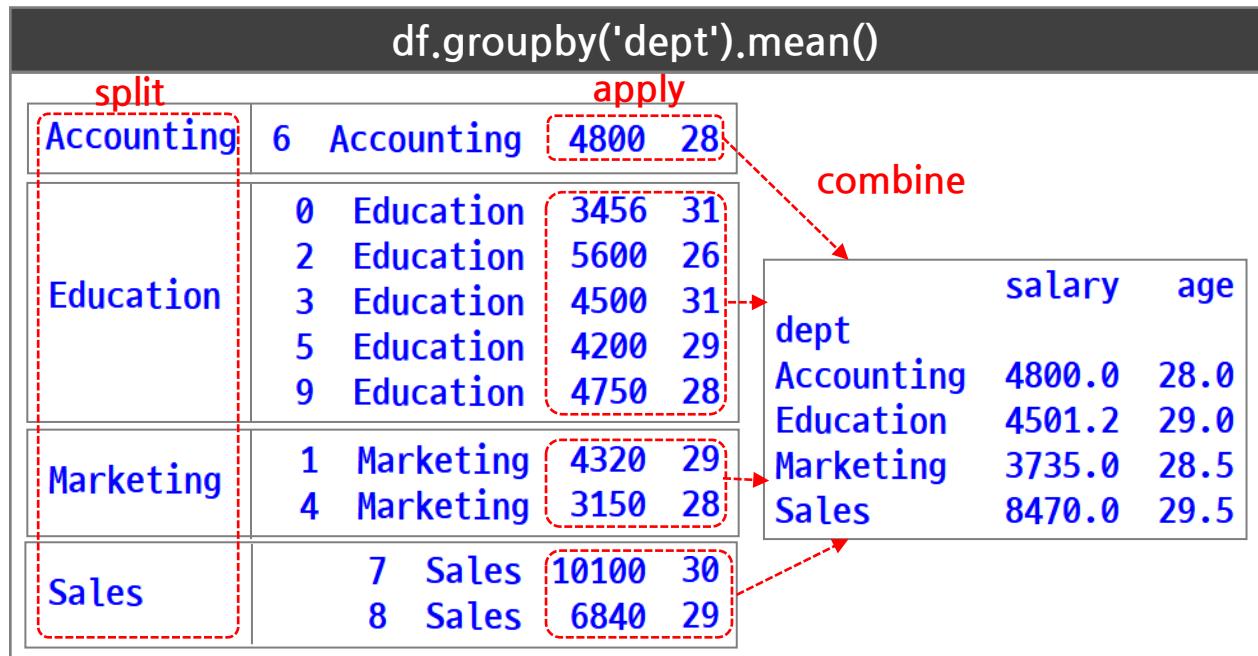
그룹작업    <https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html>  
참조        [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/groupby.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html)

# 그룹별 작업 - 이해 (Aggregation)



▶ [예제43] 그룹별 작업은 **split - apply - combine** 과정으로 동작한다

	dept	salary	age
0	Education	3456	31
1	Marketing	4320	29
2	Education	5600	26
3	Education	4500	31
4	Marketing	3150	28
5	Education	4200	29
6	Accounting	4800	28
7	Sales	10100	30
8	Sales	6840	29
9	Education	4750	28



'sample4' 생성을 위해  
makeSample4 함수를 호출한다  
(실습에 사용할 DataFrame 객체)

df.groupby('dept')['salary'].mean()

Accounting	6	4800
...	...	...
Sales	7	10100
	8	6840

dept	salary	age
Accounting	4800.0	28.0
Education	4501.2	29.0
Marketing	3735.0	28.5
Sales	8470.0	29.5

# 데이터의 그룹별 작업



그룹별 작업의 예를 살펴 보자

함수 형태로 제공되는 각 그룹별 작업 적용 후 반환되는 객체의 형태를 살펴 보자

원본 데이터			
	dept	age	salary
0	Education	31	NaN
1	Marketing	29	4320.0
2	Education	26	5600.0
3	Education	31	4500.0
4	Marketing	28	3150.0
5	Education	29	4200.0
6	Accounting	28	4800.0
7	Sales	30	10100.0
8	Sales	29	6840.0
9	Education	28	4750.0

Aggregation			
dept			
Accounting		4800.0	
Education		4762.5	
Marketing		3735.0	
Sales		8470.0	

