

## Week 6. 쉽고 강력한 데이터프레임 라이브러리 - Pandas

- pandas는 구조화된 데이터를 빠르고 쉬우면서도 다양한 형식으로 가공할 수 있는 풍부한 자료 구조와 함수 제공
- pandas는 파이썬을 강력하고 생산적인 데이터 분석 환경으로 만드는 데 꼭 필요
- 주로 pandas의 주요 객체인 DataFrame을 다룰 텐데, 이 객체는 2차원 표 또는 행과 열을 나타내는 자료 구조(엑셀로 보면 이해하기 쉽다)
- R의 `data.frame` 객체에서 따온 DataFrame. 이렇게 컴퓨터 세상에서는 좋은건 가져다 씀.
- pandas는 다차원으로 구조화 된 데이터를 뜻하는 계량 경제학 용어인 PAnel DAta와 파이썬 데이터 분석Python data analysiS에서 따온 이름
- pandas는 NumPy의 고성능 배열 계산 기능과 스프레드시트, SQL같은 관계형 데이터베이스의 유연한 데이터 조작 기능 조합. 세련된 인덱싱 기능으로 쉽게 데이터 재배치하고 잘게 조각내거나 집계하고 부분집합을 구할 수 있음
- 금융 데이터를 분석하려는 사용자에게 pandas는 풍부한 고성능 시계열 처리 기능과 금융 데이터에 딱 맞는 도구 제공. 사실 저자는 처음부터 금융 데이터 분석 애플리케이션용으로 pandas 설계

## 자료형

- `Series`
  - 시계열 데이터
  - Index를 가지는 1차원 NumPy Array
- `DataFrame`
  - 복수 필드 시계열 데이터 또는 테이블 데이터
  - Index를 가지는 2차원 NumPy Array
- `Index`
  - Label: 각각의 Row/Column에 대한 이름
  - Name: 인덱스 자체에 대한 이름

The diagram illustrates the structure of a Pandas DataFrame. It shows a table with columns labeled 'date', 'infl', 'realgdp', and 'unemp'. The 'date' column is highlighted with a red box and labeled '(Row) Index'. The 'realgdp' column is highlighted with a blue box and labeled 'Column Index'. The 'date' column is also labeled 'Index Name' and 'Index Label'.

date	infl	realgdp	unemp
1959-03-31	0.00	2710.349	5.8
1959-06-30	2.34	2778.801	5.1
1959-09-30	2.74	2775.488	5.3
1959-12-31	0.27	2785.204	5.6
1960-03-31	2.31	2847.699	5.2

# Series

---

- 일련의 객체를 담을 수 있는 1차원 배열 같은 자료 구조(어떤 NumPy 자료형이라도 담을 수 있다)
- 파이썬의 사전형과 비슷하다(정렬된 사전형이라고 보면 된다)
- 파이썬 사전형 객체에서 생성할 수 있다.
- 생성시 index 인수로 Index 지정
- Index 원소는 각 데이터에 대한 key 역할을 하는 Label
- dict

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

```
In [18]: s = pd.Series([4, 7, -5, 3], index=["a", "b", "c", "d"])
#index를 지정하지 않을 경우 0,1,2,3, ... 으로 index가 자동 지정됨

print(s)
print(type(s.values))
print(s.index)
print('-----')
```

```
a    4
b    7
c   -5
d    3
dtype: int64
<class 'numpy.ndarray'>
Index(['a', 'b', 'c', 'd'], dtype='object')
-----
```

## Series 인덱싱

```
In [8]: #Single Label
print('a값:',s['a']) # 또는 s[0]
print('-----')

# Label Slicing
print(s['a':'c']) # (마지막 원소 포함)
print(s[0:3]) # (마지막 원소 미포함)
print('-----')

# Label을 원소로 가지는 Label (Label을 사용한 List Fancy Indexing)
print(s[['a', 'd']])
print(s.get(['a','d']))
print(s[[0,3]])
print('-----')

# 조건문
print(s[s <= 3])
```

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

```
In [24]: # 값 할당
s['a'] = 10
s
```

```
Out[24]: a    10
b     7
c    -5
d     3
dtype: int64
```

```
In [25]: print("a" in s)

True
```

```
In [26]: for k, v in s.items():
          print(k, v)
```

```
a 10
b 7
c -5
d 3
```

## 인덱스 변경

```
In [27]: s.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
print(s)

Bob      10
Steve     7
Jeff     -5
Ryan      3
dtype: int64
```

## 연산

```
In [28]: #Vectorized operation 도 가능함
print(s*2)

Bob      20
Steve    14
Jeff    -10
Ryan      6
dtype: int64
```

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

#Dictornary --> Series
s2 = pd.Series(sdata)
s2
```

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

```
In [33]: s2.name = "population"
s2.index.name = "state"
s2
```

```
Out[33]: state
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
Name: population, dtype: int64
```

