Python for Data Analytics

Pandas I

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 11 – 15, 2021

Outline

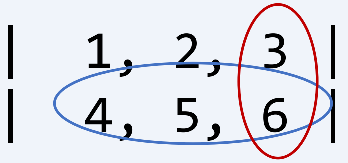
- Why Pandas?
- Pandas Series
- Pandas DataFrame
  - Creating DataFrame
  - Manipulating Columns
  - Manipulating Rows
  - Arithmetic operations
  - Group Aggregation
  - Hierarchical Indexing
  - Combining and Merging
  - Time Series Data

Why Pandas?

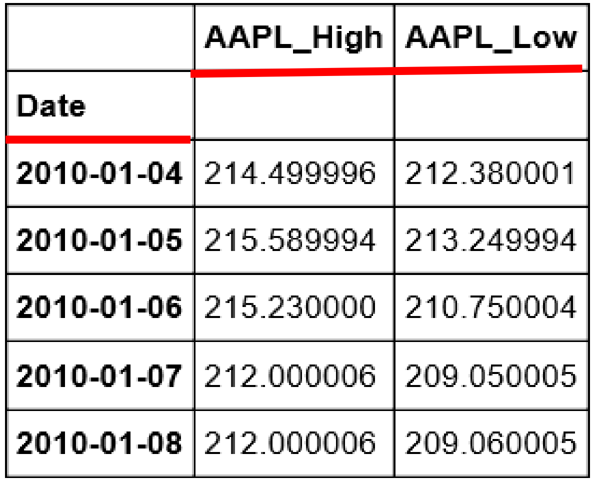
Limitations in NumPy

- Remember? Array slicing in NumPy

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a[1,:]
array([[4, 5, 6]])
>>> a[:,2]
array([3, 6])
>>> a[-1:,-2:]
array([[5, 6]])
```



How about?



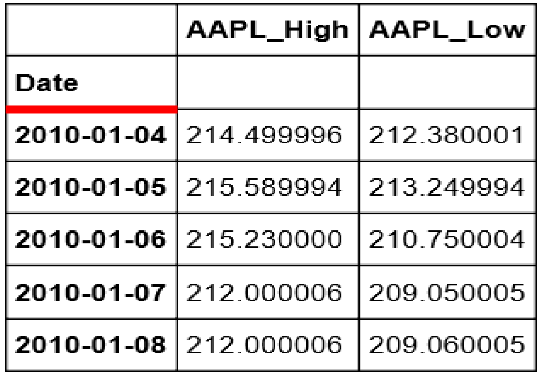
	AAPL_High	AAPL_Low
Date		
2010-01-04	214.499996	212.380001
2010-01-05	215.589994	213.249994
2010-01-06	215.230000	210.750004
2010-01-07	212.000006	209.050005
2010-01-08	212.000006	209.060005

2010-01-06 ~ 2010-01-07
사이에 발생한 data 추출?

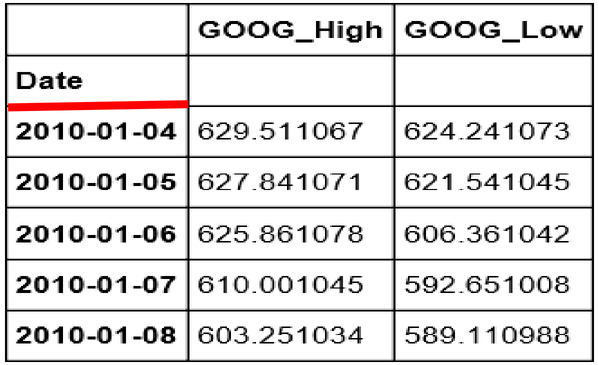
2010년에 월별로 발생한
data를 grouping?

Limitations in NumPy (cont'd)

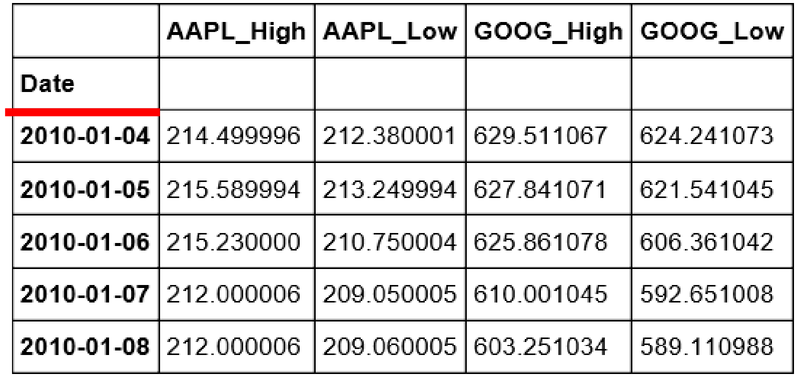
How about?



	AAPL_High	AAPL_Low
Date		
2010-01-04	214.499996	212.380001
2010-01-05	215.589994	213.249994
2010-01-06	215.230000	210.750004
2010-01-07	212.000006	209.050005
2010-01-08	212.000006	209.060005



	GOOG_High	GOOG_Low
Date		
2010-01-04	629.511067	624.241073
2010-01-05	627.841071	621.541045
2010-01-06	625.861078	606.361042
2010-01-07	610.001045	592.651008
2010-01-08	603.251034	589.110988



	AAPL_High	AAPL_Low	GOOG_High	GOOG_Low
Date				
2010-01-04	214.499996	212.380001	629.511067	624.241073
2010-01-05	215.589994	213.249994	627.841071	621.541045
2010-01-06	215.230000	210.750004	625.861078	606.361042
2010-01-07	212.000006	209.050005	610.001045	592.651008
2010-01-08	212.000006	209.060005	603.251034	589.110988

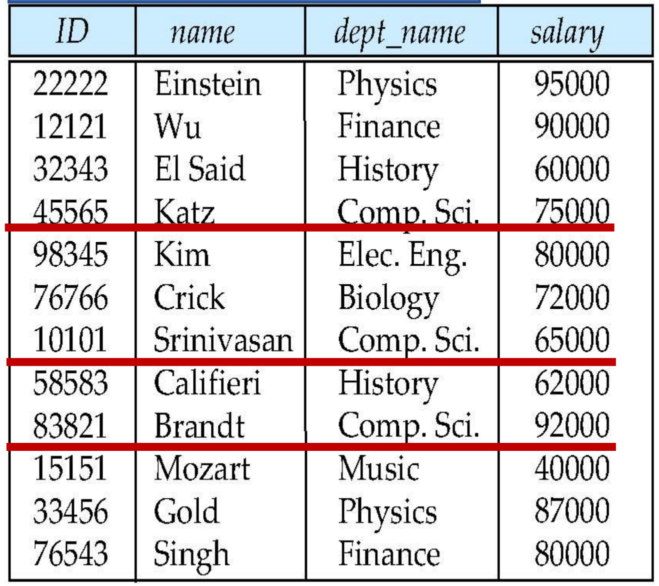
두 테이블의 join?

SQL and Tables (1)

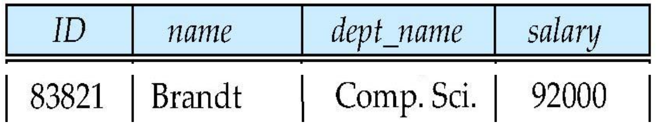
- Find all instructors in Comp. Sci. dept. with salary > 80000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 80000;
```