Filtration			
dept	age	salary	
Sales	30	10100	
Sales	29	6840	

Transformation			
0	1	2	3
4762.5	3735.0	4762.5	4762.5
4762.5	3735.0	4762.5	4762.5
4762.5	4762.5	4762.5	4762.5
3735.0	3735.0	3735.0	3735.0
4762.5	4762.5	4762.5	4762.5
4800.0	8470.0	8470.0	4762.5
8470.0	8470.0	8470.0	4762.5

Aggregation

그룹별 함수결과 형태의 객체 반환

→ dept별 salary 평균

Transformation

index-like 객체 반환

→ dept별 salary 평균

Filtration

결과가 True 인 것만 남긴 객체 반환

→ dept별 salary 평균이 전체 평균 초과

# GroupBy 객체 생성



▶ DataFrame/Series의 groupby 메서드로 그룹별 분리 작업을 한다

- `df.groupby( by=None, axis=0, level=None, sort=True, as_index=True, ... )`
- `by/level`에 의해 그룹화된 DataFrameGroupBy / SeriesGroupBy 객체 반환

by	<ul style="list-style-type: none"><li>▪ mapping, function, label or list of labels</li><li>▪ function의 경우 객체의 index 각 항을 대상으로 함</li></ul>
axis	<ul style="list-style-type: none"><li>▪ 0인 경우 행, 1인 경우 열 기준으로 그룹 나누기 작업 진행</li></ul>
level	<ul style="list-style-type: none"><li>▪ MultiIndex인 경우 level을 기준으로 그룹 나누기</li></ul>
sort	<ul style="list-style-type: none"><li>▪ 정렬할 것인지 결정하는 것으로 False가 성능면에서 좋음</li></ul>
as_index	<ul style="list-style-type: none"><li>▪ True : group_label을 index로 사용함</li></ul>

DataFrameGroupBy, SeriesGroupBy

- 데이터 그룹별로 tuple을 생산하는 groupby 객체
- Split 동작을 하며, Apply/Combine 동작을 위한 메서드를 가지고 있음

그룹별 작업은 groupby 메서드로 그룹 분리 후, GroupBy 객체의 메서드를 사용하여 처리한다

# 그룹별 작업 - 상세



## [예제44] GroupBy 객체의 구조 및 그룹별 작업 절차에 대해서 알아보자

```
df = df.loc[:, ['dept', 'salary', 'age']]
```

g1 = df.groupby('dept')	g2 = df.groupby('dept')['salary']
<ul style="list-style-type: none"><li>▪ DataFrameGroupBy 객체</li><li>▪ tuple 구성 : (그룹이름, DataFrame)</li></ul>	<ul style="list-style-type: none"><li>▪ SeriesGroupBy 객체</li><li>▪ tuple 구성 : (그룹이름, Series)</li></ul>
 Marketing 1 Marketing 4320 29 4 Marketing 3150 28	 Marketing 1 4320 4 3150 Name: salary, dtype: int32

GroupBy 객체의 메서드 사용으로  
apply/combine 과정을 수행 함

```
df1 = df.groupby('dept').mean()  
s1 = df.groupby('dept')['salary'].mean()
```

- split : 기준으로 그룹을 나눔 (indexing 으로 일부 추출 가능)
- apply : 연산 적용 가능 타입인 열(들)의 각 그룹에 대해 연산 수행 (연산별로 다름)
- combine : 그룹이름을 index, 연산 결과를 columns로 하는 DataFrame/Series 객체로 결합

```
r1 = df.groupby('dept').apply(np.mean)  
r2 = df.groupby('dept')['salary'].apply(np.mean)  
r3 = df.groupby('dept')[['age', 'salary']].mean()  
r4 = df.groupby('dept')[['age']].mean()
```

r1 ~ r4 는 어떤 객체인가?

