

ch.9-1 Introduction to Pandas

Introduction to Pandas

- Panda는 데이터를 저장하고 조작하기 위한 두 가지 편리한 데이터 구조, 즉 Series와 DataFrame을 제공합니다. Series는 1차원 배열과 유사한 반면 DataFrame은 행렬 또는 스프레드시트 테이블을 나타내는 것과 더 유사합니다.

Series - 1차원 data

- Series 객체는 인덱스 배열을 사용하여 요소를 참조할 수 있는 1차원 값 배열로 구성됩니다.
- Series 객체는 목록, numpy 배열 또는 Python 사전에서 생성할 수 있습니다.
- Series(영상 시리즈) 객체에 대부분의 numpy 함수를 적용할 수 있습니다.



1차원 데이터로 Series 다 만들수 있음.

```
from pandas import Series

# 리스트에서 시리즈를 생성합니다.
s = Series([3.1, 2.4, -1.7, 0.2, -2.9, 4.5])

# 시리즈의 값을 출력합니다.
print(s)
# 시리즈의 값만을 출력합니다.
print('Values=', s.values)
# 시리즈의 인덱스를 출력합니다.
print('Index=', s.index)
```



리스트를 사용하여 Pandas의 Series를 생성합니다. 그리고 생성된 시리즈의 값과 인덱스를 출력 합니다. 결과는 시리즈의 값과 인덱스가 출력

```
0    3.1
1    2.4
2   -1.7
3    0.2
4   -2.9
5    4.5
dtype: float64
Values= [ 3.1  2.4 -1.7  0.2 -2.9  4.5]
Index= RangeIndex(start=0, stop=6, step=1)
```

```
import pandas as pd

# RangeIndex를 사용하여 인덱스 범위를 생성하고, 해당 범위의 값을 출력합니다.
# RangeIndex(start=0, stop=10, step=2)를 사용하여 0부터 9까지의 정수 중에서 2씩 증가하는 범위를 생성합니다.
for i in pd.core.indexes.range.RangeIndex(start=0, stop=10, step=2):
    print(i)
```



코드는 0부터 9까지의 정수 중에서 2씩 증가하는 범위를 생성하여 각 값을 출력합니다. 결과는 0, 2, 4, 6, 8가 출력

```
0
2
4
6
8
```

```
print(s.mean())
```

```
0.9333333333333332
```

```
import numpy as np
from pandas import Series

# NumPy 배열에서 시리즈를 생성합니다.
s2 = Series(np.random.randn(6))

# 시리즈의 값을 출력합니다.
print(s2)
# 시리즈의 값만을 출력합니다.
print('Values=', s2.values)
# 시리즈의 인덱스를 출력합니다.
print('Index=', s2.index)
```



NumPy의 `np.random.randn()` 함수를 사용하여 무작위 값을 생성하고, 이 값을 사용하여 Pandas의 Series를 생성합니다. 그리고 생성된 시리즈의 값과 인덱스를 출력합니다. 결과는 시리즈의 값과 인덱스가 출력

```
0    0.042110
1   -0.108167
2    0.788469
3   -1.450727
4    1.162612
5    0.513184
dtype: float64
Values= [ 0.04211044 -0.10816663  0.78846862 -1.45072659  1.162612    0.51318365]
Index= RangeIndex(start=0, stop=6, step=1)
```

```
from pandas import Series

# 시리즈를 생성합니다. 인덱스를 지정하여 생성합니다 & Label 값 각각 다름
s3 = Series([1.2, 2.5, -2.2, 3.1, -0.8, -3.2],
            index=['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'])

# 시리즈를 출력합니다.
print(s3)
```

```
# 시리즈의 값만을 출력합니다.
print('Values=', s3.values)
# 시리즈의 인덱스를 출력합니다.
print('Index=', s3.index)
```

```
Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
dtype: float64
Values= [ 1.2  2.5 -2.2  3.1 -0.8 -3.2]
Index= Index(['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'], dtype='object')
```

```
# 딕셔너리 객체에서 시리즈를 생성합니다.
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}
s4 = Series(capitals)

# 시리즈를 출력합니다.
print(s4)
# 시리즈의 값만을 출력합니다.
print('Values=', s4.values)
# 시리즈의 인덱스를 출력합니다.
print('Index=', s4.index)
```



딕셔너리 객체를 사용하여 **Pandas의 Series**를 생성합니다. 딕셔너리의 키는 시리즈의 인덱스로 사용되고, 값은 시리즈의 값으로 사용됩니다. 그리고 생성된 시리즈의 값과 인덱스를 출력합니다. 결과는 시리즈의 값과 인덱스가 출력

```
CA    Sacramento
MI      Lansing
MN      St Paul
TX      Austin
dtype: object
Values= ['Sacramento' 'Lansing' 'St Paul' 'Austin']
Index= Index(['CA', 'MI', 'MN', 'TX'], dtype='object')
```

```
# 시리즈를 생성합니다.
s3 = Series([1.2, 2.5, -2.2, 3.1, -0.8, -3.2],
            index=['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'])

# 시리즈를 출력합니다.
print(s3)

# 시리즈의 요소에 접근합니다.
print('\ns3[2]=', s3[2]) # 시리즈의 세 번째 요소를 출력합니다.
print('s3[\'Jan 3\']='', s3['Jan 3']) # 시리즈에서 인덱스가 'Jan 3'인 요소를 출력합니다.

# 시리즈의 슬라이스에 접근합니다.
print('\ns3[1:3]=') # 시리즈의 일부를 출력합니다.
print(s3[1:3])
```

```
print('s3.iloc([1:3])=')      # 시리즈의 일부를 출력합니다.
print(s3.iloc[1:3])
```