Instructor relation



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



ID	*name*	*dept_name*	*salary*
83821	Brandt	Comp. Sci.	92000

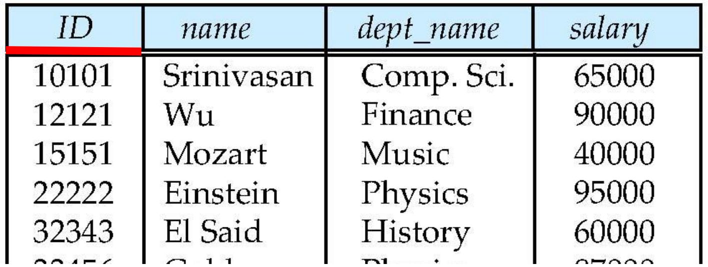
SQL and Tables (2)

- For all instructors who have taught courses, find their names and the course ID of the courses they taught

select *name*, *course_id*
from *instructor*, *teaches*
where *instructor.ID* = *teaches.ID;*

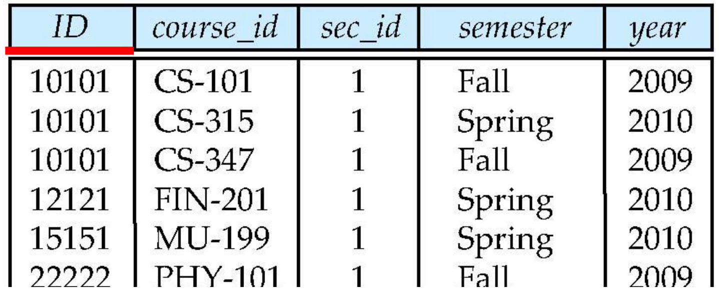
select *
from *instructor* **natural join** *teaches;*