# GroupBy - Indexing, iteration



▶ [예제45] 다음은 GroupBy의 Indexing, iteration 목록이다

Indexing, iteration	설명
gb.__iter__	▪ iter(gp)를 하면 iterator로 사용 가능
gb.groups	▪ { 그룹이름 : 그룹에 해당하는 index_label들, }의 dict 객체
gb.indices	▪ { 그룹이름 : 그룹에 해당하는 index position들, }의 dict 객체
gb.get_group(index_label)	▪ gb에서 index_label에 해당하는 group 추출 ▪ DataFrame / Series 형태로 반환

```
df = df.loc[:, ['dept', 'salary']]  
df.index = pd.Index(list('ABCDEFGHIJ'))  
gb = df.groupby('dept')  
g_edu = gb.get_group('Sales')  
printobj("GROUP BY : ", gb.groups, gb.indices, g_edu)
```

dept	salary
H	Sales 10100
I	Sales 6840

'Sales': Index(['H', 'I'], dtype='object')

'Sales': array([7, 8], dtype=int64)

# GroupBy - 메서드



GroupBy의 ‘방식-적용객체’별 메서드의 목록이다

Function application 와 Computations/descriptive state 의 두가지로 구분됨

<b>Function application</b> - GroupBy	<ul style="list-style-type: none"><li>▪ apply, agg, aggregate, transform, pipe</li><li>→ argument로 그룹별 적용 동작을 함수를 이용해 전달함</li></ul>
<b>Computations/ descriptive state</b> - GroupBy	<ul style="list-style-type: none"><li>▪ all, any, bfill, count, cumcount, cummax, cummin, cumprod</li><li>▪ cumsum, ffill, first, head, last, max, mean, median, min, ngroup</li><li>▪ nth, oclc, prod, rank, pct_change, size, sem, std, sum, var, tail</li></ul>
- DataFrameGroupBy (SeriesGroupBy 가능)	<ul style="list-style-type: none"><li>▪ all, any, bfill, corr, count, cov, cummax, cummin, cumprod</li><li>▪ cumsum, describe, diff, ffill, fillna, filter, hist, idxmax, idxmin</li><li>▪ mad, nunique, pct_change, plot, quantile, rank, resample, shift</li><li>▪ size, skew, take, tshift</li></ul>
- SeriesGroupBy Only	<ul style="list-style-type: none"><li>▪ nlargest, nsmallest, nunique, unique, value_counts</li><li>▪ is_monotonic_increasing, is_monotonic_decreasing</li></ul>
- DataFrameGroupBy Only	<ul style="list-style-type: none"><li>▪ corrwith, boxplot</li></ul>

SeriesGroupBy에서도 사용할 수 있지만 DataFrameGroupBy와 약간 다르게 적용될 수 있다

# GroupBy 작업 예 1/2



## ▶ [예제46] 다음 groupby 작업을 해석하여 보자

```
df1 = df.loc[:, ['dept', 'salary', 'chinese', 'gender']]  
g1 = df1.groupby('chinese')['chinese'].count()  
g2 = df1.groupby('dept')['salary'].mean()  
g3 = df1.groupby('dept')['gender'].value_counts()
```

```
df2 = df.loc[:, ['dept', 'salary', 'chinese']]  
g1 = df2.groupby(['dept', 'chinese'])['chinese'].count()  
g2 = df2.groupby(['dept', 'chinese']).get_group(('Education', 1))  
#printobj("", next(iter(df.groupby(['dept', 'chinese']))))
```

by가 list인 경우 get\_group( tuple )을 사용해야 함

- count 함수 : 항의 수 (정수) 반환
- mean 함수 : 항들의 평균 (실수) 반환
- value\_counts 함수 : 각 데이터의 개수 (정수) Series 반환

NA Value는 제외하고 연산

# GroupBy 작업 예 2/2



## ▶ [예제46] index와 관련된 groupby 작업을 분석하고, 결과를 찾아보자

x에는 index 객체가 전달 됨 ('age' 가 전달 됨), 함수 반환 값에 따라 group이 나뉨

```
df3 = df.loc[:, ['dept', 'age', 'gender', 'salary']]  
g1 = df3.set_index('age').groupby(lambda x : x//10*10)['gender'].value_counts()  
g2 = df3.set_index(['gender', 'dept']).groupby(level=[1, 0])['salary'].mean()  
g3 = df3.set_index(['gender', 'dept']).groupby(level=[0])['age'].mean()
```

index에서 level 0번을 사용하여 group 분리 ('gender' 기준)

dept	gender	salary
Accounting	F	NaN
	M	4800.000000
Education	F	4475.000000
	M	4518.666667
Marketing	F	3735.000000
	M	NaN
Sales	F	NaN
	M	8470.000000