생성된 시리즈에서 특정 요소나 슬라이스에 접근하는 방법을 보여줍니다. `s3[2]` 는 시리즈의 세 번째 요소를 출력하고, `s3['Jan 3']` 는 시리즈에서 인덱스가 'Jan 3'인 요소를 출력합니다. 슬라이스에는 `s3[1:3]` 또는 `s3.iloc[1:3]` 와 같은 방식으로 접근할 수 있습니다.

```
Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
dtype: float64

s3[2]= -2.2
s3['Jan 3']= -2.2

s3[1:3]=
Jan 2    2.5
Jan 3   -2.2
dtype: float64
s3.iloc([1:3])=
Jan 2    2.5
Jan 3   -2.2
dtype: float64
```

```
# 시리즈의 차원과 요소 수를 출력합니다.
print('shape =', s3.shape) # 시리즈의 차원을 출력합니다.
print('size =', s3.size)   # 시리즈의 요소 수를 출력합니다.
```

```
shape = (6,)
size = 6
```

```
# 시리즈에서 값이 양수인 요소만 선택하여 출력합니다. 음수 탈락.
print(s3[s3 > 0])
```

```
Jan 1    1.2
Jan 2    2.5
Jan 4    3.1
dtype: float64
```

```
# 시리즈에 스칼라 값을 더하여 출력합니다.
print(s3 + 4) # 각 요소에 4를 더합니다.
```

```
# 시리즈에 스칼라 값을 나누어 출력합니다.
print(s3 / 4) # 각 요소를 4로 나눕니다.
```

```

Jan 1    5.2
Jan 2    6.5
Jan 3    1.8
Jan 4    7.1
Jan 5    3.2
Jan 6    0.8
dtype: float64
Jan 1    0.300
Jan 2    0.625
Jan 3   -0.550
Jan 4    0.775
Jan 5   -0.200
Jan 6   -0.800
dtype: float64

```

```

# 시리즈에 4를 더하여 값을 갱신합니다.
s3 = s3 + 4

```

```

# 갱신된 시리즈를 출력합니다.
s3

```



기존의 Series에 4를 더하여 값을 갱신합니다. `s3 = s3 + 4`를 통해 시리즈의 각 요소에 4가 더해진 새로운 시리즈를 생성하고, 이를 변수 `s3`에 다시 할당합니다. 그리고 갱신된 시리즈를 출력

```

Jan 1    9.2
Jan 2   10.5
Jan 3    5.8
Jan 4   11.1
Jan 5    7.2
Jan 6    4.8
dtype: float64

```

```

# 시리즈에 4를 더하고, 그 결과에 대해 로그를 취하여 출력합니다.
print(np.log(s3 + 4))

```



시리즈에 4를 더한 후, 그 결과에 대해 넘파이의 로그 함수를 적용합니다. `np.log(s3 + 4)`는 시리즈의 각 요소에 4를 더한 후 로그를 취한 값을 반환합니다. 결과는 로그가 적용된 시리즈가 출력

```

Jan 1    1.648659
Jan 2    1.871802
Jan 3    0.587787
Jan 4    1.960095
Jan 5    1.163151
Jan 6   -0.223144
dtype: float64

```

DataFrame

- DataFrame 개체는 다양한 유형(숫자, 문자열, 부울 등)일 수 있는 열 집합을 포함하는 표 형식의 스프레드시트와 같은 데이터 구조입니다. 시리즈와 달리 DataFrame에는 고유한 행 및 열 인덱스가 있습니다.
- DataFrame 개체를 만드는 방법은 여러 가지가 있습니다(예: 사전, 튜플 목록 또는 심지어 numpy의 ndarray).
- Series는 DataFrame의 단일 열에 대한 데이터 구조로, 개념적으로뿐만 아니라 말 그대로 DataFrame의 데이터가 실제로 Series의 컬렉션으로 메모리에 저장됩니다.

```
from pandas import DataFrame

# 딕셔너리를 사용하여 DataFrame을 생성합니다.
cars = {'make': ['Ford', 'Honda', 'Toyota', 'Tesla'],
        'model': ['Taurus', 'Accord', 'Camry', 'Model S'],
        'MSRP': [27595, 23570, 23495, 68000]}
carData = DataFrame(cars)

# DataFrame을 출력합니다.
carData
```



딕셔너리 객체를 사용하여 Pandas의 DataFrame을 생성합니다. 딕셔너리의 키는 DataFrame의 열 이름이 되고, 값은 해당 열의 데이터가 됩니다. 그리고 생성된 DataFrame을 출력합니다. 결과는 DataFrame 형식으로 출력

	make	model	MSRP
0	Ford	Taurus	27595
1	Honda	Accord	23570
2	Toyota	Camry	23495
3	Tesla	Model S	68000

```
# DataFrame의 행 인덱스를 출력합니다.
print(carData.index)

# DataFrame의 열 인덱스를 출력합니다.
print(carData.columns)
```

```
RangeIndex(start=0, stop=4, step=1) # index 속성을 사용하여 행 인덱스를 출력
Index(['make', 'model', 'MSRP'], dtype='object') # columns 속성을 사용하여 열 인덱스를 출력
```

```
# 행 인덱스 변경
carData2 = DataFrame(cars, index=[1, 2, 3, 4])
#print(carData2)

# DataFrame에 새로운 열을 추가합니다.
carData2['year'] = 2018 # 모든 행에 대해 'year' 열에 2018을 할당합니다.
carData2['dealership'] = ['Courtesy Ford', 'Capital Honda', 'Spartan Toyota', 'N/A']

# DataFrame을 출력합니다.
carData2
```



`index=[1, 2, 3, 4]` 를 사용하여 새로운 행 인덱스를 지정하고, `['year']` 과 `['dealership']` 를 사용하여 새로운 열을 추가합니다. 각 열에는 모든 행에 같은 값을 할당

make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018
2	Honda	Accord	23570	2018
3	Toyota	Camry	23495	2018
4	Tesla	Model S	68000	2018

튜플 목록에서 DataFrame을 작성합니다.

```
# 연도(year), 온도(temp), 강수량(precip) 데이터를 포함하는 튜플 리스트를 생성합니다.
tuplelist = [(2011, 45.1, 32.4), (2012, 42.4, 34.5), (2013, 47.2, 39.2),
              (2014, 44.2, 31.4), (2015, 39.9, 29.8), (2016, 41.5, 36.7)]

# DataFrame의 열 이름을 정의합니다.
columnNames = ['year', 'temp', 'precip']

# DataFrame을 생성하고, 튜플 리스트와 열 이름을 전달합니다.
weatherData = DataFrame(tuplelist, columns=columnNames)

# 생성된 DataFrame을 출력합니다.
weatherData
```



연도별 온도와 강수량 데이터를 가지는 DataFrame을 생성합니다. 튜플 리스트의 각 요소는 연도, 온도, 강수량을 나타내며, `columnNames` 변수는 DataFrame의 열 이름을 정의합니다.

