

# Pandas

# Pandas

구조화된 데이터의 처리를 지원하는 Python 라이브러리  
**Python계의 엑셀!**

# Pandas

- 구조화된 데이터의 처리를 지원하는 Python 라이브러리
- 고성능 Array 계산 라이브러리인 Numpy와 통합하여, 강력한 “스프레드시트” 처리 기능을 제공
- 인덱싱, 연산용 함수, 전처리 함수 등을 제공함

# 데이터 로딩

```
In [1]: import pandas as pd #라이브러리 호출
```

```
In [2]: data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data' #Data URL  
df_data = pd.read_csv(data_url, sep='\\s+', header = None) #csv 파일 데이터 로드, separate는 빈공간으로 지정하고, Column은 없음
```

```
In [3]: df_data.head() #처음 다섯줄 출력
```

Out [3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

# Series

# Pandas의 구성

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	weight_0
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	1
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	1
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	1
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	1
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	1

Series

DataFrame 중 하나의 Column에 해당하는  
데이터의 모음 Object

DataFrame

Data Table 전체를 포함하는 Object

# 일반적인 padas의 활용

```
In [1]: import pandas as pd #라이브러리 호출
```

```
In [2]: data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data' #Data URL  
df_data = pd.read_csv(data_url, sep='\\s+', header = None) #csv 파일 데이터 로드, separate는 빈공간으로 지정하고, Column은 없음
```

```
In [3]: df_data.head() #처음 다섯줄 출력
```

Out [3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

기존 데이터를 불러와서  
**DataFrame**을 생성

기본적인 것 부터  
해보자

# Series

Column Vector를 표현하는 object

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

```
In [ ]: example_obj = Series()
```

Init signature: Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)

Docstring:

One-dimensional ndarray with axis labels (including time series).

Shift + TAB

# Series

```
In [2]: list_data = [1,2,3,4,5]
example_obj = Series(data = list_data)
example_obj
```

```
Out[2]: 0    1
         1    2
         2    3      data
         3    4
         4    5
dtype: int64
```

Index

data

Data type

# Series

index	values
A	5
B	6
C	12
D	-5
E	6.7

- Subclass of `numpy.ndarray`
- Data: any type
- Index labels need not be ordered
- Duplicates are possible (but result in reduced functionality)

# Series

```
In [3]: list_data = [1,2,3,4,5]
list_name = ["a","b","c","d","e"]
example_obj = Series(data = list_data, index=list_name)
example_obj
```

index 이름을 지정

```
Out[3]: a    1
         b    2
         c    3
         d    4
         e    5
dtype: int64
```

# Series

Data와 index 이름을 지정

```
In [4]: dict_data = {"a":1, "b":2, "c":3, "d":4, "e":5}  
example_obj = Series(dict_data, dtype=np.float32, name="example_data")  
example_obj
```

data type 설정

series 이름 설정

```
Out[4]: a    1.0  
        b    2.0  
        c    3.0  
        d    4.0  
        e    5.0  
Name: example_data, dtype: float32
```

# Series

**data index에 접근하기**

```
In [5]: example_obj["a"]
```

```
Out[5]: 1.0
```

**data index에 값 할당하기**

```
In [6]: example_obj["a"] = 3.2  
example_obj
```

```
Out[6]: a    3.2  
        b    2.0  
        c    3.0  
        d    4.0  
        e    5.0  
Name: example_data, dtype: float32
```

# Series

```
example_obj.values
```

값 리스트만

```
array([ 3.2000005,  2.           ,  3.           ,  4.           ,  5.           ],  
      dtype=float32)
```

```
example_obj.index
```

Index 리스트만

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

```
example_obj.name = "number"  
example_obj.index.name = "alphabet"  
example_obj
```

Data에 대한 정보를 저장

```
alphabet  
a    3.2  
b    2.0  
c    3.0  
d    4.0  
e    5.0
```

```
Name: number, dtype: float32
```

# Series

```
dict_data_1 = {"a":1, "b":2, "c":3, "d":4, "e":5}  
indexes = ["a","b","c","d","e","f","g","h"]  
series_obj_1 = Series(dict_data_1, index=indexes)  
series_obj_1          index 값을 기준으로 series 생성
```

```
a    1.0  
b    2.0  
c    3.0  
d    4.0  
e    5.0  
f    NaN  
g    NaN  
h    NaN  
dtype: float64
```