instructor

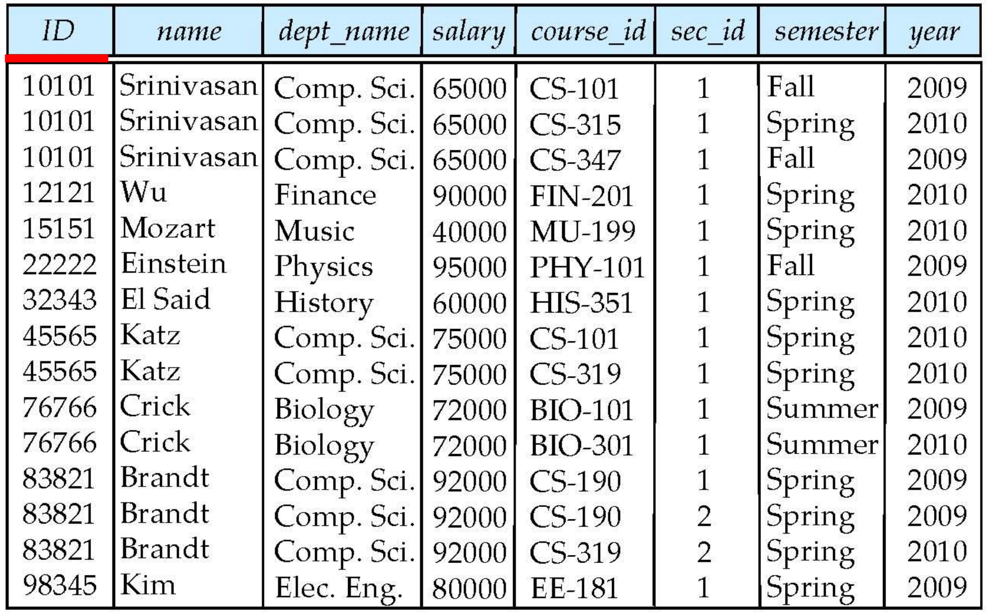


ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches



ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009



ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

instructor $\bowtie$ *teaches*

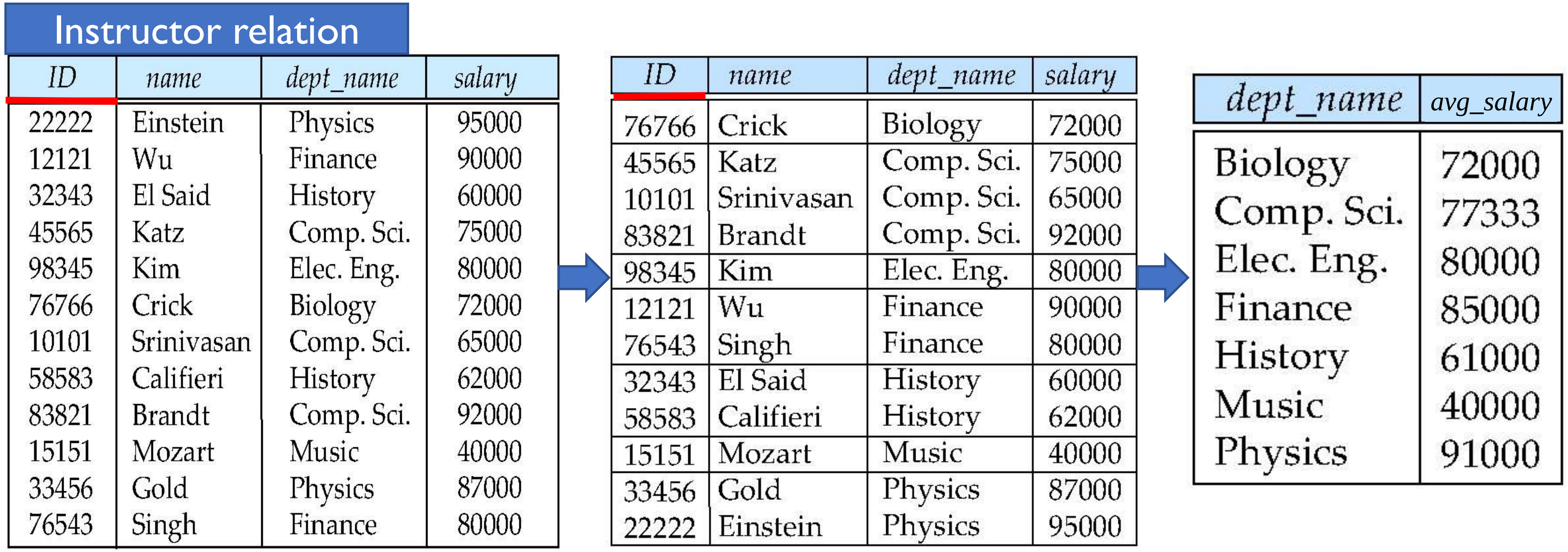
SQL and Tables (3)

- Group instructors in each department

```
select *
from instructor
group by dept_name;
```

- Find the average salary of instructors in each department

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```



Instructor relation

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

What is "Pandas" Module?