```
In [39]: # 사전으로 형성된 객체에, 새 인덱스가 추가되면 값이 NaN으로 매핑된다.
states = ['California', 'Ohio', 'Oregon', 'Texas']
s3 = pd.Series(sdata, index=states)
s3.name = "population"
s3.index.name = "state"
print(s3)

# 데이터가 Null 인지 확인
print('-----')
print(pd.isnull(s3)) # s3.isnull() 도 가능
```

```
state
California      NaN
Ohio            35000.0
Oregon          16000.0
Texas           71000.0
Name: population, dtype: float64
-----
state
California      True
Ohio            False
Oregon          False
Texas           False
Name: population, dtype: bool
```

```
In [42]: # Value만 가지고 순서대로 연산
print(s2.values)
print(s3.values)
print('-----')
print(s2.values + s3.values)
```

```
[35000 16000 71000  5000]
[   nan  35000.  16000.  71000.]
-----
[   nan  51000.  87000.  76000.]
```

```
In [43]: # Index 정보를 가지고 연산
# 서로 가지고 있지 않은 데이터는 Index는 반영되지만 값은 반영 안됨.

print(s2 + s3)
```

```
state
California      NaN
Ohio            70000.0
Oregon          32000.0
Texas           142000.0
Utah            NaN
Name: population, dtype: float64
```

# !!DataFrame

- **Multi-Series**
  - 동일한 Row 인덱스를 사용하는 복수 Series
  - Series를 value로 가지는 dict
- **2차원 행렬**
  - DataFrame을 행렬로 생각하면 각 Series는 행렬의 Column의 역할
  - (Row) Index와 Column Index를 가진다.
- NumPy **ndArray** 와 차이점
  - 각 Column(Series)마다 type이 달라도 된다.

```
In [51]: # Dict 자료형으로 데이터를 생성
data = {
    'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
    'year': [2000, 2001, 2002, 2000, 2001, 2002],
    'pop': [1.5, 1.7, 3.6, 2.2, 2.4, 2.9]
}

# Dict 자료형 -> dataframe 형 생성
df2 = pd.DataFrame(data,
                    columns=['year', 'state', 'pop'])
df2
```

```
Out[51]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2000	Nevada	2.2
4	2001	Nevada	2.4
5	2002	Nevada	2.9

## 데이터 접근(indexing)

```
In [61]: print(df2.columns)
print(df2.index)

Index(['year', 'state', 'pop'], dtype='object')
RangeIndex(start=0, stop=6, step=1)
```

```
In [55]: print(df2["state"]) # 또는 df.state # 데이터를 series 로 취급
print('-----')
print(type(df2['state']))
```

```
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
-----
<class 'pandas.core.series.Series'>
```

```
In [70]: print(df2.ix[2])
```

```
year      2002
state     Ohio
pop        3.6
Name: 2, dtype: object
```

```
In [79]: print(df2.ix[2, ['year', 'pop']])
```

```
year      2002
pop        3.6
Name: 2, dtype: object
```

```
In [80]: print(df2.ix[[1, 2], ["state", "pop"]])
```

```
   state  pop
1  Ohio  1.7
2  Ohio  3.6
```

```
In [93]: print(df2.ix[:, ["state"]]) # ix를 사용하는 경우 열은 ':' 필요함
```

```
   state
0  Ohio
1  Ohio
2  Ohio
3  Nevada
4  Nevada
5  Nevada
```

```
In [94]: print(df2.ix[[3, 4], :2])
```

```
   year  state
3  2000  Nevada
4  2001  Nevada
```

```
In [95]: print(df2[["state"]])
# 대괄호가 2개이면 dataframe으로 취급

type(df2[["state"]])
```

```
   state
0  Ohio
1  Ohio
2  Ohio
3  Nevada
4  Nevada
5  Nevada
```

```
Out[95]: pandas.core.frame.DataFrame
```

참고!

Table 5-6. Indexing options with DataFrame

Type	Notes
obj[val]	Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
obj.ix[val]	Selects single row of subset of rows from the DataFrame.
obj.ix[:, val]	Selects single column of subset of columns.
obj.ix[val1, val2]	Select both rows and columns.
reindex method	Conform one or more axes to new indexes.
xs method	Select single row or column as a Series by label.
icol, irow methods	Select single column or row, respectively, as a Series by integer location.
get_value, set_value methods	Select single value by row and column label.