# Dataframe Overview

# Pandas의 구성

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	weight_0
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	1
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	1
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	1
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	1
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	1

Series

DataFrame 중 하나의 Column에 해당하는  
데이터의 모음 Object

DataFrame

Data Table 전체를 포함하는 Object

# Dataframe

columns	foo	bar	baz	qux
index				
A	→ 0	x	2.7	True
B	→ 4	y	6	True
C	→ 8	z	10	False
D	→ -12	w	NA	False
E	→ 16	a	18	False

- NumPy array-like
- Each column can have a different type
- Row and column index
- Size mutable: insert and delete columns

# DataFrame

Series를 모아서 만든 Data Table = 기본 2차원

```
In [4]: from pandas import Series, DataFrame  
import pandas as pd  
import numpy as np
```

```
In [ ]: DataFrame()
```

Shift + TAB

```
In [3]: Init signature: DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

**Docstring:**

Two-dimensional size-mutable, potentially heterogeneous tabular data  
structure with labeled axes (rows and columns). Arithmetic operations

# DataFrame

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

**column\_name : data**

```
In [2]: # Example from - https://christianbon.com/python/pandas_map_values_to_values.html/  
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],  
            'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],  
            'age': [42, 52, 36, 24, 73],  
            'city': ['San Francisco', 'Baltimore', 'Miami', 'Douglas', 'Boston']}  
df = pd.DataFrame(raw_data, columns = ['first_name', 'last_name', 'age', 'city'])  
df
```

Out[2]:

	first_name	last_name	age	city
0	Jason	Miller	42	San Francisco
1	Molly	Jacobson	52	Baltimore
2	Tina	Ali	36	Miami
3	Jake	Milner	24	Douglas
4	Amy	Cooze	73	Boston

# DataFrame

```
In [3]: DataFrame(raw_data, columns = ["age", "city"])
```

Out [3]:

	age	city
0	42	San Franciso
1	52	Baltimore
2	36	Miami
3	24	Douglas
4	73	Boston

column 선택

```
In [4]: DataFrame(raw_data, columns = ["first_name", "last_name", "age", "city", "debt"])
```

Out [4]:

	first_name	last_name	age	city	debt
0	Jason	Miller	42	San Franciso	NaN
1	Molly	Jacobson	52	Baltimore	NaN
2	Tina	Ali	36	Miami	NaN
3	Jake	Milner	24	Douglas	NaN
4	Amy	Cooze	73	Boston	NaN

새로운 column 추가

# DataFrame

```
df = DataFrame(raw_data, columns = ["first_name", "last_name", "age", "city", "debt"])
df.first_name
```

```
0    Jason
1    Molly
2    Tina
3    Jake
4    Amy
Name: first_name, dtype: object
```

## column 선택 – series 추출

```
df["first_name"]
```

```
0    Jason
1    Molly
2    Tina
3    Jake
4    Amy
Name: first_name, dtype: object
```

## column 선택 – series 추출

# DataFrame

```
df
```

	first_name	last_name	age	city	debt
0	Jason	Miller	42	San Francisco	NaN
1	Molly	Jacobson	52	Baltimore	NaN
2	Tina	Ali	36	Miami	NaN
3	Jake	Milner	24	Douglas	NaN
4	Amy	Cooze	73	Boston	NaN

```
df.loc[1]
```

## loc – index location

```
first_name      Molly
last_name       Jacobson
age              52
city            Baltimore
debt             NaN
Name: 1, dtype: object
```

```
df["age"].iloc[1:]
```

## iloc – index position

```
1    52
2    36
3    24
4    73
Name: age, dtype: int64
```

# DataFrame

**loc은 index 이름, iloc은 index number**

```
# Example from - https://stackoverflow.com/questions/31593201/pandas-iloc-vs-ix-vs-loc-explanation
s = pd.Series(np.nan, index=[49, 48, 47, 46, 45, 1, 2, 3, 4, 5])
s.loc[:3]
```

```
49    NaN
48    NaN
47    NaN
46    NaN
45    NaN
1     NaN
2     NaN
3     NaN
dtype: float64
```

```
s.iloc[:3]
```

```
49    NaN
48    NaN
47    NaN
dtype: float64
```

# DataFrame

## Column에 새로운 데이터 할당