Name: salary, dtype: float64

gender	count
20 F	4
20 M	3
30 M	3

Name: gender, dtype: int64

gender	mean
F	28.500000
M	29.166667

Name: age, dtype: float64

# GroupBy - apply, agg 개요



## ▶ GroupBy의 apply, agg 메서드로 그룹별 연산 지정하기

### GroupBy.apply(func, \*args, \*\*kwargs)

- func : function
- func 전달 argument : **그룹별 Dataframe/Series 객체**
- 반환 : DataFrame / Series (index 변경 됨, agg보다 속도 느림)

### GroupBy.agg(func, \*args, \*\*kwargs)

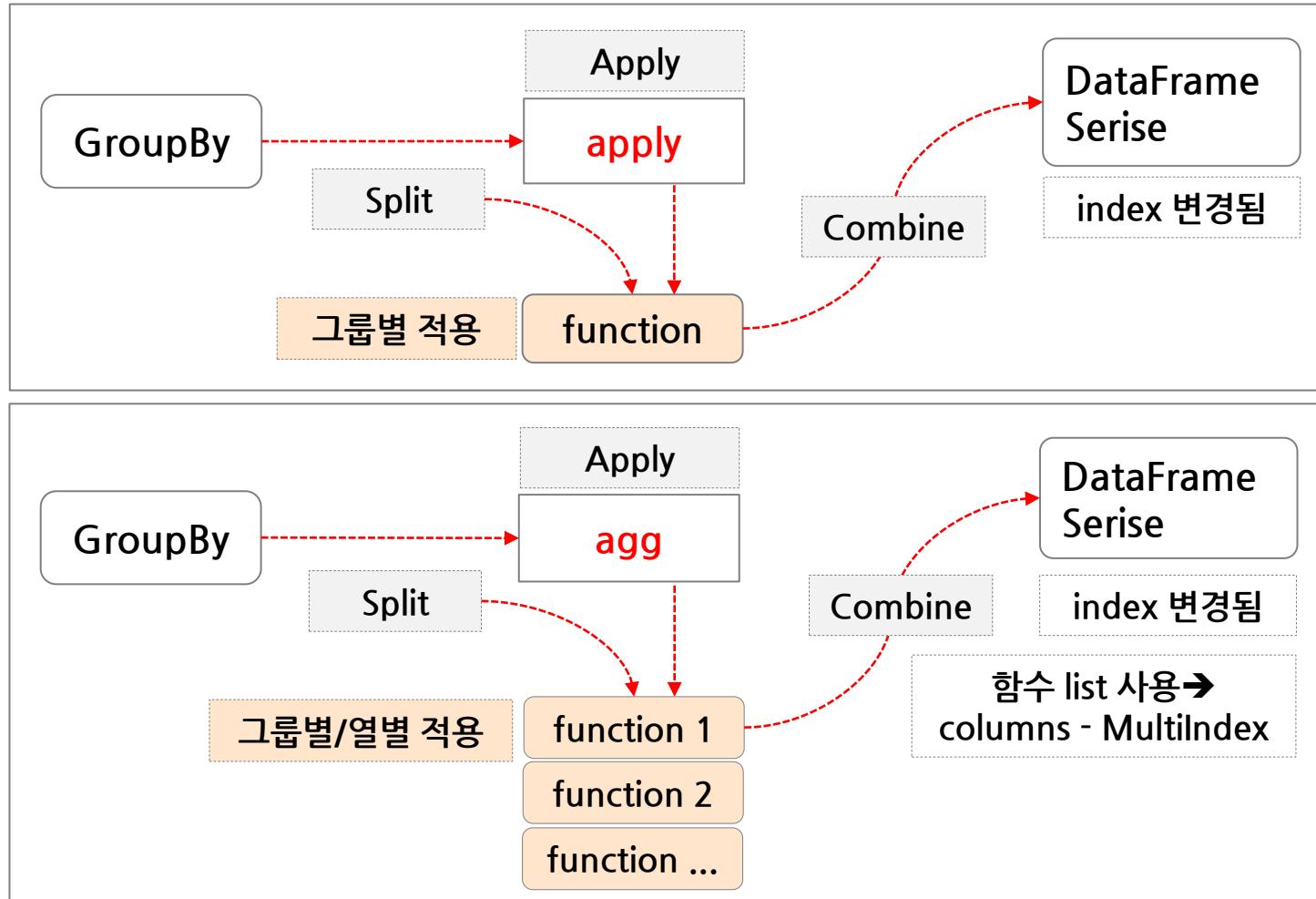
- func : function, str, list or dict
  - str : GroupBy의 메서드 인 경우 문자열 형태로 지정 가능
  - list : 한 개 이상의 함수 목록을 사용할 수 있음
  - dict : 열 별로 적용 함수 목록을 다르게 지정 할 수 있음
- func 전달 argument : **그룹별 DataFrame/Series 객체의 각 열**
- 반환 : DataFrame / Series (index 변경 됨)