- panel data analysis or Python data analysis
- For building and manipulating "relational" or "tabular" data both easy and intuitive
- Built on top of NumPy (2005)
- Open source
  - Original author: Wes McKinney
  - Now part of the PyData project focused on improving Python data libraries
  - http://pandas.pydata.org

- `>>> import panda as pd`

Pandas History

- Developer Wes McKinney started working on Pandas in 2008 while at AQR Capital Management (global investment management firm)
- Need for a high performance, flexible analysis tool for quantitative analysis on financial data
- Before leaving AQR, he was able to convince management to allow him to open source the library
- Another AQR employee, Chang She, joined the effort in 2012 as the second major contributor to the library
- In 2015, Pandas signed on as a sponsored project of NumFOCUS, a non-profit charity in United States

Pandas Module

- Primary data structures
  - Series (1-dimensional)
  - DataFrame (2-dimensional) -- similar to *data.frame* in R
  - Panel (3-dimensional or more)
- Things that pandas does well
  - Easy handling of missing data
  - Size mutability: columns can be inserted and deleted (Add & drop columns)
  - Powerful, flexible group by functionality: Groupby & aggregation
  - Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
  - Intuitive merging and joining data sets: Join (merge) two data
  - Robust I/O tools for loading data from CSV & Excel files, database, and web sources

Series

Series

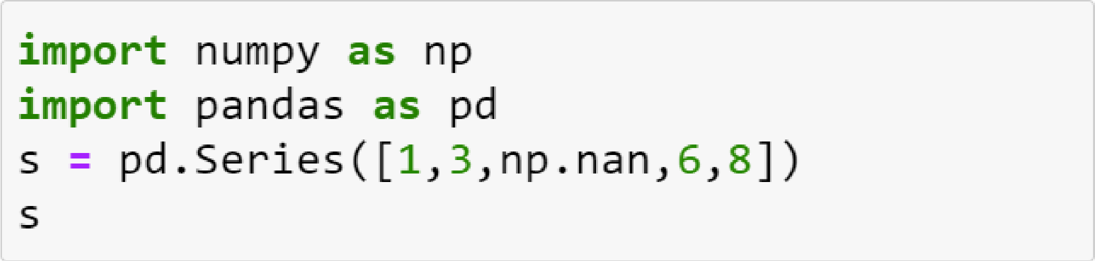
- A one-dimensional array-like object containing a sequence of values and an associated array of data labels, called its index

Index (Integer)	Data
0	1.0
1	2.1
2	1.5
3	4.7
4	3.2
5	1.9

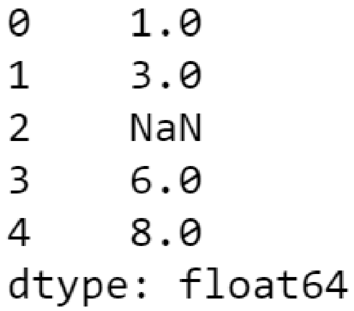
Index (Label)	Data
'Sun'	0
'Mon'	8
'Tue'	9
'Wed'	8
'Thu'	10
'Fri'	6
'Sat'	4

Creating Series (1)

- From a Python list

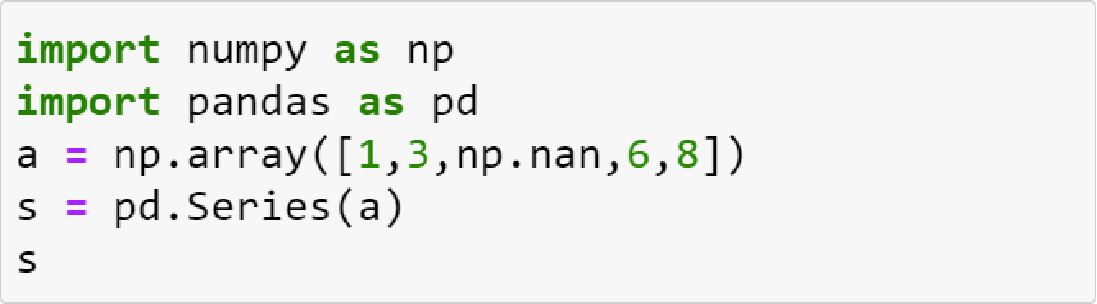


```
import numpy as np
import pandas as pd
s = pd.Series([1,3,np.nan,6,8])
s
```