```
df.debt = df.age > 40  
df
```

	first_name	last_name	age	city	debt
0	Jason	Miller	42	San Francisco	True
1	Molly	Jacobson	52	Baltimore	True
2	Tina	Ali	36	Miami	False
3	Jake	Milner	24	Douglas	False
4	Amy	Gooze	73	Boston	True

```
df.T
```

# DataFrame

```
In [13]: df.T
```

```
Out[13]:
```

	0	1	2	3	4
first_name	Jason	Molly	Tina	Jake	Amy
last_name	Miller	Jacobson	Ali	Milner	Cooze
age	42	52	36	24	73
city	San Francisco	Baltimore	Miami	Douglas	Boston
debt	True	True	False	False	True

## transpose

```
In [14]: df.values
```

```
Out[14]: array([['Jason', 'Miller', 42, 'San Francisco', True],
                 ['Molly', 'Jacobson', 52, 'Baltimore', True],
                 ['Tina', 'Ali', 36, 'Miami', False],
                 ['Jake', 'Milner', 24, 'Douglas', False],
                 ['Amy', 'Cooze', 73, 'Boston', True]], dtype=object)
```

## 값 출력

```
In [20]: df.to_csv()
```

```
Out[20]: ',first_name,last_name,age,city,debt\n0,Jason,Miller,42,San Francisco,True\n1,Molly,Jacobson,52,Baltimore,True\n2,Tina,Ali,36,Miami,False\n3,Jake,Milner,24,Douglas,False\n4,Amy,Cooze,73,Boston,True\n'
```

## CSV 변환

# DataFrame

## Column을 삭제함

```
del df[ "debt" ]
```

```
df
```

	first_name	last_name	age	city
0	Jason	Miller	42	San Francisco
1	Molly	Jacobson	52	Baltimore
2	Tina	Ali	36	Miami
3	Jake	Milner	24	Douglas
4	Amy	Cooze	73	Boston

# DataFrame

```
# Example from Python for data analysis
|
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
Column 값           Nested dict에서는
Index 값           DataFrame(pop)
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

# Selection & Drop

# Selection with column names

```
df[ "account" ].head( 3 )
```

한개의 column 선택시

```
0    211829  
1    320563  
2    648336
```

```
Name: account, dtype: int64
```

1개 이상의 column 선택

```
df[ [ "account", "street", "state" ] ].head( 3 )
```

	account	street	state
0	211829	34456 Sean Highway	Texas
1	320563	1311 Alvis Tunnel	NorthCarolina
2	648336	62184 Schamberger Underpass Apt. 231	Iowa

# Selection with index number

df[:3] column 이름 없이 사용하는 index number는 row 기준 표시

	account	name	street	city	state	postal-code	Jan	Feb	Mar
0	211829	Kerluke, Koepp and Hilpert	34456 Sean Highway	New Jaycob	Texas	28752	10000	62000	35000
1	320563	Walter-Trantow	1311 Alvis Tunnel	Port Khadijah	NorthCarolina	38365	95000	45000	35000
2	648336	Bashirian, Kunde and Price	62184 Schamberger Underpass Apt. 231	New Lilianland	Iowa	76517	91000	120000	35000

df["account"][:3] column이름과 함께 row index 사용시, 해당 column만

```
0    211829
1    320563
2    648336
Name: account, dtype: int64
```

# Series selection

```
account_serires = df["account"]  
account_serires[:3]
```

```
0    211829  
1    320563  
2    648336  
Name: account, dtype: int64
```

```
account_serires[[0,1,2]]
```

```
0    211829      1개 이상의  
1    320563      index  
2    648336  
Name: account, dtype: int64
```

```
account_serires[account_serires<250000]
```

```
0    211829  
3    109996  
4    121213  
5    132971  
6    145068  
7    205217  
8    209744  
9    212303  
10   214098  
11   231907  
12   242368
```

```
Name: account, dtype: int64
```

**Boolean index**

# Index 변경

```
df.index = df[ "account" ]  
del df[ "account" ]  
df.head()
```

	<b>name</b>	<b>street</b>	<b>city</b>	<b>state</b>	<b>postal-code</b>	<b>Jan</b>	<b>Feb</b>	<b>Mar</b>
<b>account</b>								
<b>211829</b>	Kerluke, Koepp and Hilpert	34456 Sean Highway	New Jaycob	Texas	28752	10000	62000	35000
<b>320563</b>	Walter-Trantow	1311 Alvis Tunnel	Port Khadijah	NorthCarolina	38365	95000	45000	35000
<b>648336</b>	Bashirian, Kunde and Price	62184 Schamberger Underpass Apt. 231	New Lilianland	Iowa	76517	91000	120000	35000
<b>109996</b>	D'Amore, Gleichner and Bode	155 Fadel Crescent Apt. 144	Hyattburgh	Maine	46021	45000	120000	10000
<b>121213</b>	Bauch-Goldner	7274 Marissa Common	Shanahanchester	California	49681	162000	120000	35000