- 기본 argument 외에 함수에 추가 전달 arguments 사용 가능
- args : positional arguments 전달
- kwargs : keyword arguments 전달

# GroupBy - apply, agg 이해



## ▶ apply와 agg의 차이를 알아보자



# GroupBy - apply, agg 예



## ▶ [예제47] apply, agg 의 사용 방법을 살펴보자

```
df1 = df.loc[:, ['dept', 'chinese', 'japanese']]  
#grade = lambda x : np.where(x>0, 1, 0).sum()  
grade = lambda x : np.where(x>0, 1, 0)  
g1 = df1.groupby('dept').apply(grade) → DataFrame에 grade 적용  
g2 = df1.groupby('dept').agg(grade) → DataFrame의 각 열에 grade 적용
```

```
df2 = df.loc[:, ['dept', 'chinese', 'japanese']]  
grade = lambda x : np.where(x>0, 1, 0).sum()  
g1 = df2.groupby('dept').agg([grade, max]) → 각 열에 grade 와 max 적용
```

chinese      japanese  
<lambda\_0> max <lambda\_0> max

age      salary  
min max      mean      min      max

```
df3 = df.loc[:, ['dept', 'age', 'salary']]  
g1 = df3.groupby('dept').agg({"age" : [min, max],  
                                  "salary" : [np.mean, min, max]})
```

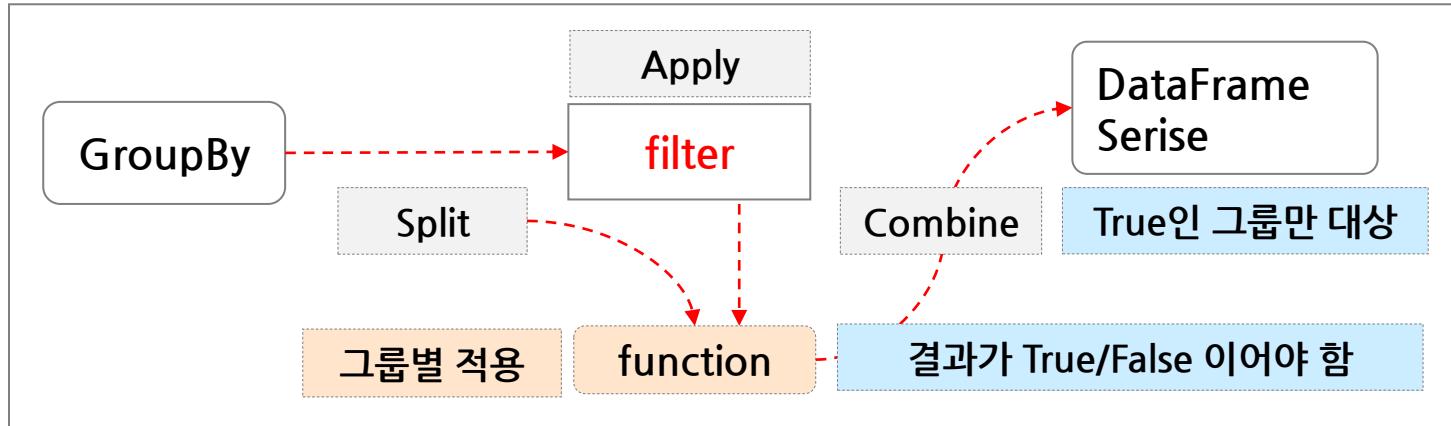
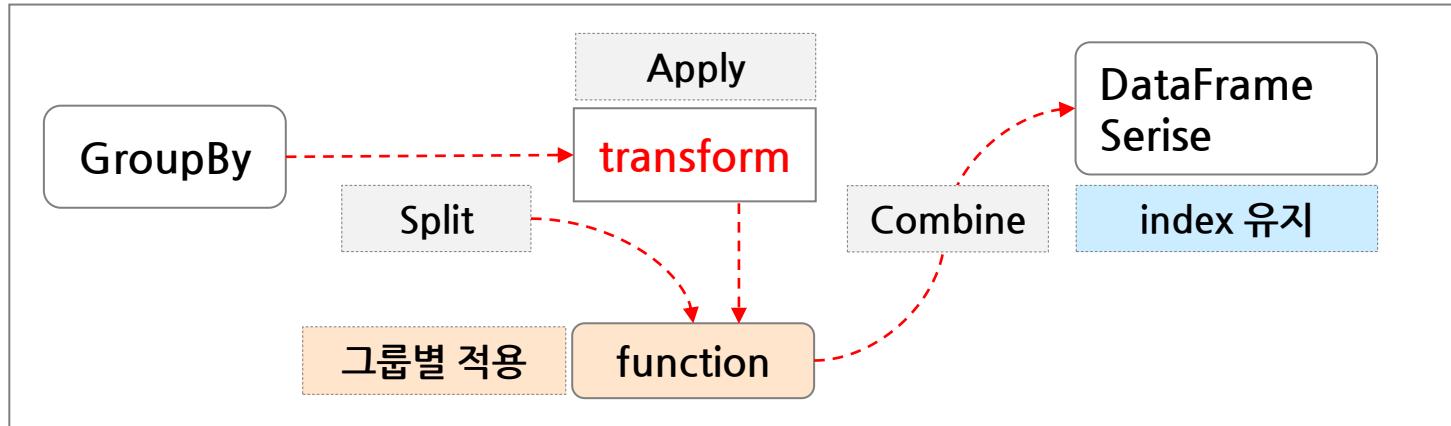
열 별 다른 함수 목록 적용