`DataFrame()` 함수를 사용하여 튜플 리스트와 열 이름 리스트를 전달하여 DataFrame을 생성하고, 생성된 DataFrame을 출력

	year	temp	precip
0	2011	45.1	32.4
1	2012	42.4	34.5
2	2013	47.2	39.2
3	2014	44.2	31.4
4	2015	39.9	29.8
5	2016	41.5	36.7

```
# 온도(temp) 열에서 최대값을 출력합니다.
print(weatherData['temp'].max())

# 온도(temp) 열에서 평균값을 출력합니다.
print(weatherData['temp'].mean())

# 온도(temp) 열에서 표준편차를 출력합니다.
print(weatherData['temp'].std())
```

```
47.2
43.383333333333326
```

2.639254945371264

- numpy ndarray에서 DataFrame 생성

```
import numpy as np

# 5x3 크기의 랜덤 행렬을 생성합니다.
npdata = np.random.randn(5, 3)

# DataFrame의 열 이름을 정의합니다.
columnNames = ['x1', 'x2', 'x3']

# DataFrame을 생성하고, 랜덤 행렬과 열 이름을 전달합니다.
data = DataFrame(npdata, columns=columnNames)

# 생성된 DataFrame을 출력합니다.
data
```



NumPy의 `np.random.randn()` 함수를 사용하여 5x3 크기의 랜덤 행렬을 생성하고, 이를 DataFrame으로 변환합니다. 열 이름은 `columnNames` 변수에 정의된 것을 사용합니다. 생성된 DataFrame은 행렬의 값과 함께 출력

	x1	x2	x3
0	-0.055157	-0.990822	-1.290890
1	0.090387	0.777583	0.238338
2	-0.802349	-1.060129	-0.045762
3	1.126908	-0.652445	0.594889
4	1.131444	-1.702500	0.420432

- DataFrame의 요소는 다양한 방법으로 액세스할 수 있습니다.

```
# 'x2' 열을 선택하여 해당 열의 데이터를 출력합니다.
print(data['x2'])

# 선택된 열의 데이터 타입을 출력합니다.
print(type(data['x2']))
```

```
0    -0.990822
1     0.777583
2    -1.060129
3    -0.652445
4    -1.702500
Name: x2, dtype: float64
<class 'pandas.core.series.Series'>
```

```
print(data[0:1])
```

```
      x1      x2      x3
0 -0.055157 -0.990822 -1.29089
```



```
print(data[0::2])
```

```
      x1      x2      x3
0 -0.055157 -0.990822 -1.290890
2 -0.802349 -1.060129 -0.045762
4  1.131444 -1.702500  0.420432
```

```
# 'data' 테이블의 세 번째 행을 선택하고 출력합니다.
print('Row 3 of data table:')
print(data.iloc[2])          # DataFrame의 세 번째 행을 반환합니다.

# 선택된 객체의 유형을 출력합니다. (Series여야 함)
print(type(data.iloc[2]))

# 'carData2' 테이블의 세 번째 행을 선택하고 출력합니다.
print('\nRow 3 of car data table:')
print(carData2.iloc[2])     # 행에는 다양한 유형의 객체가 포함됩니다.
```



`iloc[2]` 를 사용하여 DataFrame `data` 의 세 번째 행에 접근하고 출력합니다. 그런 다음 반환된 객체의 유형을 출력하여 실제로 Series임을 확인합니다.

마지막으로 다른 DataFrame `carData2` 의 세 번째 행에 접근하고 출력하며, 행이 서로 다른 유형의 객체를 포함할 수 있다는 점을 언급

```
Row 3 of data table:
x1      1.644521
x2     -1.262173
x3     -1.584139
Name: 2, dtype: float64
<class 'pandas.core.series.Series'>
```

```
Row 3 of car data table:
MSRP      23495
make      Toyota
model     Camry
year      2018
dealership Spartan Toyota
Name: 3, dtype: object
```

carData2

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	23570	2018	Capital Honda
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	N/A

```
# DataFrame에서 특정 요소에 접근하여 출력합니다.
print(carData2.iloc[1, 2])      # 두 번째 행, 세 번째 열의 값을 반환합니다. (정수 인덱싱)
print(carData2.loc[1, 'model']) # 두 번째 행, 'model' 열의 값을 반환합니다.

# DataFrame의 슬라이스에 접근하여 출력합니다.
print('carData2.iloc[1:3, 1:3]=')
print(carData2.iloc[1:3, 1:3]) # 두 번째부터 세 번째 행과 두 번째부터 세 번째 열을 포함하는 슬라이스를 반환함
```



`iloc` 및 `loc` 를 사용하여 DataFrame에서 특정 요소에 접근하고 출력합니다. 그런 다음 `iloc` 을 사용하여 DataFrame의 슬라이스에 접근하고 출력

```
23570
Taurus
carData2.iloc[1:3,1:3]=
   model  MSRP
2 Accord 23570
3  Camry 23495
```

```
# DataFrame에서 특정 요소를 업데이트합니다.
carData2.iloc[1, 1] = 'Accord'      # 두 번째 행, 두 번째 열의 값을 'Accord'로 변경합니다.
carData2.loc[4, 'dealership'] = 'SF Tesla' # 네 번째 행, 'dealership' 열의 값을 'SF Tesla'로 변경함
carData2.loc[2, 'dealership'] = np.nan  # 두 번째 행, 'dealership' 열의 값을 null 값으로 변경합니다.
carData2
```



`iloc` 및 `loc` 을 사용하여 DataFrame에서 특정 요소를 업데이트하는 방법을 설명합니다.