# Basic, loc, iloc selection

Column 과  
index number

```
df[["name", "street"]][:2]
```

	name	street
account		
211829	Kerluke, Koepp and Hilpert	34456 Sean Highway
320563	Walter-Trantow	1311 Alvis Tunnel

Column number와  
index number

```
df.iloc[:2, :2]
```

	name	street
account		
211829	Kerluke, Koepp and Hilpert	34456 Sean Highway
320563	Walter-Trantow	1311 Alvis Tunnel

```
df.loc[[211829, 320563], ["name", "street"]]
```

Column 과  
index name

	name	street
account		
211829	Kerluke, Koepp and Hilpert	34456 Sean Highway
320563	Walter-Trantow	1311 Alvis Tunnel

# index 재설정

```
df.index = list(range(0,15))  
df.head()
```

	<b>name</b>	<b>street</b>	<b>city</b>	<b>state</b>	<b>postal-code</b>
<b>0</b>	Kerluke, Koepp and Hilpert	34456 Sean Highway	New Jaycob	Texas	28752
<b>1</b>	Walter-Trantow	1311 Alvis Tunnel	Port Khadijah	NorthCarolina	38365
<b>2</b>	Bashirian, Kunde and Price	62184 Schamberger Underpass Apt. 231	New Lilianland	Iowa	76517

# Data drop

```
df.drop(1) Index number로 drop
```

	<b>name</b>	<b>street</b>	<b>city</b>	<b>s</b>
<b>0</b>	Kerluke, Koepp and Hilpert	34456 Sean Highway	New Jaycob	T
<b>2</b>	Bashirian, Kunde and Price	62184 Schamberger Underpass Apt. 231	New Lilianland	Ic

# Data drop

df.drop([0,1,2,3]) 한개 이상의 Index number로 drop

	account	name	street	city	state
4	121213	Bauch-Goldner	7274 Marissa Common	Shanahanchester	California
5	132971	Williamson, Schumm and Hettinger	89403 Casimer Spring	Jeremieburgh	Arkansas
6	145068	Casper LLC	340 Consuela Bridge Apt. 400	Lake Gabriellaton	Mississipi

# Data drop

axis 지정으로 축을 기준으로 drop → column 중에 “city”

```
df.drop("city",axis=1) # df.drop(["city", "state"],axis=1)
```

	<b>name</b>	<b>street</b>	<b>state</b>	<b>postal-code</b>
0	Kerluke, Koepp and Hilpert	34456 Sean Highway	Texas	28752
1	Walter-Trantow	1311 Alvis Tunnel	NorthCarolina	38365
2	Bashirian, Kunde and Price	62184 Schamberger Underpass Apt. 231	Iowa	76517

# Dataframe Operations

# Series operation

```
s1 = Series(  
    range(1,6), index=list("abced"))  
s1  
  
a    1  
b    2  
c    3  
e    4  
d    5  
dtype: int64
```

```
s2 = Series(  
    range(5,11), index=list("bcedef"))  
s2  
  
b    5  
c    6  
e    7  
d    8  
e    9  
f   10  
dtype: int64
```

s1.add(s2)

```
a      NaN  
b      7.0  
c      9.0  
d     13.0  
e     11.0  
e     13.0  
f      NaN  
dtype: float64
```

s1 + s2

```
a      NaN  
b      7.0  
c      9.0  
d     13.0  
e     11.0  
e     13.0  
f      NaN  
dtype: float64
```

index 으로 기준으로 연산수행

겹치는 index가 없을 경우  
NaN값으로 반환

# Dataframe operation

```
df1 = DataFrame(  
    np.arange(9).reshape(3,3),  
    columns=list("abc"))  
df1
```

	a	b	c
0	0	1	2
1	3	4	5
2	6	7	8

```
df2 = DataFrame(  
    np.arange(16).reshape(4,4),  
    columns=list("abcd"))  
df2
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

df1 + df2

	a	b	c	d
0	0.0	2.0	4.0	NaN
1	7.0	9.0	11.0	NaN
2	14.0	16.0	18.0	NaN
3	NaN	NaN	NaN	NaN

df1.add(df2, fill\_value=0)

	a	b	c	d
0	0.0	2.0	4.0	3.0
1	7.0	9.0	11.0	7.0
2	14.0	16.0	18.0	11.0
3	12.0	13.0	14.0	15.0

df는 column과 index를 모두 고려

add operation을 쓰면 NaN값 0으로 변환  
Operation types: add, sub, div, mul

# Series + Dataframe