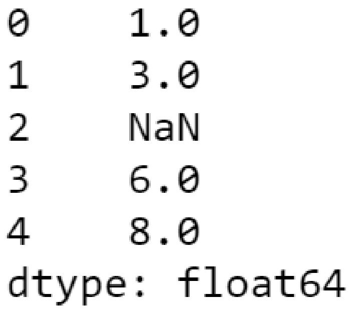


```
0    1.0
1    3.0
2    NaN
3    6.0
4    8.0
dtype: float64
```

- From a NumPy ndarray



```
import numpy as np
import pandas as pd
a = np.array([1,3,np.nan,6,8])
s = pd.Series(a)
s
```

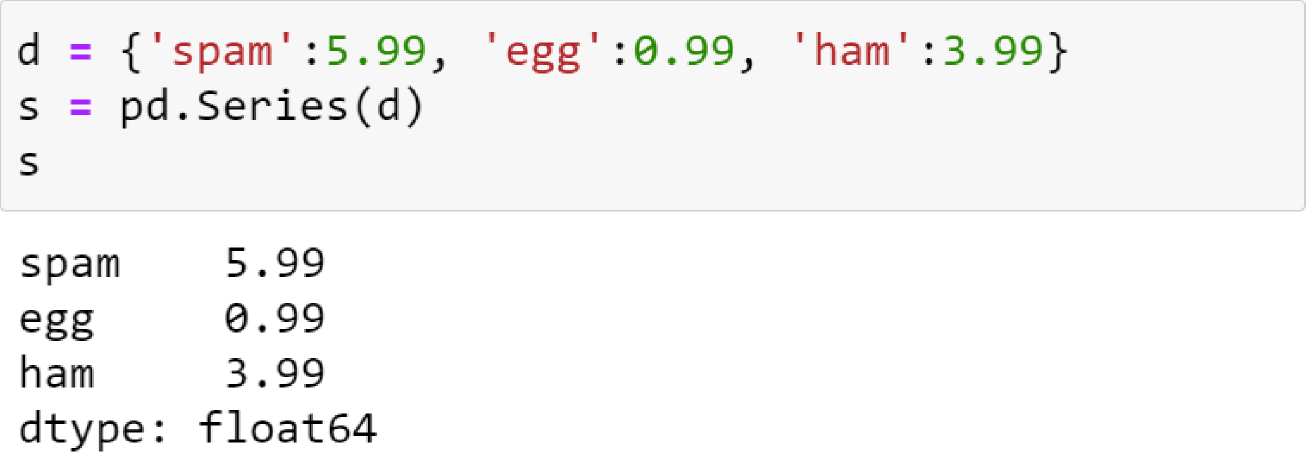


```
0    1.0
1    3.0
2    NaN
3    6.0
4    8.0
dtype: float64
```

automatic indexing
(record id/key)

Creating Series (2)

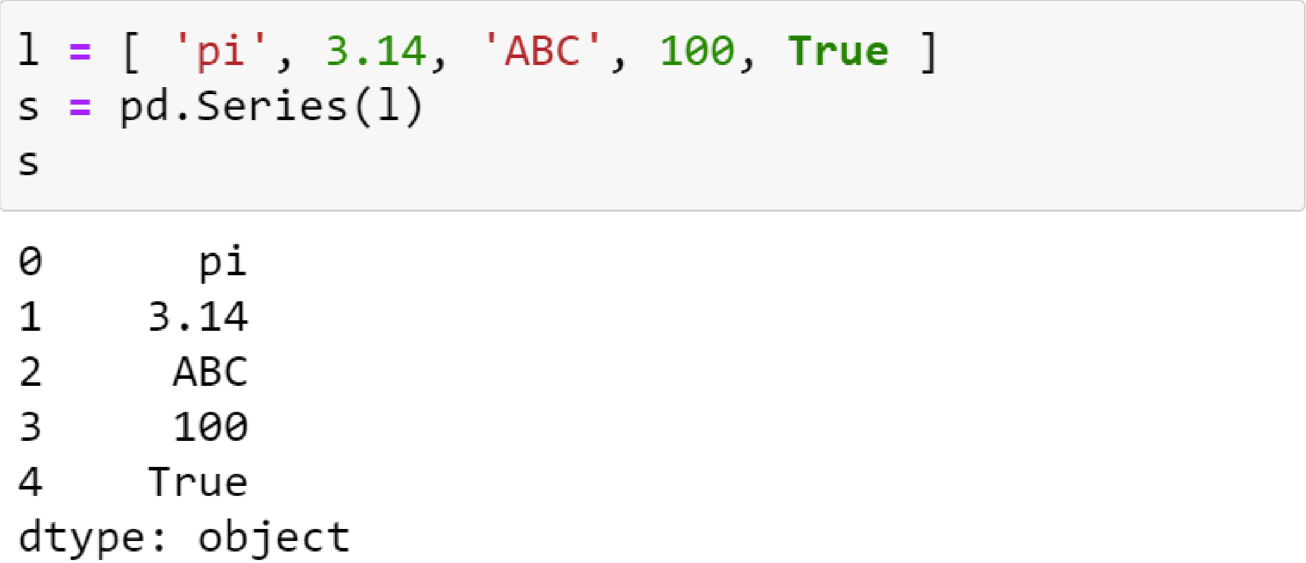
- From a Python dictionary



```
d = {'spam':5.99, 'egg':0.99, 'ham':3.99}
s = pd.Series(d)
s
```

```
spam    5.99
egg     0.99
ham     3.99
dtype: float64
```