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	23570	2018	NaN
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	SF Tesla

carData2

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	Accord	2018	NaN
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	SF Tesla

carData2

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	23570	2018	Capital Honda
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	N/A

```
carData2.iloc[1, 1] = 'Accord'
carData2.loc[4, 'dealership'] = 'SF Tesla'
carData2.loc[2, 'dealership'] = np.nan # null value
carData2['sale price'] = carData2['MSRP'] * 1.1
```

```
# 5번째 열 추가 후 삭제
carData2[5] = 1
carData2 = carData2.drop(5, axis=1)
```

```
# 5행 추가
carData2.loc[5] = 1
```

```
# 10행 추가
carData2.loc[10] = 100
```

```
# 첫 4개의 행 출력
carData2.make[:4]
```

```
1      Ford
2      Honda
3      Toyota
4      Tesla
Name: make, dtype: object
```

carData2

	make	model	MSRP	year	dealership	sale price
1	Ford	Taurus	27595	2018	Courtesy Ford	30354.5
2	Honda	Accord	23570	2018	NaN	25927.0
3	Toyota	Camry	23495	2018	Spartan Toyota	25844.5
4	Tesla	Model S	68000	2018	SF Tesla	74800.0
5	1	1	1	1	1	1.0
10	100	100	100	100	100	100.0

```
print('carData2.shape =', carData2.shape)
print('carData2.size =', carData2.size)
```

```
carData2.shape = (6, 6)
carData2.size = 36
```

carData2

	make	model	MSRP	year	dealership	sale price
1	Ford	Taurus	27595	2018	Courtesy Ford	30354.5
2	Honda	Accord	23570	2018	NaN	25927.0
3	Toyota	Camry	23495	2018	Spartan Toyota	25844.5

4	Tesla	Model S	68000	2018	SF Tesla	74800.0
5	1	1	1	1	1	1.0
10	100	100	100	100	100	100.0

```
carData2
```

	make	model	MSRP
0	Ford	Taurus	27595
1	Honda	Accord	23570
2	Toyota	Camry	23495
3	Tesla	Model S	68000

```
carData[[True,False, False, True]]
```

	make	model	MSRP
0	Ford	Taurus	27595
3	Tesla	Model S	68000

- DataFrame에 대한 인덱싱 연산자의 주요 목적은 문자열 또는 문자열 목록을 사용하여 하나 이상의 열을 선택하는 것입니다.
 - 갑자기 이 예제는 전체 행이 부울 값으로 선택된다는 것을 보여줍니다.
 - 이것이 Pandas를 사용하기에 가장 혼란스러운 라이브러리 중 하나로 만드는 것입니다.
- 단지 인덱싱 연산자에 과부하가 걸릴 뿐입니다.
 - 이것은 입력에 따라 Pandas가 완전히 다른 일을 한다는 것을 의미합니다. 여기에 여러분이 인덱싱 연산자에게만 전달하는 다른 객체에 대한 규칙이 있습니다.
- string — 열을 문자열의 직렬 목록으로 반환 - 모든 열을 DataFrame 슬라이스로 반환 - 행 선택(라벨 및 정수 위치 모두 수행 - 혼동!)
 - 부울 시퀀스는 True 요약에서 주로 인덱싱 연산자만 열을 선택하지만 전달하면 True인 모든 행이 선택됩니다.

```
carData[['MSRP', 'make']]
```

	MSRP	make
0	27595	Ford
1	23570	Honda
2	23495	Toyota
3	68000	Tesla

```
carData[:2] # 0~1 까지의 row(행)만 출력
```

	make	model	MSRP
0	Ford	Taurus	27595
1	Honda	Accord	23570

```
# DataFrame에서 MSRP가 25000보다 큰 행을 선택하고 출력합니다.
print('carData2[carData2.MSRP > 25000]')
```

```
print(carData2[carData2.MSRP > 25000])

# DataFrame에서 MSRP가 25000보다 큰 행을 선택하고 출력합니다.
# (대괄호 안에 열 이름을 따옴표로 감싸는 방식)
print()
print(carData2[carData2['MSRP'] > 25000])
```

```
carData2[carData2.MSRP > 25000]
   make  model  MSRP  year  dealership  sale price
1  Ford  Taurus  27595  2018  Courtesy Ford    30354.5
4  Tesla  Model S  68000  2018      SF Tesla    74800.0
```

```
   make  model  MSRP  year  dealership  sale price
1  Ford  Taurus  27595  2018  Courtesy Ford    30354.5
4  Tesla  Model S  68000  2018      SF Tesla    74800.0
```

Arithmetic Operations

```
# DataFrame을 출력합니다.
print(data)

# DataFrame의 전치 연산을 수행하여 출력합니다.
print('Data transpose operation:')
print(data.T)    # transpose operation

# DataFrame에 상수 값을 더한 결과를 출력합니다.
print('Addition:')
print(data + 4)    # addition operation

# DataFrame에 상수 값을 곱한 결과를 출력합니다.
print('Multiplication:')
print(data * 10)    # multiplication operation
```

```
x1      x2      x3
0 -0.924658 -0.003035 -0.952447
1  1.210632 -0.108034 -0.655278
2  0.150605  1.420714  0.459163
3 -0.052479 -2.112469 -0.652657
4  0.374842 -0.554976 -0.021826
Data transpose operation:
      0      1      2      3      4
x1 -0.924658  1.210632  0.150605 -0.052479  0.374842
x2 -0.003035 -0.108034  1.420714 -2.112469 -0.554976
x3 -0.952447 -0.655278  0.459163 -0.652657 -0.021826
Addition:
      x1      x2      x3
0  3.075342  3.996965  3.047553
1  5.210632  3.891966  3.344722
2  4.150605  5.420714  4.459163
3  3.947521  1.887531  3.347343
4  4.374842  3.445024  3.978174
Multiplication:
      x1      x2      x3
```

```

0  -9.246584  -0.030348  -9.524468
1  12.106321  -1.080336  -6.552781
2   1.506052  14.207143   4.591629
3  -0.524787 -21.124686  -6.526568
4   3.748423  -5.549757  -0.218264