```
df = DataFrame(  
    np.arange(16).reshape(4,4),  
    columns=list("abcd"))  
  
df
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

10 11 12 13  
10 11 12 13  
10 11 12 13  
10 11 12 13



```
s = Series(  
    np.arange(10,14),  
    index=list("abcd"))  
  
s
```

a 10  
b 11  
c 12  
d 13  
dtype: int64

df + s

	a	b	c	d
0	10	12	14	16
1	14	16	18	20
2	18	20	22	24
3	22	24	26	28

column을 기준으로  
broadcasting이 발생함

# Series + Dataframe

```
df = DataFrame(  
    np.arange(16).reshape(4,4),  
    columns=list("abcd"))  
df
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

```
s2 = Series(np.arange(10,14))  
s2  
  
0    10  
1    11  
2    12  
3    13  
dtype: int64
```

```
df + s2
```

	a	b	c	d	0	1	2	3
0	NaN							
1	NaN							
2	NaN							
3	NaN							

```
df.add(s2, axis=0)
```

	a	b	c	d
0	10	11	12	13
1	15	16	17	18
2	20	21	22	23
3	25	26	27	28

axis를 기준으로  
row broadcasting 실행

lambda, map, apply

# Lambda 함수

- 한 줄로 함수를 표현하는 익명 함수 기법
- Lisp 언어에서 시작된 기법으로 오늘날 현대언어에 많이 사용

lambda argument : expression

```
lambda x,y: x + y
```

```
def f(x , y):  
    return x + y
```

```
f = lambda x,y: x + y  
f(1,4)
```

# Lambda 함수

```
f = lambda x: x / 2  
f(3)
```

1.5

```
f = lambda x: x ** 2  
f(3)
```

9

```
(lambda x: x +1)(5)
```

6

하나의 argument만 처리하는  
lambda함수

이름을 할당하지 않는 lambda 함수

# map 함수

- 함수와 sequence형 데이터를 인자로 받아
- 각 element마다 입력받은 함수를 적용하여 list로 반환
- 일반적으로 함수를 lambda형태로 표현함

map(function, sequence)

```
ex = [1,2,3,4,5]
f = lambda x: x ** 2
list(map(f, ex))
```

[1, 4, 9, 16, 25]

# map 함수

```
f = lambda x, y: x + y  
list(map(f, ex, ex))
```

```
[2, 4, 6, 8, 10]
```

두 개이상의 argument가 있을 때는  
두 개의 sequence형을 써야함

익명 함수 그대로 사용할 수 있음

```
list(map(lambda x: x+x, ex))  
#python 3에는 list를 꼭 붙여줘야함
```

```
[2, 4, 6, 8, 10]
```

# map for series

- Pandas의 series type의 데이터에도 map 함수 사용 가능
- function 대신 dict, sequence형 자료등으로 대체 가능

```
s1 = Series(np.arange(10))  
s1.head(5)
```

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

```
s1.map(lambda x: x**2).head(5)
```

```
0    0  
1    1  
2    4  
3    9  
4   16  
dtype: int64
```

# map for series

```
z = {1: 'A', 2: 'B', 3: 'C'}  
s1.map(z).head(5)
```

```
0      NaN  
1        A  
2        B  
3        C  
4      NaN  
dtype: object
```

dict type으로  
데이터 교체  
없는 값은 NaN

```
s2 = Series(np.arange(10,20))  
s1.map(s2).head(5)
```

```
0    10  
1    11  
2    12  
3    13  
4    14  
dtype: int64
```

같은 위치의 데이터를  
s2로 전환

# Example - map for series