Index Label이 없는 경우 :

Label이 지정되지 않는 경우에는 integer slicing을 label slicing으로 간주하여 마지막 값을 포함한다

## 데이터 update

```
In [99]: df2['debt'] = np.arange(6)
df2
```

Out[99]:

	year	state	pop	debt
0	2000	Ohio	1.5	0
1	2001	Ohio	1.7	1
2	2002	Ohio	3.6	2
3	2000	Nevada	2.2	3
4	2001	Nevada	2.4	4
5	2002	Nevada	2.9	5



```
In [100]: df2['debt'] = pd.Series([-1.2, -1.5, -1.7], index=[2, 3, 4])
df2
```

```
Out[100]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	-1.2
3	2000	Nevada	2.2	-1.5
4	2001	Nevada	2.4	-1.7
5	2002	Nevada	2.9	NaN

## column 추가

```
In [121]: df2['eastern'] = df2.state == 'Ohio'
df2
```

```
Out[121]:
```

	year	state	pop	debt	eastern
0	2000	Ohio	1.5	NaN	True
1	2001	Ohio	1.7	NaN	True
2	2002	Ohio	3.6	-1.2	True
3	2000	Nevada	2.2	-1.5	False
4	2001	Nevada	2.4	-1.7	False
5	2002	Nevada	2.9	NaN	False

```
In [120]: df2.drop(['eastern'],1,inplace=True)
df2
```

```
Out[120]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	-1.2
3	2000	Nevada	2.2	-1.5
4	2001	Nevada	2.4	-1.7
5	2002	Nevada	2.9	NaN

## column 제거

```
In [122]: del df2['eastern'] # 또는 df2.drop('eastern')
df2
```

```
Out[122]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	-1.2
3	2000	Nevada	2.2	-1.5
4	2001	Nevada	2.4	-1.7
5	2002	Nevada	2.9	NaN

## drop 메소드를 사용한 row/column 제거

- `del` 함수
  - inplace 연산
- `drop` 메소드
  - 삭제된 Series/DataFrame 출력
  - Series는 Row 삭제
  - DataFrame은 axis 인수로 Row/Column 선택
    - `axis=0` (디폴트): Row
    - `axis=1`: Column

```
In [123]: df = pd.DataFrame(np.arange(16).reshape((4, 4)),
                             index=['Ohio', 'Colorado', 'Utah', 'New York'],
                             columns=['one', 'two', 'three', 'four'])
df
```

```
Out[123]:
```

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

```
In [127]: df.drop(['Colorado', 'Ohio'],axis=0)
#row 삭제
```

```
Out[127]:
```

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

```
In [128]: df.drop(['two', 'four'], axis=1)
#column 삭제
```

```
Out[128]:
```

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

## nested dict를 사용한 dataframe 생성

```
In [130]: pop = {
    'Nevada': {
        2001: 2.4,
        2002: 2.9
    },
    'Ohio': {
        2000: 1.5,
        2001: 1.7,
        2002: 3.6
    }
}

df3 = pd.DataFrame(pop)
df3
```

```
Out[130]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [131]: # Transpose
df3.T
```

```
Out[131]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

## 인덱스 설정 및 제거

- `set_index` : 기존의 행 인덱스를 제거하고 데이터 열 중 하나를 인덱스로 설정
  - `set_index` 명령으로 다른 column을 인덱스로 설정할 수 있다. 이 때 기존 인덱스는 없어진다.
- `reset_index` : 기존의 행 인덱스를 제거하고 인덱스를 마지막 데이터 열로 추가
  - `reset_index` 명령으로 인덱스 열을 보통의 자료열로 넣을 수 있다. 이 때 인덱스 열은 자료열의 가장 선두로 삽입된다. 인덱스는 숫자로 된 디폴트 인덱스가 된다.
  - `drop=True` 로 설정하면 인덱스 열을 보통의 자료열로 올리는 것이 아니라 그냥 버리게 된

```
In [89]: np.random.seed(0)
```

```
# randint(최소값, 최대값, 만들 크기)
df = pd.DataFrame(np.random.randint(1, 10, (10, 4)),
                  columns=["C1", "C2", "C3", "C4"])
df
```

```
Out[89]:
```

	C1	C2	C3	C4
0	6	1	4	4
1	8	4	6	3
2	5	8	7	9
3	9	2	7	8
4	8	9	2	6
5	9	5	4	1
6	4	6	1	3
7	4	9	2	4
8	4	4	8	1
9	2	1	5	8

## 데이터 열 중 하나를 인덱스로 설정

```
In [90]: df1 = df.set_index("C1")
df1
```

```
Out[90]:
```

	C2	C3	C4
C1			
6	1	4	4
8	4	6	3
5	8	7	9
9	2	7	8
8	9	2	6
9	5	4	1
4	6	1	3
4	9	2	4
4	4	8	1
2	1	5	8

In [91]: df1.reset\_index()

Out[91]:

	C1	C2	C3	C4
0	6	1	4	4
1	8	4	6	3
2	5	8	7	9
3	9	2	7	8
4	8	9	2	6
5	9	5	4	1
6	4	6	1	3
7	4	9	2	4
8	4	4	8	1
9	2	1	5	8