agg 작성 시 [함수 목록]을 사용할 수 있으며, 이때 columns는 MultiIndex 가 된다

# GroupBy - transform, filter



## transform, filter 의 동작원리를 알아보자



# transform, filter 사용

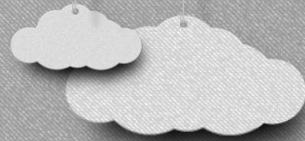


## ▶ [예제48] transform, filter 의 사용방법을 살펴보자

```
df['s_mean'] = df.groupby(['dept', 'sex'])['salary'].transform(np.mean)
df['salary'] = df['salary'].mask(df['salary'].isna(), df['s_mean'])
```

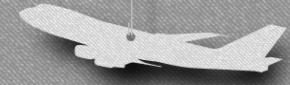
```
df = df.loc[:, ['dept', 'age', 'sex']]
age_group = lambda x : (x//10 *10)
df['age_group'] = df.groupby('dept')['age'].transform(age_group)
```

```
df = df.loc[:, ['dept', 'age', 'sex', 'salary']]
df1 = df.groupby('dept').filter( lambda x : len(x['sex'].unique()) >1 )
```



# live 특강

where, mask, 피벗 테이블



# 데이터 변환 - where, mask



## where, mask 메서드를 사용한 데이터 변환 작업을 알아보자

- `x.where(cond, other = nan, inplace=False, axis=None, ...)` # `x : DataFrame, Series`

cond 값에 따라 값을 유지하거나 other의 값을 취한 뒤 반환

cond	<ul style="list-style-type: none"><li>▪ boolean Series, DataFrame, array-like 및 callable</li><li>▪ True인 것은 값을 유지하고, False인 것은 other에서 값을 취함</li><li>▪ callable 인 경우 boolean Serise/DataFrame을 반환해야 함</li></ul>
other	<ul style="list-style-type: none"><li>▪ scalar, Series/DataFrame 및 callable</li></ul>
axis	<ul style="list-style-type: none"><li>▪ 0 or 'index' : 각 column에 적용, 1 or 'columns' : 각 row에 적용</li></ul>

- `x.mask(cond, other = nan, inplace=False, axis=None, ...)` # `x : DataFrame, Series`

cond 값에 따라 값을 유지하거나 other의 값을 취한 뒤 반환

cond	<ul style="list-style-type: none"><li>▪ False인 것은 값을 유지하고, True인 것은 other에서 값을 취함</li></ul>
------	---