- Each element doesn't have to be the same type

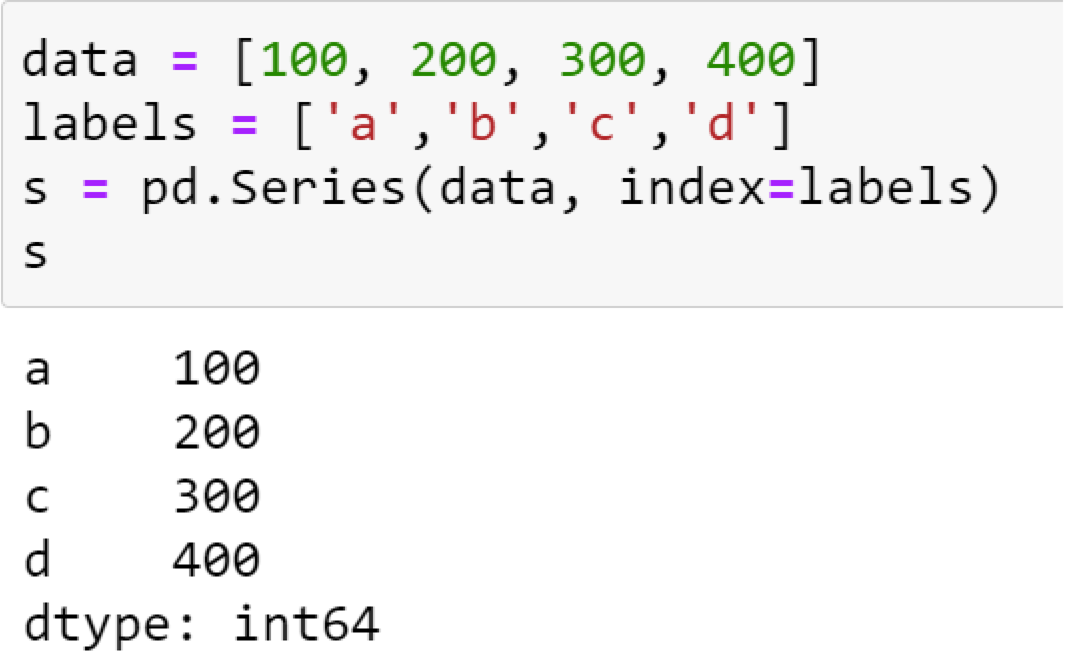


```
l = [ 'pi', 3.14, 'ABC', 100, True ]
s = pd.Series(l)
s
```

```
0      pi
1    3.14
2     ABC
3     100
4    True
dtype: object
```

Index Labels

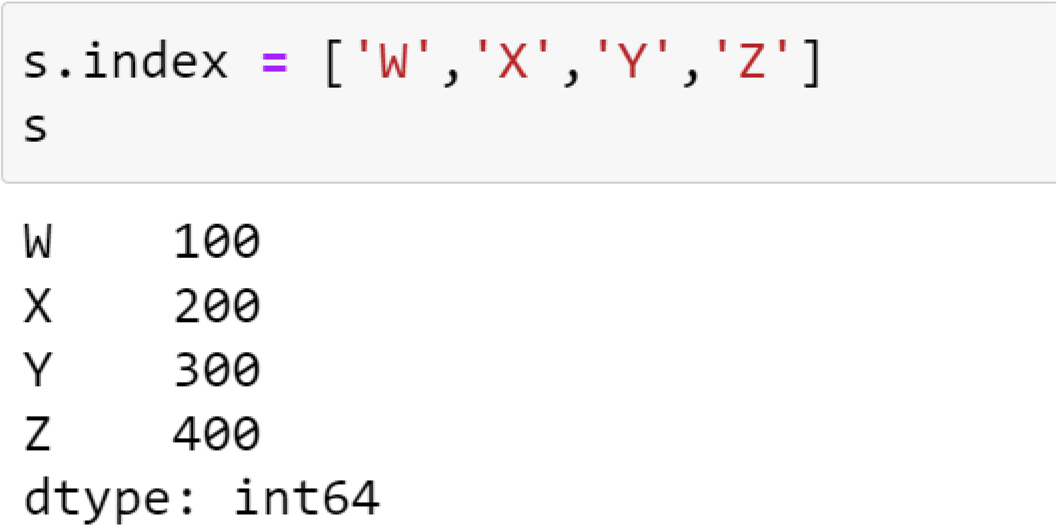
- Index labels can be specified when Series is created



```
data = [100, 200, 300, 400]
labels = ['a','b','c','d']
s = pd.Series(data, index=labels)
s
```

```
a     100
b     200
c     300
d     400
dtype: int64
```