```
df = pd.read_csv("wages.csv")
df.head()
```

	earn	height	sex	race	ed	age
0	79571.299011	73.89	male	white	16	49
1	96396.988643	66.23	female	white	16	62
2	48710.666947	63.77	female	white	16	33
3	80478.096153	63.22	female	other	16	95
4	82089.345498	63.08	female	white	17	43

# Example - map for series

```
df.sex.unique()
```

```
array(['male', 'female'], dtype=object)
```

```
df["sex_code"] = df.sex.map({"male":0, "female":1})  
df.head(5)
```

성별 str → 성별 code

	earn	height	sex	race	ed	age	sex_code
0	79571.299011	73.89	male	white	16	49	0
1	96396.988643	66.23	female	white	16	62	1
2	48710.666947	63.77	female	white	16	33	1
3	80478.096153	63.22	female	other	16	95	1
4	82089.345498	63.08	female	white	17	43	1

# Replace function

- Map 함수의 기능중 데이터 변환 기능만 담당
- 데이터 변환시 많이 사용하는 함수

```
df.sex.replace(  
    {"male":0, "female":1}  
).head()
```

0 0      **dict type 적용**  
1 1  
2 1  
3 1  
4 1  
Name: sex, dtype: int64

```
df.sex.replace(  
    ["male", "female"],  
    [0,1], inplace=True)  
df.head(5)
```

	earn	height	sex	race	ed	age	sex_code
0	79571.299011	73.89	0	white	16	49	0
1	96396.988643	66.23	1	white	16	62	1
2	48710.666947	63.77	1	white	16	33	1

**inplace ← 데이터 변환결과를 적용**

**Target list**  
**Conversion list**

# apply for dataframe

- map과 달리, series 전체(column)에 해당 함수를 적용
- 입력값이 series 데이터로 입력받아 handling 가능

```
df_info = df[["earn", "height", "age"]]  
df_info.head()
```

	earn	height	age
0	79571.299011	73.89	49
1	96396.988643	66.23	62
2	48710.666947	63.77	33
3	80478.096153	63.22	95
4	82089.345498	63.08	43

```
f = lambda x : x.max() - x.min()  
df_info.apply(f)
```

```
earn      318047.708444  
height      19.870000  
age       73.000000  
dtype: float64
```

각 column 별로 결과값 반환

# apply for dataframe

- 내장 연산 함수를 사용할 때도 똑같은 효과를 거둘 수 있음
- mean, std 등 사용가능

```
df_info.sum()
```

```
earn      4.474344e+07  
height    9.183125e+04  
age       6.250800e+04  
dtype: float64
```

```
df_info.apply(sum)
```

```
earn      4.474344e+07  
height    9.183125e+04  
age       6.250800e+04  
dtype: float64
```

# apply for dataframe

- scalar 값 이외에 series값의 반환도 가능함

```
def f(x):
    return Series([x.min(), x.max()], index=["min", "max"])
df_info.apply(f)
```

	earn	height	age
min	-98.580489	57.34	22
max	317949.127955	77.21	95

# applymap for dataframe

- series 단위가 아닌 element 단위로 함수를 적용함
- series 단위에 apply를 적용시킬 때와 같은 효과

```
f = lambda x : -x  
df_info.applymap(f).head(5)
```

	earn	height	age
0	-79571.299011	-73.89	-49
1	-96396.988643	-66.23	-62
2	-48710.666947	-63.77	-33
3	-80478.096153	-63.22	-95
4	-82089.345498	-63.08	-43

```
f = lambda x : -x  
df_info["earn"].apply(f).head(5)
```

```
0    -79571.299011  
1    -96396.988643  
2    -48710.666947  
3    -80478.096153  
4    -82089.345498  
Name: earn, dtype: float64
```

# Pandas Built-in functions

# describe

- Numeric type 데이터의 요약 정보를 보여줌

```
df = pd.read_csv("wages.csv")
df.head()
```

	earn	height	sex	race	ed	age
0	79571.299011	73.89	male	white	16	49
1	96396.988643	66.23	female	white	16	62
2	48710.666947	63.77	female	white	16	33
3	80478.096153	63.22	female	other	16	95
4	82089.345498	63.08	female	white	17	43

```
df.describe()
```

	earn	height	ed	age
count	1379.000000	1379.000000	1379.000000	1379.000000
mean	32446.292622	66.592640	13.354605	45.328499
std	31257.070006	3.818108	2.438741	15.789715
min	-98.580489	57.340000	3.000000	22.000000
25%	10538.790721	63.720000	12.000000	33.000000
50%	26877.870178	66.050000	13.000000	42.000000
75%	44506.215336	69.315000	15.000000	55.000000
max	317949.127955	77.210000	18.000000	95.000000

# unique

- series data의 유일한 값을 list를 반환함