[참고] `np.where(cond, A, B)` : True, False에 대해 각각 대응 값 지정 방식임

# where, mask 동작 이해



👉 그림을 사용하여 where와 mask 동작을 이해하여 보자

`df.where(df>=10, np.nan)`

	'a'	'b'	'c'
'A'	10	1	20
'B'	13	5	5
'C'	5	2	15

df

	'a'	'b'	'c'
'A'	T	F	T
'B'	T	F	F
'C'	F	F	T

df>=10

True/False 값으로 이루어진 bool 타입

→

	'a'	'b'	'c'
'A'	10	NaN	20
'B'	13	NaN	F
'C'	NaN	NaN	15

`df.mask(df< 10, np.nan), df[df< 10] = np.nan`

	'a'	'b'	'c'
'A'	10	1	20
'B'	13	5	5
'C'	5	2	15

df

	'a'	'b'	'c'
'A'	F	T	F
'B'	F	T	T
'C'	T	T	F

df < 10

→

	'a'	'b'	'c'
'A'	10	NaN	20
'B'	13	NaN	F
'C'	NaN	NaN	15

# 피벗 테이블(pivot table)의 이해



- ▶ 피벗 테이블은 데이터를 요약하는 통계표이다
- ▶ 유용한 정보에 집중할 수 있도록 하기 위해 통계를 정렬 또는 재정렬 한다
- ▶ 요약에는 합계, 평균 및 기타 통계가 포함될 수 있다

피벗 테이블 필드

보고서에 추가할 필드 선택:

검색

ID  
 name  
 birth  
 dept  
 english  
 japanese  
 chinese  
 salary  
 overtime  
[기타 테이블...](#)

아래 영역 사이에 필드를 끌어 놓으십시오.

▼ 필터      열

행      Σ 값

dept      평균 : salary

행 레이블	합계 : chinese	합계 : japanese	합계 : english
Accounting	0	0	2
Education	4	4	4
Marketing	0	2	0
Sales	3	2	4
<b>총합계</b>	<b>7</b>	<b>8</b>	<b>10</b>

행 레이블	평균 : salary
Accounting	4800
Education	4501.2
Marketing	3735
Sales	7185
<b>총합계</b>	<b>4914.6</b>

# 피벗 테이블



```
df.pivot_table(values=None, index=None, columns=None, aggfunc='mean',  
               fill_value = None, ...) → DataFrame
```

- spreadsheet 스타일의 pivot table을 형태의 DataFrame을 반환함
- pivot table의 level은 결과 DataFrame의 열과 index의 MultiIndex로 저장됨

values	<ul style="list-style-type: none"><li>▪ columns to aggregate</li><li>▪ 집계 대상 column, 수치 데이터</li></ul>
index	<ul style="list-style-type: none"><li>▪ column, Grouper, array or list of the previous</li><li>▪ 피벗 테이블 행 index가 될 column 명</li></ul>
columns	<ul style="list-style-type: none"><li>▪ column, Grouper, array, or list of the previous</li><li>▪ 피벗 테이블 열 index가 될 column 명</li></ul>
aggfunc	<ul style="list-style-type: none"><li>▪ function, list of functions, dict, default numpy.mean</li><li>▪ 집계함수 지정 - sum, mean, min, max, std, var 등</li></ul>
fill_value	<ul style="list-style-type: none"><li>▪ scalar, default None</li><li>▪ missing value를 채우기 할 값</li></ul>

# 피벗 테이블 예



▶ [예제30] 다음을 실행하여 피벗 테이블을 확인하여 보도록 한다

```
pivot = df.pivot_table(values = ['english', 'japanese', 'chinese'],
                      index = 'dept',
                      aggfunc = np.count_nonzero)
print(pivot, end='\n\n')

pivot = df.pivot_table(values = ['age', 'salary'],
                      index = ['dept'],
                      columns = 'gender',
                      aggfunc = {'age': max,
                                 'salary' : min},
                      fill_value = "0")
print(pivot)
```

dept	chinese	english	japanese	age		salary		
	gender	F	M	F	M	F	M	
dept	Accounting	Education	Marketing	Sales	Accounting	Education	Marketing	Sales
Accounting	0	1	0	2	0	28	0	4800
Education	3	1	2	2	29	31	4200	3456
Marketing	0	0	1	2	29	0	3150	0
Sales	2	2	2	2	0	30	0	6840

# 여러 가지 통계량 출력



## ▶ [예제31] DataFrame 또는 Series의 숫자 형 변수에 대한 통계량 출력

x.describe(percentiles=None, ...)