```

```

# 첫 번째 DataFrame을 출력합니다.
print('data =')
print(data)

# 두 번째 DataFrame을 생성하고 출력합니다.
columnNames = ['x1', 'x2', 'x3']
data2 = DataFrame(np.random.randn(5,3), columns=columnNames)
print('\ndata2 =')
print(data2)

# 두 DataFrame을 더한 결과를 출력합니다.
print('\ndata + data2 = ')
print(data.add(data2))

# 두 DataFrame을 곱한 결과를 출력합니다.
print('\ndata * data2 = ')
print(data.mul(data2))

```



첫 번째 DataFrame인 `data` 와 두 번째 DataFrame인 `data2` 를 생성하고 출력합니다. 그런 다음 두 DataFrame 간의 덧셈 및 곱셈 연산을 수행하여 결과를 출력

```

data =
      x1      x2      x3
0 -0.055157 -0.990822 -1.290890
1  0.090387  0.777583  0.238338
2 -0.802349 -1.060129 -0.045762
3  1.126908 -0.652445  0.594889
4  1.131444 -1.702500  0.420432

data2 =
      x1      x2      x3
0  0.749343 -0.434704 -0.306548
1 -0.183713 -0.548752 -1.720078
2 -1.693204 -1.008143 -0.422786
3 -0.348182 -2.241761 -0.449988
4  1.256190  1.539085  0.795187

data + data2 =
      x1      x2      x3
0  0.694186 -1.425526 -1.597438
1 -0.093326  0.228831 -1.481741
2 -2.495553 -2.068272 -0.468548
3  0.778726 -2.894206  0.144901
4  2.387634 -0.163416  1.215619

data * data2 =
      x1      x2      x3

```

```

0 -0.041331  0.430714  0.395720
1 -0.016605 -0.426700 -0.409959
2  1.358540  1.068762  0.019347
3 -0.392369  1.462625 -0.267693
4  1.421308 -2.620292  0.334322

```

data

	x1	x2	x3
0	-0.055157	-0.990822	-1.290890
1	0.090387	0.777583	0.238338
2	-0.802349	-1.060129	-0.045762
3	1.126908	-0.652445	0.594889
4	1.131444	-1.702500	0.420432

```

# DataFrame의 각 요소에 대한 절대값을 반환합니다.
print(data.abs())    # get the absolute value for each element

# 각 열의 최대값을 반환합니다.
print('\nMaximum value per column:')
print(data.max())    # get maximum value for each column

# 각 행의 최소값을 반환합니다.
print('\nMinimum value per row:')
print(data.min(axis=1))    # get minimum value for each row

# 각 열의 값의 합을 반환합니다.
print('\nSum of values per column:')
print(data.sum())    # get sum of values for each column

# 모든 값의 합을 반환합니다.
print('\nSum of all values:')
print(data.sum().sum())    # get sum over all values

# 각 행의 평균값을 반환합니다.
print('\nAverage value per row:')
print(data.mean(axis=1))    # get average value for each row

# 각 열의 최대값과 최소값의 차이를 계산합니다.
print('\nCalculate max - min per column')
f = lambda x: x.max() - x.min()
print(data.apply(f))

# 각 행의 최대값과 최소값의 차이를 계산합니다.
print('\nCalculate max - min per row')
f = lambda x: x.max() - x.min()
print(data.apply(f, axis=1))

```

```

x1      x2      x3
0  0.055157  0.990822  1.290890
1  0.090387  0.777583  0.238338
2  0.802349  1.060129  0.045762

```

```
3  1.126908  0.652445  0.594889
4  1.131444  1.702500  0.420432
```

Maximum value per column:

```
x1    1.131444
x2    0.777583
x3    0.594889
dtype: float64
```

Minimum value per row:

```
0   -1.290890
1    0.090387
2   -1.060129
3   -0.652445
4   -1.702500
dtype: float64
```

Sum of values per column:

```
x1    1.491234
x2   -3.628313
x3   -0.082993
dtype: float64
```

Sum of all values:

```
-2.2200729548632316
```

Average value per row:

```
0   -0.778956
1    0.368769
2   -0.636080
3    0.356451
4   -0.050208
dtype: float64
```

Calculate max - min per column

```
x1    1.933792
x2    2.480083
x3    1.885779
dtype: float64
```

Calculate max - min per row

```
0    1.235734
1    0.687196
2    1.014368
3    1.779353
4    2.833944
dtype: float64
```

Plotting Series and DataFrame

- Series(영상 시리즈) 또는 DataFrame(데이터 프레임)에 저장된 데이터를 플롯하는 데 사용할 수 있는 내장 기능이 있습니다.