- Index labels can be changed in-place

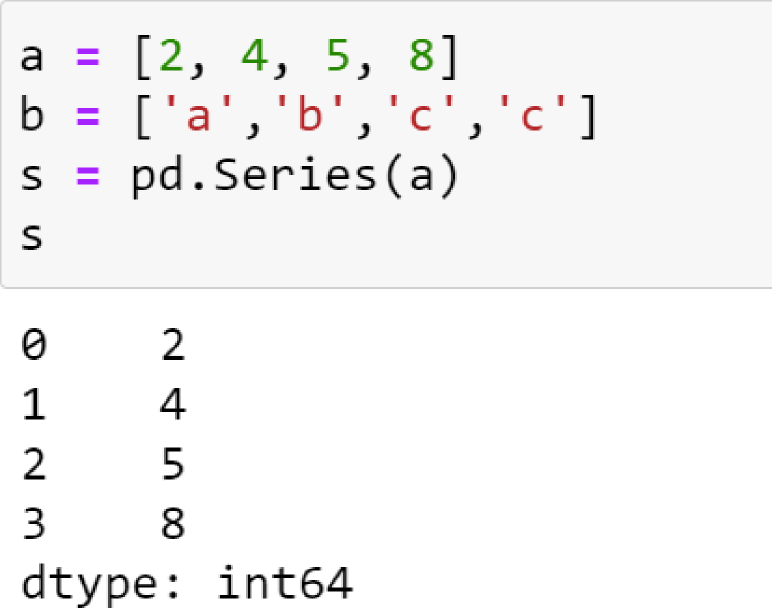


```
s.index = ['W','X','Y','Z']
s
```

```
W     100
X     200
Y     300
Z     400
dtype: int64
```

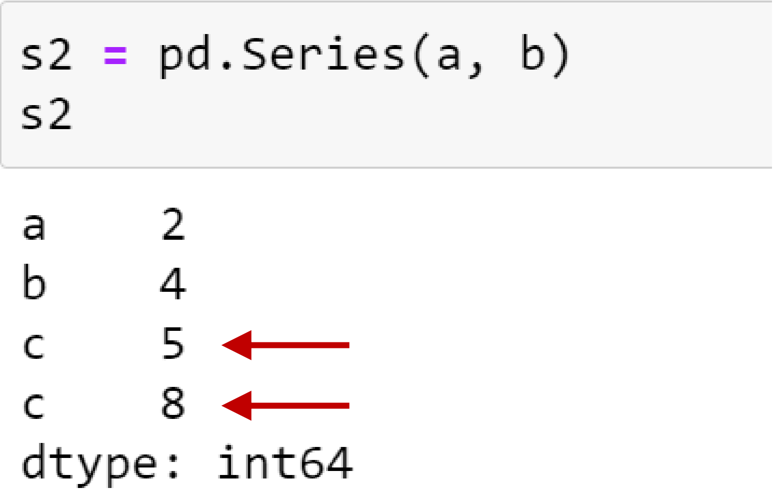
pandas.Series()

- *pd*.`Series(`[*data*], [*index*], [*dtype*], … `)`
  - One-dimensional ndarray with axis labels (including time series)
  - *data*: contains data stored in Series
  - *index*: values must be hashable and have the same length as *data* (default: `np.arange(len(`*data*`))`)
  - Non-unique index values are allowed



```
a = [2, 4, 5, 8]
b = ['a','b','c','c']
s = pd.Series(a)
s
```