- DataFrame/Series의 숫자 형 변수에 대한 여러 가지 통계량 출력
- 개수, 평균값, 표준편차, 최솟값, 1, 2 (중앙값), 3사분위 수, 최대값

percentiles

- 새로운 index 생성에 사용할 column/column 목록(columns labels로 작성)

```
data = pd.read_csv('datasets/state-population.csv')
print(data.describe([0.2, 0.5, 0.9]))
```

	year	population
count	2544.000000	2.524000e+03
mean	2001.500000	6.805558e+06
std	6.923547	2.855014e+07
min	1990.000000	1.013090e+05
20%	1994.000000	6.477120e+05
50%	2001.500000	1.597005e+06
90%	2011.000000	9.391577e+06
max	2013.000000	3.161288e+08

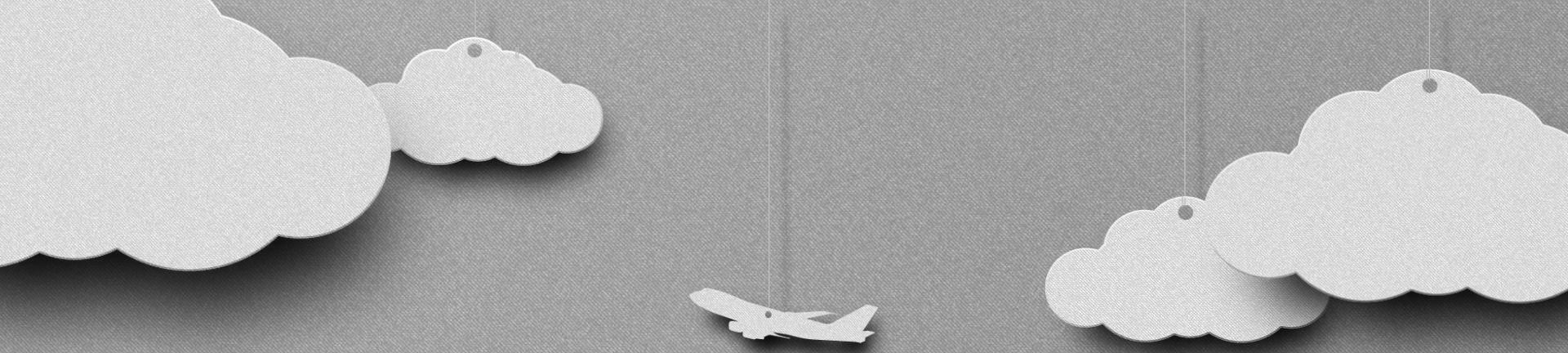
# 그 외 자주 사용되는 메서드



## ▣ 다음은 분석을 위해 자주 사용되는 메서드이다

df.corr	<ul style="list-style-type: none"><li>▪ 상관관계를 분석하는 것으로 -1은 반비례, 1은 정비례를 의미하고 0은 상관관계가 없음을 의미한다</li><li>▪ <a href="https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html">https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html</a></li></ul>
df.quantile	<ul style="list-style-type: none"><li>▪ 백분위수 구하기</li><li>▪ <a href="https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.quantile.html">https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.quantile.html</a></li></ul>

-1.0 ~ -0.7	<ul style="list-style-type: none"><li>▪ 강한 음의 선형관계</li></ul>
-0.7 ~ -0.3	<ul style="list-style-type: none"><li>▪ 뚜렷한 음의 선형관계</li></ul>
-0.3 ~ -0.1	<ul style="list-style-type: none"><li>▪ 약한 음의 선형관계</li></ul>
-0.1 ~ 0.1	<ul style="list-style-type: none"><li>▪ 거의 무시될 수 있는 선형관계</li></ul>
0.1 ~ 0.3	<ul style="list-style-type: none"><li>▪ 약한 양의 선형관계</li></ul>
0.3 ~ 0.7	<ul style="list-style-type: none"><li>▪ 뚜렷한 양의 선형관계</li></ul>
0.7 ~ 1.0	<ul style="list-style-type: none"><li>▪ 강한 양의 선형관계</li></ul>



# Don't worry Be happy!

조금씩 알아 가다 보면 언젠가 많이 알 수 있다!