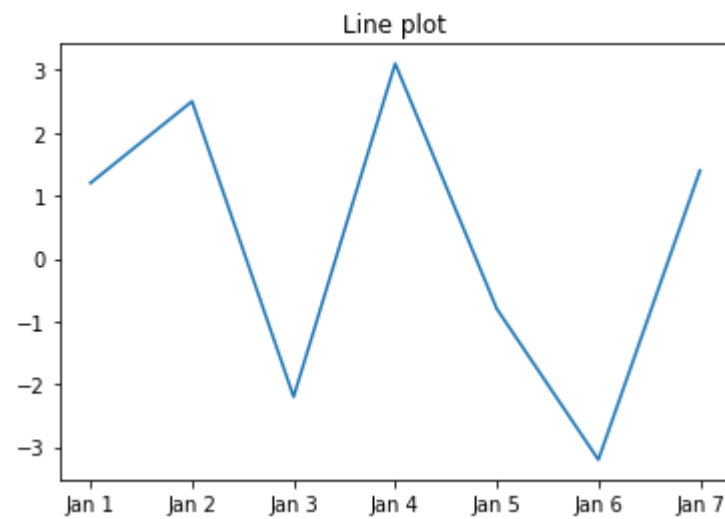
```
%matplotlib inline # 주피터 노트북에서 그래프를 인라인으로 표시하기 위한 매직 명령어
```

```
import matplotlib.pyplot as plt # matplotlib의 pyplot 모듈을 plt 별칭으로 가져옴
```



```
# Series 객체 정의
s3 = Series([1.2, 2.5, -2.2, 3.1, -0.8, -3.2, 1.4], # 값 리스트
            index=['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6', 'Jan 7']) # 값에 대응하는 인덱스

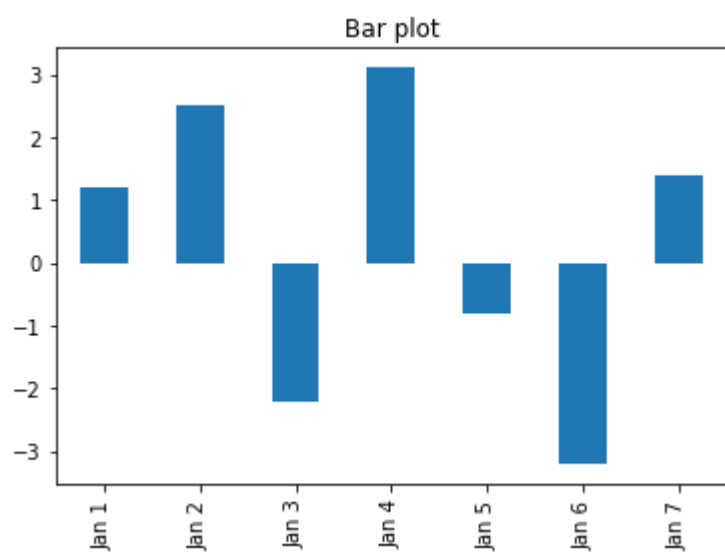
# 선 그래프 그리기
s3.plot(kind='line', title='Line plot') # 'line' 종류의 선 그래프를 그리고, 제목을 'Line plot'으로 설정
plt.show() # 그래프를 표시
```



```
s3.plot(kind='bar', title='Bar plot') # 'bar' 종류의 막대 그래프를 그리고, 제목을 'Bar plot'으로 설정
plt.show() # 그래프를 표시
```



`s3.plot(kind='bar', title='Bar plot')` 부분은 막대 그래프를 그리는데 사용됩니다. `kind='bar'` 는 그래프의 종류를 막대 그래프로 지정하고, `title='Bar plot'` 는 그래프의 제목을 설정



s3

```
Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
Jan 7    1.4
dtype: float64
```

```
s3.plot(kind='hist', title = 'Histogram', bins=3)
plt.show()
```

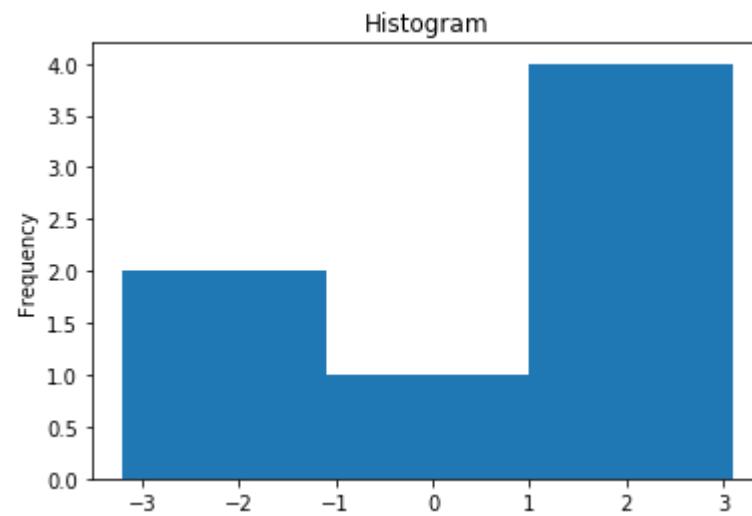


`s3.plot(kind='hist', title='Histogram', bins=3)` 부분은 히스토그램을 그리는데 사용됩니다.

`kind='hist'` 는 그래프의 종류를 히스토그램으로 지정하고,

`title='Histogram'` 는 그래프의 제목을 설정합니다.

`bins=3` 는 히스토그램의 구간(bin) 개수를 설정합니다. 이 경우에는 데이터를 3개의 구간으로 나누어 히스토그램을 표현



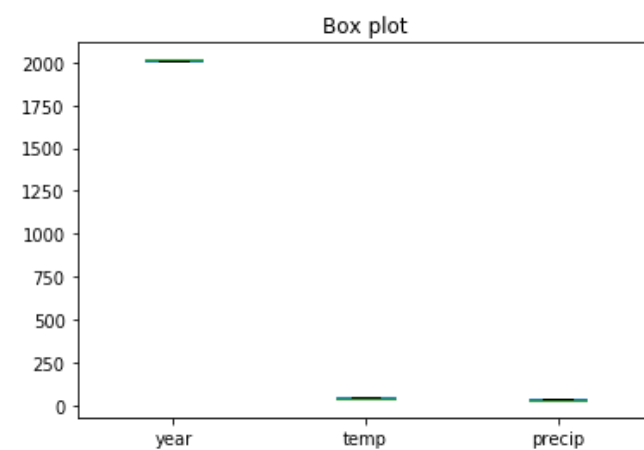
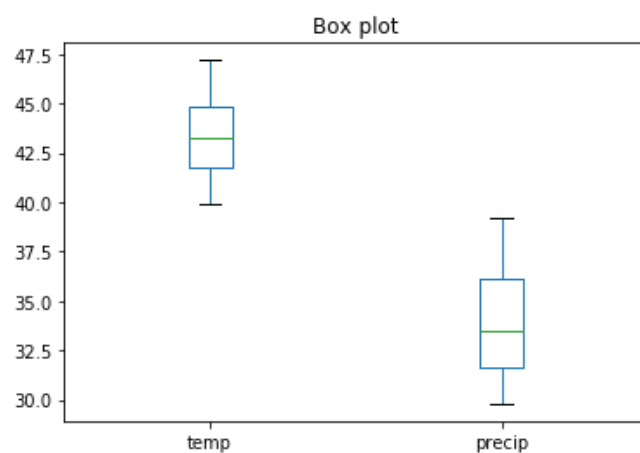
```
# 연도, 온도, 강수량 데이터를 담은 튜플 리스트
tuplelist = [(2011, 45.1, 32.4), (2012, 42.4, 34.5), (2013, 47.2, 39.2),
              (2014, 44.2, 31.4), (2015, 39.9, 29.8), (2016, 41.5, 36.7)]

# 데이터프레임의 열 이름 지정
columnNames = ['year', 'temp', 'precip']

# 튜플 리스트를 데이터프레임으로 변환하여 날씨 데이터 생성
weatherData = DataFrame(tuplelist, columns=columnNames)

# 온도와 강수량에 대한 상자 그림을 그리고, 제목을 'Box plot'으로 설정
weatherData[['temp', 'precip']].plot(kind='box', title='Box plot')

# 전체 데이터에 대한 상자 그림을 그리고, 제목을 'Box plot'으로 설정
weatherData.plot(kind='box', title='Box plot')
plt.show()
```



