작성자 - 고우주 / 데이터쿱와(주)

파이썬 - pandas 기초 실습

1. pands 개요

1.1 pandas 개요

- 금융회사에 다니고 있던 Wes Mckinney가 처음에 금융 데이터 분석을 위해 2008년 설계
- Pandas: 계량 경제학 용어인 panel data와 analysis의 합성어
- 구조화된 데이터를 빠르고 쉬우면서 다양한 형식으로 가공할 수 있는 풍부한 자료 구조와 함수를 제공
- Pandas의 기본 구조는 numpy로

1.2 pandas 특징

- 빅데이터 분석에 최적화 된 필수 패키지
- 데이터는 시계열(series)이나 표(table)의 형태
- 표 데이터를 다루기 위한 시리즈(series) 클래스 변환
- 데이터프레임(dataframe) 클래스 변환

1.3 pandas 설치

- pandas는 기본 패키치로 설치되어 있으나, 만약 별로 설치 한다면,
- 터미널과 cmd에서 설치 시 pip install pandas 실행 또는 Jupyter notebook에서 직접 실행

In [1]:

```
!pip install numpy
```

```
Requirement already satisfied: numpy in /anaconda3/lib/python3.7/site-packages (1.15.4)
You are using pip version 19.0.3, however version 19.1.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

1.4 pandas 패키지 import

- 데이터프레임을 사용하기 위해 pandas 패키지를 임포트 해야한다.
- Pandas는 pd라는 축약어 사용이 관례이다.

In [2]:

```
import pandas as pd
import numpy as np
```

2. 시리즈 클래스 (series class)

2.1 시리즈 정의

- 데이터를 리스트나 1차원 배열 형식으로 series 클래스 생성자에 넣어주면 시리즈 클래스 객체를 만들 수 있다.
- 시리즈 클래스는 NumPy에서 제공하는 1차원 배열과 비슷하지만 각 데이터의 의미를 표시하는 **인덱스(index)**를 붙일 수 있다. 데이터 자체는 **값(value)**라고 한다.

> 시리즈 = 인덱스(index) + 값(value)

In [3]:

```
# Series 정의하여 생성하기
obj = pd.Series([4, 5, -2, 8])
obj
```

Out[3]:

```
0    4
1    5
2   -2
3    8
dtype: int64
```

2.2 시리즈 확인

In [4]:

```
# Series의 값만 확인하기
obj.values
```

Out[4]:

```
array([ 4,  5, -2,  8])
```

In [5]:

```
# series의 인덱스 확인하기
obj.index
```

Out[5]:

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

In [6]:

```
# series의 데이터타입 확인하기
obj.dtypes
```

Out[6]:

```
dtype('int64')
```

2.3 시리즈의 인덱스

- 인덱스의 길이는 데이터의 길이와 같아야 하며, 인덱스의 값은 인덱스 라벨(label)
- 인덱스 라벨은 문자열 뿐 아니라 날짜, 시간, 정수 등도 가능

In [7]:

```
# 인덱스를 리스트로 별도 지정, 반드시 " " 쌍따옴표 사용
obj1 = pd.Series([4, 5, -2, 8], index=["a", "b", "c", "d"])
obj1
```

Out[7]:

```
a    4
b    5
c   -2
d    8
dtype: int64
```

2.4 시리즈와 딕셔너리 자료형

- Python의 dictionary 자료형을 Series data로 만들 수 있다.
- dictionary의 key가 Series의 index가 된다

In [8]:

```
data = {"Kim": 35000, "Park": 67000, "Joon": 12000, "Choi": 4000}
```

In [9]:

```
obj2 = pd.Series(data)
obj2
```

Out[9]:

```
Kim     35000
Park    67000
Joon    12000
Choi     4000
dtype: int64
```

In [10]:

```
# 시리즈 이름 지정 및 index name 지정
obj2.name = "Salary"
obj2.index.name = "Names"

obj2
```

Out[10]:

```
Names
Kim     35000
Park    67000
Joon    12000
Choi     4000
Name: Salary, dtype: int64
```

In [11]:

```
# index 이름 변경
obj2.index = ["A", "B", "C", "D"]
obj2
```

Out[11]:

```
A    35000
B    67000
C    12000
D     4000
Name: Salary, dtype: int64
```

2.5 2.4 시리즈 연산

- Numpy 배열처럼 시리즈도 벡터화 연산 가능
- 시리즈의 값에만 적용되며 인덱스 값은 변하지 않는다

In [12]:

```
obj * 10
```

Out[12]:

```
0    40
1    50
2   -20
3    80
dtype: int64
```

In [13]:

```
# 인덱싱 끼리 연산
obj1 * obj1
```

Out[13]:

```
a    16
b    25
c     4
d    64
dtype: int64
```

In [14]:

```
# values 값끼리 연산
obj1.values + obj.values
```

Out[14]:

```
array([ 8, 10, -4, 16])
```

2.5 시리즈 인덱싱

- 시리즈는 numpy 배열의 인덱스 방법처럼 사용 외에 인덱스 라벨을 이용한 인덱싱
- 배열 인덱싱은 자료의 순서를 바꾸거나 특정한 자료만 선택 가능
- 라벨 값이 영문 문자열인 경우에는 마치 속성인것처럼 점(.)을 이용하여 접근

In [15]:

```
a = pd.Series([1024, 2048, 3096, 6192],
              index=["서울", "부산", "인천", "대구"])
a
```

Out[15]:

```
서울    1024
부산    2048
인천    3096
대구    6192
dtype: int64
```

In [16]:

```
a[1], a["부산"]
```

Out[16]:

```
(2048, 2048)
```

In [17]:

```
a[3], a["대구"]
```

Out[17]:

```
(6192, 6192)
```

In [18]:

```
a[[0, 3, 1]]
```

Out[18]:

```
서울    1024
대구    6192
부산    2048
dtype: int64
```

In [19]:

```
a[["서울", "대구", "부산"]]
```

Out[19]:

```
서울    1024
대구    6192
부산    2048
dtype: int64
```

In [20]:

```
# 라벨 값이 영문 문자열인 경우에는 마치 속성인것처럼 점(.)을 이용하여 접근
obj2.A
```

Out[20]:

```
35000
```

In [21]:

```
obj2.C
```

Out[21]:

```
12000
```

2.6 시리즈 슬라이싱(slicing)

- 배열 인덱싱이나 인덱스 라벨을 이용한 슬라이싱(slicing)도 가능
- 문자열 라벨을 이용한 슬라이싱은 콜론(:) 기호 뒤에 오는 인덱스에 해당하는 값이 결과에 포함

In [22]:

```
a[1:3]
```

Out[22]:

```
부산    2048
인천    3096
dtype: int64
```

In [23]:

```
a["부산":"대구"]
```

Out[23]:

```
부산    2048
인천    3096
대구    6192
dtype: int64
```

2.7 시리즈의 데이터 갱신, 추가, 삭제

- 인덱싱을 이용하여 딕셔너리처럼 데이터를 갱신(update)하거나 추가(add)
- 데이터 삭제 시 딕셔너리처럼 del 명령 사용

In [24]:

```
# 데이터 갱신
a["부산"] = 1234
a
```

Out[24]:

```
서울    1024
부산    1234
인천    3096
대구    6192
dtype: int64
```

In [25]:

```
# 데이터 갱신
a["대구"] = 6543
a
```

Out[25]:

```
서울    1024
부산    1234
인천    3096
대구    6543
dtype: int64
```

In [26]:

```
# del 명령어로 데이터 삭제
del a["서울"]
a
```

Out[26]:

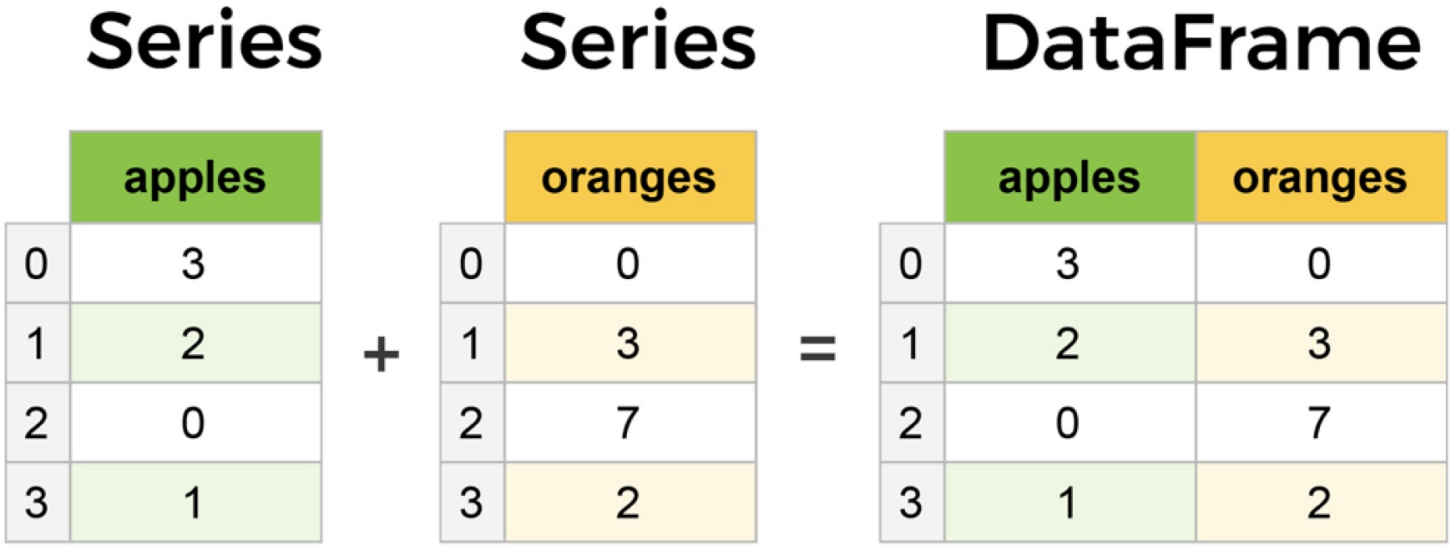
```
부산    1234
인천    3096
대구    6543
dtype: int64
```

3. 데이터프레임(DataFrame)

3.1 데이터프레임(DataFrame) 개요

- 시리즈가 1차원 벡터 데이터에 행 방향 인덱스(row index)이라면,
- 데이터프레임(data-frame) 클래스는 2차원 행렬 데이터에 합친 것으로
- 행 인덱스(row index)와 열 인덱스(column index)를 지정

> 데이터프레임 = 시리즈{인덱스(index) + 값(value)} + 시리즈 + 시리즈의 연속체



3.2 데이터프레임 특성

- 데이터프레임은 공통 인덱스를 가지는 열 시리즈(column series)를 딕셔너리로 묶어놓은 것
- 데이터프레임은 numpy의 모든 2차원 배열 속성이나 메서드를 지원

3.3 데이터프레임 생성

1. 우선 하나의 열이 되는 데이터를 리스트나 일차원 배열을 준비
2. 각 열에 대한 이름(label)의 키(key)를 갖는 딕셔너리를 생성
3. pandas의 DataFrame 클래스로 생성
4. 열방향 인덱스는 columns 인수로, 행방향 인덱스는 index 인수로 지정

In [27]:

```
# Data Frame은 python의 dictionary 또는 numpy의 array로 정의
data = {
'name': ["Choi", "Choi", "Choi", "Kim", "Park"],
'year': [2013, 2014, 2015, 2016, 2017],
'points': [1.5, 1.7, 3.6, 2.4, 2.9]
}
df = pd.DataFrame(data)
df
```

Out[27]:

	name	year	points
0	Choi	2013	1.5
1	Choi	2014	1.7
2	Choi	2015	3.6
3	Kim	2016	2.4
4	Park	2017	2.9

In [28]:

```
# 행 방향의 index
df.index
```

Out[28]:

```
RangeIndex(start=0, stop=5, step=1)
```

In [29]:

```
# 열 방향의 index
df.columns
```

Out[29]:

```
Index(['name', 'year', 'points'], dtype='object')
```

In [30]:

```
df.values
```

Out[30]:

```
array([['Choi', 2013, 1.5],
       ['Choi', 2014, 1.7],
       ['Choi', 2015, 3.6],
       ['Kim', 2016, 2.4],
       ['Park', 2017, 2.9]], dtype=object)
```

3.4 데이터프레임 열 갱신 추가

- 데이터프레임은 열 시리즈의 딕셔너리로 볼 수 있으므로 열 단위로 데이터를 갱신하거나 추가, 삭제
- data에 포함되어 있지 않은 값은 nan(not a number)으로 나타내는 null과 같은 개념
- 딕셔너리, numpy의 배열, 시리즈의 다양한 방법으로 추가 가능

In [31]:

```
# DataFrame을 만들면서 columns와 index를 설정
df = pd.DataFrame(data, columns=["year", "name", "points", "penalty"],
                                    index=(["one", "two", "three", "four", "five"]))
df
```

Out[31]:

	year	name	points	penalty
one	2013	Choi	1.5	NaN
two	2014	Choi	1.7	NaN
three	2015	Choi	3.6	NaN
four	2016	Kim	2.4	NaN
five	2017	Park	2.9	NaN

In [32]:

```
# 특정 열만 선택
df[["year","points"]]
```

Out[32]:

	year	points
one	2013	1.5
two	2014	1.7
three	2015	3.6
four	2016	2.4
five	2017	2.9

In [33]:

```
# 특정 열을 선택하고, 값(0.5)을 대입
df["penalty"] = 0.5
df
```

Out[33]:

	year	name	points	penalty
one	2013	Choi	1.5	0.5
two	2014	Choi	1.7	0.5
three	2015	Choi	3.6	0.5
four	2016	Kim	2.4	0.5
five	2017	Park	2.9	0.5

In [34]:

```
# 또는 python의 리스트로 대입
df['penalty'] = [0.1, 0.2, 0.3, 0.4, 0.5]
df
```

Out[34]:

	year	name	points	penalty
one	2013	Choi	1.5	0.1
two	2014	Choi	1.7	0.2
three	2015	Choi	3.6	0.3
four	2016	Kim	2.4	0.4
five	2017	Park	2.9	0.5

In [35]:

```
import numpy as np

# 또는 numpy의 np.arange로 새로운 열을 추가하기
df['zeros'] = np.arange(5)
df
```

Out[35]:

	year	name	points	penalty	zeros
one	2013	Choi	1.5	0.1	0
two	2014	Choi	1.7	0.2	1
three	2015	Choi	3.6	0.3	2
four	2016	Kim	2.4	0.4	3
five	2017	Park	2.9	0.5	4

In [36]:

```
# 또는 index인자로 특정행을 지정하여 시리즈(Series)로 추가
val = pd.Series([-1.2, -1.5, -1.7], index=['two','four','five'])

df['debt'] = val
df
```

Out[36]:

	year	name	points	penalty	zeros	debt
one	2013	Choi	1.5	0.1	0	NaN
two	2014	Choi	1.7	0.2	1	-1.2
three	2015	Choi	3.6	0.3	2	NaN
four	2016	Kim	2.4	0.4	3	-1.5
five	2017	Park	2.9	0.5	4	-1.7

In [37]:

```
# 연산 후 새로운 열을 추가하기
df["net_points"] = df["points"] - df["penalty"]
df
```

Out[37]:

	year	name	points	penalty	zeros	debt	net_points
one	2013	Choi	1.5	0.1	0	NaN	1.4
two	2014	Choi	1.7	0.2	1	-1.2	1.5
three	2015	Choi	3.6	0.3	2	NaN	3.3
four	2016	Kim	2.4	0.4	3	-1.5	2.0
five	2017	Park	2.9	0.5	4	-1.7	2.4

In [38]:

```
# 조건 연산으로 열 추가
df["high_points"] = df["net_points"] > 2.0
df
```

Out[38]:

	year	name	points	penalty	zeros	debt	net_points	high_points
one	2013	Choi	1.5	0.1	0	NaN	1.4	False
two	2014	Choi	1.7	0.2	1	-1.2	1.5	False
three	2015	Choi	3.6	0.3	2	NaN	3.3	True
four	2016	Kim	2.4	0.4	3	-1.5	2.0	False
five	2017	Park	2.9	0.5	4	-1.7	2.4	True

In [39]:

```
# 열 삭제하기
del df["high_points"]
del df["net_points"]
del df["zeros"]
df
```

Out[39]:

	year	name	points	penalty	debt
one	2013	Choi	1.5	0.1	NaN
two	2014	Choi	1.7	0.2	-1.2
three	2015	Choi	3.6	0.3	NaN
four	2016	Kim	2.4	0.4	-1.5
five	2017	Park	2.9	0.5	-1.7

In [40]:

```
# 컬럼명 확인하기
df.columns
```

Out[40]:

```
Index(['year', 'name', 'points', 'penalty', 'debt'], dtype='object')
```

In [41]:

```
# index와 columns 이름 지정
df.index.name = "Order"
df.columns.name = "Info"
df
```

Out[41]:

Info	year	name	points	penalty	debt
Order					
one	2013	Choi	1.5	0.1	NaN
two	2014	Choi	1.7	0.2	-1.2
three	2015	Choi	3.6	0.3	NaN
four	2016	Kim	2.4	0.4	-1.5
five	2017	Park	2.9	0.5	-1.7

3.4 데이터프레임 인덱싱

열 인덱싱

- 데이터프레임을 인덱싱을 할 때도 열 라벨(column label)을 키 값으로 생각하여 인덱싱
- 인덱스로 라벨 값을 하나만 넣으면 시리즈 객체가 반환되고 라벨의 배열 또는 리스트를 넣으면 부분적인 데이터프레임이 반환
- 하나의 열만 빼내면서 데이터프레임 자료형을 유지하고 싶다면 원소가 하나인 리스트를 써서 인덱싱

행 인덱싱

- 행 단위로 인덱싱을 하고자 하면 항상 슬라이싱(slicing)을 해야 한다.
- 인덱스의 값이 문자 라벨이면 라벨 슬라이싱

In [42]:

```
# 열 인덱싱
df["year"]
```

Out[42]:

```
Order
one      2013
two      2014
three    2015
four     2016
five     2017
Name: year, dtype: int64
```

In [43]:

```
# 다른 방법의 열 인덱싱
df.year
```

Out[43]:

```
Order
one      2013
two      2014
three    2015
four     2016
five     2017
Name: year, dtype: int64
```

In [44]:

```
# 행 인덱싱은 슬라이싱으로 0번째부터 1번째로 지정하면 1행을 반환
df[0:1]
```

Out[44]:

Info	year	name	points	penalty	debt
Order					
one	2013	Choi	1.5	0.1	NaN

In [45]:

```
# 행 인덱싱 슬라이싱으로 0번째 부터 2(3-1) 번째까지 반환
df[0:3]
```

Out[45]:

Info	year	name	points	penalty	debt
Order					
one	2013	Choi	1.5	0.1	NaN
two	2014	Choi	1.7	0.2	-1.2
three	2015	Choi	3.6	0.3	NaN

3.5 loc 인덱싱

인덱스의 라벨값 기반의 2차원 (행, 열)인덱싱

In [46]:

```
# .loc 함수를 사용하여 시리즈로 인덱싱
df.loc["two"]
```

Out[46]:

```
Info
year       2014
name       Choi
points      1.7
penalty     0.2
debt       -1.2
Name: two, dtype: object
```

In [47]:

```
# .loc 또는 .iloc 함수를 사용하여 데이터프레임으로 인덱싱
df.loc["two":"four"]
```

Out[47]:

Info	year	name	points	penalty	debt
Order					
two	2014	Choi	1.7	0.2	-1.2
three	2015	Choi	3.6	0.3	NaN
four	2016	Kim	2.4	0.4	-1.5

In [48]:

```
df.loc["two":"four", "points"]
```

Out[48]:

```
Order
two      1.7
three    3.6
four     2.4
Name: points, dtype: float64
```

In [49]:

```
# == df['year']
df.loc[:,'year']
```

Out[49]:

```
Order
one      2013
two      2014
three    2015
four     2016
five     2017
Name: year, dtype: int64
```

In [65]:

```
df.loc[:,['year','names']]
```

```
/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py:1472: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-listlike
  return self._getitem_tuple(key)
```

Out[65]:

Info	year	names
Order		
one	2013.0	NaN
two	2014.0	NaN
three	2015.0	NaN
four	2016.0	NaN
five	2017.0	NaN
six	2013.0	NaN

In [51]:

```
df.loc["three":"five","year":"penalty"]
```

Out[51]:

Info	year	name	points	penalty
Order				
three	2015	Choi	3.6	0.3
four	2016	Kim	2.4	0.4
five	2017	Park	2.9	0.5

3.6 iloc 인덱싱

인덱스의 숫자 기반의 2차원 (행, 열)인덱싱

In [52]:

```
# 새로운 행 삽입하기
df.loc['six',:] = [2013,'Jun',4.0,0.1,2.1]
df
```

Out[52]:

Info	year	name	points	penalty	debt
Order					
one	2013.0	Choi	1.5	0.1	NaN
two	2014.0	Choi	1.7	0.2	-1.2
three	2015.0	Choi	3.6	0.3	NaN
four	2016.0	Kim	2.4	0.4	-1.5
five	2017.0	Park	2.9	0.5	-1.7
six	2013.0	Jun	4.0	0.1	2.1

In [53]:

```
# 4번째 행을 가져오기 위해 .iloc 사용:: index 번호를 사용
df.iloc[3]
```

Out[53]:

```
Info
year       2016
name        Kim
points      2.4
penalty     0.4
debt       -1.5
Name: four, dtype: object
```

In [54]:

```
# 슬라이싱으로 지정하여 반환
df.iloc[3:5, 0:2]
```

Out[54]:

Info	year	name
Order		
four	2016.0	Kim
five	2017.0	Park

In [55]:

```
# 각각의 행과 열을 지정하여 반환하기
df.iloc[[0,1,3], [1,2]]
```

Out[55]:

Info	name	points
Order		
one	Choi	1.5
two	Choi	1.7
four	Kim	2.4

In [56]:

```
# 행을 전체, 열은 두번째열부터 마지막까지 슬라이싱으로 지정하여 반환
df.iloc[:,1:4]
```

Out[56]:

Info	name	points	penalty
Order			
one	Choi	1.5	0.1
two	Choi	1.7	0.2
three	Choi	3.6	0.3
four	Kim	2.4	0.4
five	Park	2.9	0.5
six	Jun	4.0	0.1

In [66]:

```
df.iloc[1,1]
```

Out[66]:

```
'Choi'
```

3.7 Boolean 인덱싱

In [57]:

```
df
```

Out[57]:

Info	year	name	points	penalty	debt
Order					
one	2013.0	Choi	1.5	0.1	NaN
two	2014.0	Choi	1.7	0.2	-1.2
three	2015.0	Choi	3.6	0.3	NaN
four	2016.0	Kim	2.4	0.4	-1.5
five	2017.0	Park	2.9	0.5	-1.7
six	2013.0	Jun	4.0	0.1	2.1

In [58]:

```
# year가 2014보다 큰 boolean data
df["year"] > 2014
```

Out[58]:

```
Order
one      False
two      False
three     True
four      True
five      True
six      False
Name: year, dtype: bool
```

In [71]:

```
df.loc[df['name'] == "Choi", ['name','points']]
```

Out[71]:

Info	name	points
Order		
one	Choi	1.5
two	Choi	1.7
three	Choi	3.6

In [72]:

```
# numpy에서와 같이 논리연산을 응용할 수 있다.
df.loc[(df["points"] > 2) & (df["points"]<3), :]
```

Out[72]:

Info	year	name	points	penalty	debt
Order					
four	2016.0	Kim	2.4	0.4	-1.5
five	2017.0	Park	2.9	0.5	-1.7

4. 데이터프레임 다루기

4.1 numpy randn 데이터프레임 생성

In [73]:

```
# DataFrame을 만들때 index, column을 설정하지 않으면 기본값으로 0부터 시작하는 정수형 숫자로 입력된다.
df = pd.DataFrame(np.random.randn(6,4))
df
```

Out[73]:

	0	1	2	3
0	1.309472	0.851566	2.052958	-0.184364
1	-0.056565	-2.466590	-1.390127	-0.279191
2	2.132569	-0.414397	0.689673	0.228209
3	1.117153	0.778170	1.099894	0.878116
4	-0.099149	1.013096	-2.122878	-0.853181
5	-0.620518	-1.440219	-0.662198	0.586739

4.2 시계열 데이트 함수 date_range

In [74]:

```
df.columns = ["A", "B", "C", "D"]
```

In [75]:

```
#pandas에서 제공하는 date range함수는 datetime 자료형으로 구성된, 날짜/시간 함수
df.index = pd.date_range('20160701', periods=6)
df
```

Out[75]:

	A	B	C	D
2016-07-01	1.309472	0.851566	2.052958	-0.184364
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191
2016-07-03	2.132569	-0.414397	0.689673	0.228209
2016-07-04	1.117153	0.778170	1.099894	0.878116
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739

In [76]:

```
df.index
```

Out[76]:

```
DatetimeIndex(['2016-07-01', '2016-07-02', '2016-07-03', '2016-07-04',
               '2016-07-05', '2016-07-06'],
              dtype='datetime64[ns]', freq='D')
```

In [77]:

```
df
```

Out[77]:

	A	B	C	D
2016-07-01	1.309472	0.851566	2.052958	-0.184364
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191
2016-07-03	2.132569	-0.414397	0.689673	0.228209
2016-07-04	1.117153	0.778170	1.099894	0.878116
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739

4.3 numpy로 데이터프레임 결측치 다루기

In [78]:

```
# np.nan은 NaN값을 의미
df["F"] = [1.0, np.nan, 3.5, 6.1, np.nan, 7.0]
df
```

Out[78]:

	A	B	C	D	F
2016-07-01	1.309472	0.851566	2.052958	-0.184364	1.0
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191	NaN
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-04	1.117153	0.778170	1.099894	0.878116	6.1
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181	NaN
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

In [79]:

```
# 행의 값중 하나라도 nan인 경우 그 행을 없앤다.
df.dropna(how="any")
```

Out[79]:

	A	B	C	D	F
2016-07-01	1.309472	0.851566	2.052958	-0.184364	1.0
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-04	1.117153	0.778170	1.099894	0.878116	6.1
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

In [80]:

```
# 행의 값의 모든 값이 nan인 경우 그 행을 없앤다.
df.dropna(how='all')
```

Out[80]:

	A	B	C	D	F
2016-07-01	1.309472	0.851566	2.052958	-0.184364	1.0
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191	NaN
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-04	1.117153	0.778170	1.099894	0.878116	6.1
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181	NaN
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

In [81]:

```
# NaN에 특정 value 값 넣기
df.fillna(value=0.5)
```

Out[81]:

	A	B	C	D	F
2016-07-01	1.309472	0.851566	2.052958	-0.184364	1.0
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191	0.5
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-04	1.117153	0.778170	1.099894	0.878116	6.1
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181	0.5
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

4.4 drop 명령어

In [82]:

```
# 특정 행 drop하기
df.drop(pd.to_datetime('20160701'))
```

Out[82]:

	A	B	C	D	F
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191	NaN
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-04	1.117153	0.778170	1.099894	0.878116	6.1
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181	NaN
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

In [83]:

```
# 2개 이상도 가능
df.drop([pd.to_datetime('20160702'),pd.to_datetime('20160704')])
```

Out[83]:

	A	B	C	D	F
2016-07-01	1.309472	0.851566	2.052958	-0.184364	1.0
2016-07-03	2.132569	-0.414397	0.689673	0.228209	3.5
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181	NaN
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739	7.0

In [84]:

```
# 특정 열 삭제하기
df.drop('F', axis = 1)
```

Out[84]:

	A	B	C	D
2016-07-01	1.309472	0.851566	2.052958	-0.184364
2016-07-02	-0.056565	-2.466590	-1.390127	-0.279191
2016-07-03	2.132569	-0.414397	0.689673	0.228209
2016-07-04	1.117153	0.778170	1.099894	0.878116
2016-07-05	-0.099149	1.013096	-2.122878	-0.853181
2016-07-06	-0.620518	-1.440219	-0.662198	0.586739

In [85]:

```
# 2개 이상의 열도 가능
df.drop(['B','D'], axis = 1)
```

Out[85]:

	A	C	F
2016-07-01	1.309472	2.052958	1.0
2016-07-02	-0.056565	-1.390127	NaN
2016-07-03	2.132569	0.689673	3.5
2016-07-04	1.117153	1.099894	6.1
2016-07-05	-0.099149	-2.122878	NaN
2016-07-06	-0.620518	-0.662198	7.0

5. pandas 데이터 입출력

- pandas는 데이터 분석을 위해 여러 포맷의 데이터 파일을 읽고 쓸수 있다.
- csv, excel, html, json, hdf5, sas, stata, sql

5.1 pandas 데이터 불러오기

In [123]:

```
pd.read_csv('data/sample1.csv')
```

Out[123]:

	c1	c2	c3
0	1	1.11	one
1	2	2.22	two
2	3	3.33	three

In [124]:

```
# c1을 인덱스로 불러오기
pd.read_csv('data/sample1.csv', index_col="c1")
```

Out[124]:

c1	c2	c3
1	1.11	one
2	2.22	two
3	3.33	three

5.2 pandas 데이터 쓰기

In [121]:

```
df.to_csv("data/sample6.csv")
```

In [125]:

```
df.to_csv("data/sample9.csv", index=False, header=False)
```

5.3 인터넷에서 데이터 불러오기

In [127]:

```
# 인터넷 링크의 데이터 불러오기
titanic = pd.read_excel("http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls")
titanic.head()
```

Out[127]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	S	NaN	135.0	Montreal, PQ / Chesterville, ON
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

In [129]:

```
titanic.head(3)
```

Out[129]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

In [130]:

```
titanic.tail()
```

Out[130]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
1304	3	0	Zabour, Miss. Hileni	female	14.5	1	0	2665	14.4542	NaN	C	NaN	328.0	NaN
1305	3	0	Zabour, Miss. Thamine	female	NaN	1	0	2665	14.4542	NaN	C	NaN	NaN	NaN
1306	3	0	Zakarian, Mr. Mapriededer	male	26.5	0	0	2656	7.2250	NaN	C	NaN	304.0	NaN
1307	3	0	Zakarian, Mr. Ortin	male	27.0	0	0	2670	7.2250	NaN	C	NaN	NaN	NaN
1308	3	0	Zimmerman, Mr. Leo	male	29.0	0	0	315082	7.8750	NaN	S	NaN	NaN	NaN

6. 데이터 처리하기

6.1 정렬(Sort)

- 데이터를 정렬로 sort_index는 인덱스 값을 기준으로,
- sort_values는 데이터 값을 기준으로 정렬

In [91]:

```
# np.random으로 시리즈 생성
s = pd.Series(np.random.randint(6, size=100))
s.head()
```

Out[91]:

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

In [92]:

```
# value_counts 메서드로 값을 카운트
s.value_counts()
```

Out[92]:

```
0    22
2    19
5    17
4    15
3    15
1    12
dtype: int64
```

In [93]:

```
# sort_index 메서드로 정렬하기
s.value_counts().sort_index()
```

Out[93]:

```
0    22
1    12
2    19
3    15
4    15
5    17
dtype: int64
```

In [94]:

```
# ascending=False 인자로 내림차순 정리
s.sort_values(ascending=False)
```

Out[94]:

```
99    5
84    5
2     5
4     5
13    5
14    5
25    5
26    5
29    5
52    5
67    5
76    5
78    5
0     5
91    5
87    5
90    5
42    4
7     4
27    4
30    4
95    4
12    4
43    4
44    4
10    4
98    4
54    4
56    4
18    4
     ..
55    1
8     1
39    1
53    1
97    1
33    1
3     1
31    1
15    0
94    0
16    0
6     0
85    0
21    0
62    0
82    0
74    0
71    0
68    0
89    0
66    0
61    0
93    0
59    0
46    0
86    0
36    0
34    0
32    0
49    0
Length: 100, dtype: int64
```

6.2 apply 함수

- 행이나 열 단위로 더 복잡한 처리를 하고 싶을 때는 apply 메서드를 사용
- 인수로 행 또는 열을 받는 함수를 apply 메서드의 인수로 넣으면 각 열(또는 행)을 반복하여 수행

lambda 함수:

- 파이썬에서 "lambda" 는 런타임에 생성해서 사용할 수 있는 익명 함수
- lambda는 쓰고 버리는 일시적인 함수로 생성된 곳에서만 적용

In [95]:

```
df = pd.DataFrame({
    'A': [1, 3, 4, 3, 4],
    'B': [2, 3, 1, 2, 3],
    'C': [1, 5, 2, 4, 4]
})
df
```

Out[95]:

	A	B	C
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

In [96]:

```
# 람다 함수 사용
df.apply(lambda x: x.max() - x.min())
```

Out[96]:

```
A    3
B    2
C    4
dtype: int64
```

In [97]:

```
# 만약 행에 대해 적용하고 싶으면 axis=1 인수 사용
df.apply(lambda x: x.max() - x.min(), axis=1)
```

Out[97]:

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

In [98]:

```
# apply로 value_counts로 값의 수를 반환
df.apply(pd.value_counts)
```

Out[98]:

	A	B	C
1	1.0	1.0	1.0
2	NaN	2.0	1.0
3	2.0	2.0	NaN
4	2.0	NaN	2.0
5	NaN	NaN	1.0

In [99]:

```
# NaN 결측치에 fillna(0)으로 0을 채우고 순차적으로 정수로 변환
df.apply(pd.value_counts).fillna(0).astype(int)
```

Out[99]:

	A	B	C
1	1	1	1
2	0	2	1
3	2	2	0
4	2	0	2
5	0	0	1

6.3 describe 메서드

describe() 함수는 DataFrame의 계산 가능한 값들의 통계값을 보여준다

In [100]:

```
df.describe()
```

Out[100]:

	A	B	C
count	5.000000	5.00000	5.000000
mean	3.000000	2.20000	3.200000
std	1.224745	0.83666	1.643168
min	1.000000	1.00000	1.000000
25%	3.000000	2.00000	2.000000
50%	3.000000	2.00000	4.000000
75%	4.000000	3.00000	4.000000
max	4.000000	3.00000	5.000000

7. pandas 시계열 분석

7.1 pd.to_datetime 함수

- 날짜/시간을 나타내는 문자열을 자동으로 datetime 자료형으로 바꾼 후 -
- datetimeindex 자료형 인덱스를 생성

In [101]:

```
date_str = ["2018, 1, 1", "2018, 1, 4", "2018, 1, 5", "2018, 1, 6"]

idx = pd.to_datetime(date_str)
idx
```

Out[101]:

```
DatetimeIndex(['2018-01-01', '2018-01-04', '2018-01-05', '2018-01-06'], dtype='datetime64[ns]', freq
=None)
```

In [102]:

```
# 인덱스를 사용하여 시리즈나 데이터프레임을 생성
np.random.seed(0)

s = pd.Series(np.random.randn(4), index=idx)
s
```

Out[102]:

```
2018-01-01    1.764052
2018-01-04    0.400157
2018-01-05    0.978738
2018-01-06    2.240893
dtype: float64
```

7.2 pd.date_range 함수

시작일과 종료일 또는 시작일과 기간을 입력하면 범위 내의 인덱스를 생성

In [103]:

```
pd.date_range("2018-4-1", "2018-4-5")
```

Out[103]:

```
DatetimeIndex(['2018-04-01', '2018-04-02', '2018-04-03', '2018-04-04',
               '2018-04-05'],
              dtype='datetime64[ns]', freq='D')
```

In [104]:

```
pd.date_range(start="2018-4-1", periods=5)
```

Out[104]:

```
DatetimeIndex(['2018-04-01', '2018-04-02', '2018-04-03', '2018-04-04',
               '2018-04-05'],
              dtype='datetime64[ns]', freq='D')
```