```
df.race.unique()    유일한 인종의 값 list
```

```
array(['white', 'other', 'hispanic', 'black'], dtype=object)
```

```
np.array(dict(enumerate(df["race"].unique())))    dict type으로 index
```

```
array({0: 'white', 1: 'other', 2: 'hispanic', 3: 'black'}, dtype=object)
```

```
value = list(map(int, np.array(list(enumerate(df["race"].unique())))[ :, 0].tolist()))
key = np.array(list(enumerate(df["race"].unique()))), dtype=str)[ :, 1].tolist()
```

```
value, key    label index 값과 label 값 각각 추출
```

```
([0, 1, 2, 3], ['white', 'other', 'hispanic', 'black'])
```

# unique

label str → index 값으로 변환

```
df["race"].replace(to_replace=key, value=value, inplace=True)
```

```
value = list(map(int, np.array(list(enumerate(df["sex"].unique()))))[:, 0].tolist()))
key = np.array(list(enumerate(df["sex"].unique())), dtype=str)[:, 1].tolist()
```

```
value, key
```

성별에 대해서도 동일하게 적용

```
([0, 1], ['male', 'female'])
```

```
df["sex"].replace(to_replace=key, value=value, inplace=True)
df.head(5)
```

	earn	height	sex	race	ed	age
0	79571.299011	73.89	0	0	16	49
1	96396.988643	66.23	1	0	16	62

"sex"와 "race" column의  
index labelling

# sum

- 기본적인 column 또는 row 값의 연산을 지원
- sub, mean, min, max, count, median, mad, var 등

`df.sum(axis=0)` | column 별

```
earn      4.474344e+07  
height    9.183125e+04  
sex       8.590000e+02  
race      5.610000e+02  
ed        1.841600e+04  
age       6.250800e+04  
dtype: float64
```

`df.sum(axis=1)` row 별

```
0          79710.189011  
1          96542.218643  
2          48824.436947  
3          80654.316153  
4          82213.425498  
5          15423.882901  
6          47231.711821
```

# isnull

- column 또는 row 값의 NaN (null) 값의 index를 반환함

```
df.isnull()
```

	earn	height	sex	race	ed	age
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False

```
df.isnull().sum()
```

earn 0 Null인  
height 0 값의 합  
sex 0  
race 0  
ed 0  
age 0  
dtype: int64

# sort\_values

- column 값을 기준으로 데이터를 sorting

```
df.sort_values(["age", "earn"], ascending=True).head(10)
```

	earn	height	sex	race	ed	age
1038	-56.321979	67.81	0	2	10	22
800	-27.876819	72.29	0	0	12	22
963	-25.655260	68.90	0	0	12	22
1105	988.565070	64.71	1	0	12	22
801	1000.221504	64.09	1	0	12	22

ascending → 오름차순

# Correlation & Covariance

- 상관계수와 공분산을 구하는 함수
- corr, cov, corrwith

```
: df.age.corr(df.earn)
```

```
: 0.074003491778360575
```

```
: df.age.cov(df.earn)
```

```
: 36523.6992104089
```

$$\rho_{X,Y} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

```
df.corrwith(df.earn)
```

earn	1.000000
height	0.291600
sex	-0.337328
race	-0.063977
ed	0.350374
age	0.074003

```
dtype: float64
```

# Correlation & Covariance

```
df.corr()
```

	<b>earn</b>	<b>height</b>	<b>sex</b>	<b>race</b>	<b>ed</b>	<b>age</b>
<b>earn</b>	1.000000	0.291600	-0.337328	-0.063977	0.350374	0.074003
<b>height</b>	0.291600	1.000000	-0.703672	-0.045974	0.114047	-0.133727
<b>sex</b>	-0.337328	-0.703672	1.000000	0.000858	-0.061747	0.070036
<b>race</b>	-0.063977	-0.045974	0.000858	1.000000	-0.049487	-0.056879
<b>ed</b>	0.350374	0.114047	-0.061747	-0.049487	1.000000	-0.129802
<b>age</b>	0.074003	-0.133727	0.070036	-0.056879	-0.129802	1.000000