Lab.

```
%%bash
cat stocks.txt
```

```
import pandas as pd # pandas 라이브러리를 pd 별칭으로 가져옴

# 'stocks.txt' 파일을 탭('\t')으로 구분하여 데이터프레임으로 읽어옴
stocks = pd.read_csv('stocks.txt', delimiter='\t')
stocks[:5] # 데이터프레임의 첫 5개 행을 출력
```

	ymbol	date	closing_price
0	AAPL	2015-01-23	112.98
1	AAPL	2015-01-22	112.40
2	AAPL	2015-01-21	109.55
3	AAPL	2015-01-20	108.72
4	AAPL	2015-01-16	105.99

```
import numpy as np # numpy 라이브러리를 np 별칭으로 가져옴

np.unique(stocks['symbol']) # 'symbol' 열의 고유한 값들을 찾아 출력
# np.unique(stocks.symbol) # 더 RDB 스타일
```

```
array(['AAPL', 'FB', 'MSFT'], dtype=object)
```

```
stocks.symbol
```

```
stocks.loc[:10, 'symbol']
```

```
0    AAPL
1    AAPL
2    AAPL
3    AAPL
4    AAPL
5    AAPL
6    AAPL
7    AAPL
8    AAPL
9    AAPL
10   AAPL
Name: symbol, dtype: object
```

```
stocks.closing_price > 200
```

```
# Result:
# 0      False
# 1      False
# 2      False
```

```
# 3      False
# 4      False
...

# 16552   False
# 16553   False
# 16554   False
# Name: closing_price, Length: 16555, dtype: bool
```

```
stocks[stocks.closing_price > 112]
```

ymbol	date	closing_price
0	AAPL	2015-01-23
1	AAPL	2015-01-22
9	AAPL	2015-01-09
16	AAPL	2014-12-30
17	AAPL	2014-12-29


...

```
stocks[stocks['symbol'] == 'AAPL'] # 'symbol' 열이 'AAPL'인 행들을 선택하여 출력
# stocks[stocks.symbol == 'AAPL'] # SQL의 select * from stocks where stocks.symbol == 'AAPL'과
```

symbol	date	closing_price
0	AAPL	2015-01-23
1	AAPL	2015-01-22
2	AAPL	2015-01-21
3	AAPL	2015-01-20
4	AAPL	2015-01-16

```
type(stocks['symbol'][0])

# Result: str
```

 `stocks['symbol'][0]` 는 'stocks' 데이터프레임의 'symbol' 열에서 첫 번째 요소를 선택하는 역할을 합니다. 그리고 `type()` 함수를 사용하여 해당 요소의 데이터 타입을 확인

```
type(stocks['date'][0])

# str
```

```
type(stocks['closing_price'][0])

# numpy.float64
```

```
date_parser = lambda x: pd.datetime.strptime(x, '%Y-%m-%d') # 주어진 문자열을 datetime 형식으로 변환

stocks = pd.read_csv('stocks.txt', delimiter='\t', parse_dates=['date'], date_parser=date_parser)
```



설명

여기서 `date_parser` 는 주어진 문자열을 `%Y-%m-%d` 형식에 맞게 `datetime` 형식으로 변환하는 람다 함수입니다.

`pd.read_csv()` 함수의 `parse_dates=['date']` 인자는 'date' 열을 파싱하여 `datetime` 형식으로 변환하라는 것을 의미하며, `date_parser=date_parser` 는 이를 위해 사용할 파서 함수를 지정합니다.

따라서 이 코드를 실행하면 'stocks.txt' 파일에서 데이터를 읽어와서 'date' 열의 데이터를 `datetime` 형식으로 변환하여 `DataFrame`으로 저장합니다.

```
stocks['date'][0]

# Timestamp('2015-01-23 00:00:00')
```

```
# 'date' 열이 '1980-01-01' 이후인 행들을 선택하여 출력
# 'pd.datetime.strptime()' 함수를 사용하여 문자열을 datetime 형식으로 변환하여 비교합니다.
stocks[stocks['date'] >= pd.datetime.strptime('1980-01-01', '%Y-%m-%d')]
```



'symbol' 열이 'AAPL' 또는 'MSFT'인 행들을 선택하여 출력하려는 시도

아래와 같이 논리 연산자 'or'를 사용하면 오류가 발생합니다. 따라서 논리 연산자 'or' 대신 '|'를 사용해야 합니다.

```
stocks[stocks['symbol'] == 'AAPL' | stocks['symbol'] == 'MSFT']
```

symbol	date	closing_price
0	AAPL	2015-01-23
1	AAPL	2015-01-22
2	AAPL	2015-01-21
3	AAPL	2015-01-20
4	AAPL	2015-01-16

```
stocks[(stocks['symbol'] == 'AAPL') & (stocks['date'] >= pd.datetime.strptime('1980-01-01', '%Y-%m-%d'))]
```

'symbol' 열이 'AAPL'이고 'date' 열이 '1980-01-01' 이후인 행들을 선택하여 출력하는 것을 설명합니다. '&' 연산자를 사용하여 'and' 조건을 나타내고, '~' 연산자를 사용하여 'not' 연산을 수행할 수 있습니다. '|' 연산자를 사용하여 'or' 조건을 나타낼 수도 있습니다.

	symbol	date	closing_price
0	AAPL	2015-01-23	112.98
1	AAPL	2015-01-22	112.40
2	AAPL	2015-01-21	109.55
3	AAPL	2015-01-20	108.72
4	AAPL	2015-01-16	105.99

.....

8599	AAPL	1980-12-17	0.40
------	------	------------	------

8600	AAPL	1980-12-16	0.39
8601	AAPL	1980-12-15	0.42
8602	AAPL	1980-12-12	0.44

8603 rows × 3 columns

```
# 데이터프레임의 특정 열만 선택하여 출력하는 것을 'projection'이라는 변수에 저장합니다.
projection = stocks[['symbol', 'closing_price']]
# projection = stocks.loc[:, ['symbol', 'closing_price']] # 위 코드와 동일한 결과를 반환합니다.
projection
```



데이터프레임의 'symbol'과 'closing_price' 열만 선택하여 출력하는 것을 'projection'이라는 변수에 할당하는 것을 설명합니다. 또한 주식에는 'projection' 변수를 생성하는 다른 방법인 `.loc` 메서드도 제시되어 있습니다.

	symbol	closing_price
0	AAPL	112.98
1	AAPL	112.40
2	AAPL	109.55
3	AAPL	108.72
4	AAPL	105.99

....

```
# 데이터프레임에서 'closing_price' 열의 값이 20 이하인 행들을 선택하여 출력하는 것을
# 'selection'이라는 변수에 저장합니다.
```

```
selection = projection[projection['closing_price'] <= 20]
```



'closing_price' 열의 값이 20 이하인 행들을 선택하여 출력하는 것을 'selection'이라는 변수에 할당하는 것을 설명합니다.

이것은 'projection' 데이터프레임에서 'closing_price'가 20 이하인 행들을 선택하고, 'selection'이라는 변수에 저장합니다.

	symbol	closing_price
1390	AAPL	19.96
1391	AAPL	19.87
1392	AAPL	19.25
1393	AAPL	19.2

...

16489	FB	19.16
16491	FB	19.05
16492	FB	19.87

11074 rows × 2 columns

```
aggregation = selection.groupby('symbol')
aggregation.count()
```



'selection' 데이터프레임을 'symbol' 열을 기준으로 그룹화하고, 각 그룹의 행 수를 세어 출력하는 것을 'aggregation' 이라는 변수에 할당하는 것을 설명합니다. 'symbol' 열로 그룹화된 데이터프레임에 대해 `count()` 메서드를 호출하여 각 그룹의 행 수를 계산합니다.

symbol	closing_price
AAPL	7017
FB	30
MSFT	4027

```
aggregation.sum() # 각 그룹의 'closing_price' 열의 합을 계산
```

symbol	closing_price
AAPL	17925.96
FB	577.25
MSFT	25909.30

```
aggregation.std() # 각 그룹의 'closing_price' 열의 표준 편차를 계산
```

symbol	closing_price
AAPL	3.869626
FB	0.500848
MSFT	7.334630

Example: Rescaling

```
# 'closing_price' 열의 평균과 표준 편차를 계산하여 변수에 할당
price_mean = stocks['closing_price'].mean() # 'closing_price' 열의 평균 계산
price_std_dev = stocks['closing_price'].std() # 'closing_price' 열의 표준 편차 계산

# 표준화된 가격 계산
# 'closing_price' 열의 각 요소에서 평균을 빼고, 표준 편차로 나눈 값을 구하여 'standardized_prices' 변수에 저장
standardized_prices = (stocks['closing_price'] - price_mean) / price_std_dev
```



'closing_price' 열의 각 요소를 해당 열의 평균에서 빼고, 표준 편차로 나누어 표준화된 가격 (`standardized_prices`) 을 계산하는 것을 설명합니다. 이를 위해 먼저 'closing_price' 열의 평균과 표준 편차를 계산하고, 이를 사용하여 각 요소를 표준화

```
0      4.761541
1      4.733303
2      4.594547
3      4.554137
4      4.421224

...

16552   0.770247
16553   0.917766
```

```
16554      1.122248
Name: closing_price, Length: 16555, dtype: float64
```

```
# 'standarded_prices'를 'stocks' 데이터프레임에 'std_closing_price' 열로 추가
stocks['std_closing_price'] = standarded_prices
```



표준화된 가격(standarded_prices)을 'stocks' 데이터프레임에 'std_closing_price' 열로 추가하는 것을 설명합니다. 'standarded_prices' 를 새로운 열로 추가하여 데이터프레임을 확장합니다. 이를 통해 표준화된 가격 정보를 데이터프레임에 포함합니다.

	symbol	date	closing_price	std_closing_prices
0	AAPL	2015-01-23	112.98	4.761541
1	AAPL	2015-01-22	112.40	4.733303
2	AAPL	2015-01-21	109.55	4.594547
3	AAPL	2015-01-20	108.72	4.554137

.....

16552	FB	2012-05-22	31.00	0.770247
16553	FB	2012-05-21	34.03	0.917766
16554	FB	2012-05-18	38.23	1.122248

16555 rows × 4 columns

```
# 'standarded_prices'를 'stocks' 데이터프레임의 'closing_price' 열로 대체
stocks['closing_price'] = standarded_prices
```



표준화된 가격(standarded_prices)을 'stocks' 데이터프레임의 'closing_price' 열로 대체하는 것을 설명합니다. 'standarded_prices' 를 기존의 'closing_price' 열에 대입하여 해당 열을 업데이트합니다. 이를 통해 표준화된 가격으로 'closing_price' 열을 대체

	symbol	date	closing_price	std_closing_price
0	AAPL	2015-01-23	4.761541	4.761541
1	AAPL	2015-01-22	4.733303	4.733303
2	AAPL	2015-01-21	4.594547	4.594547
3	AAPL	2015-01-20	4.554137	4.554137

.....

16552	FB	2012-05-22	0.770247	0.770247
16553	FB	2012-05-21	0.917766	0.917766
16554	FB	2012-05-18	1.122248	1.122248

16555 rows × 4 columns