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

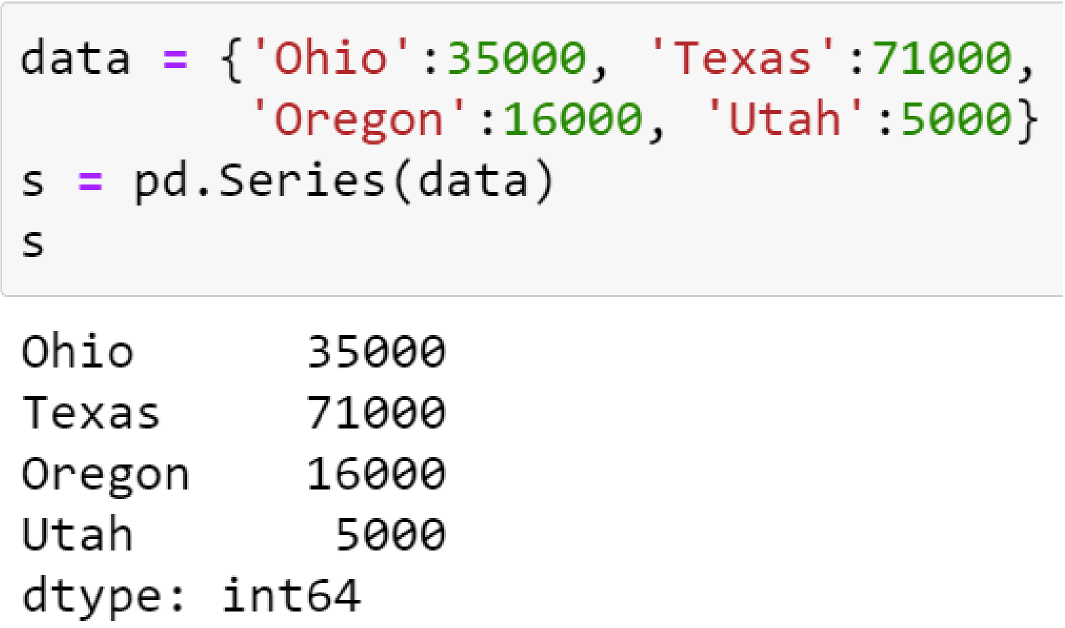


```
s2 = pd.Series(a, b)
s2
```

```
a    2
b    4
c    5 ←
c    8 ←
dtype: int64
```

Handling Missing Entries

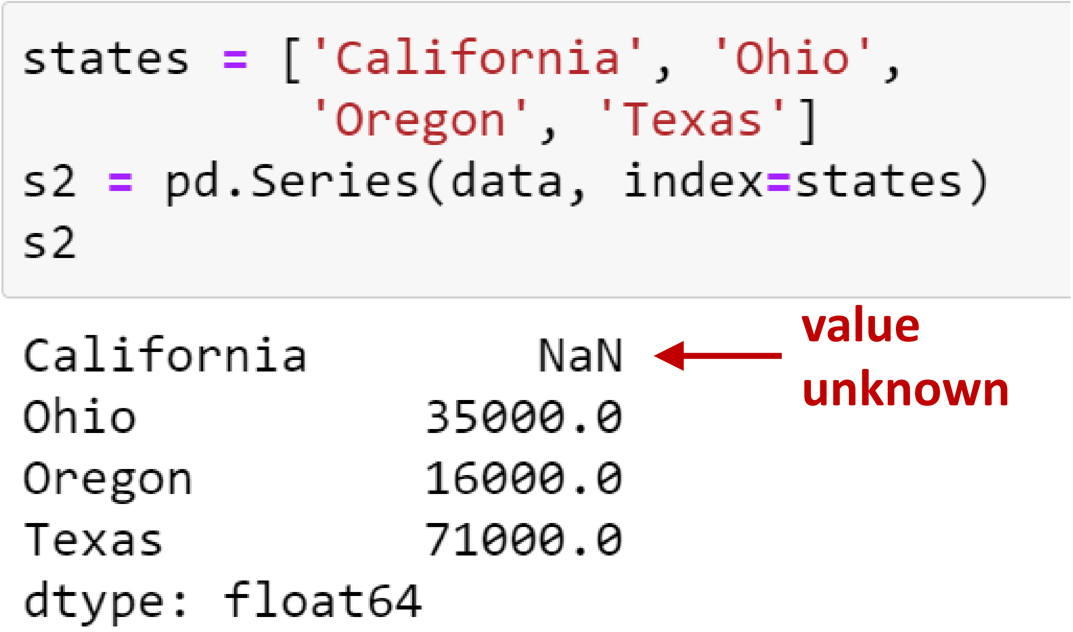
- Series creation from dictionary



```
data = {'Ohio':35000, 'Texas':71000,
        'Oregon':16000, 'Utah':5000}
s = pd.Series(data)
s
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

- Extracting data from another dictionary



```
states = ['California', 'Ohio',
          'Oregon', 'Texas']
s2 = pd.Series(data, index=states)
s2
```

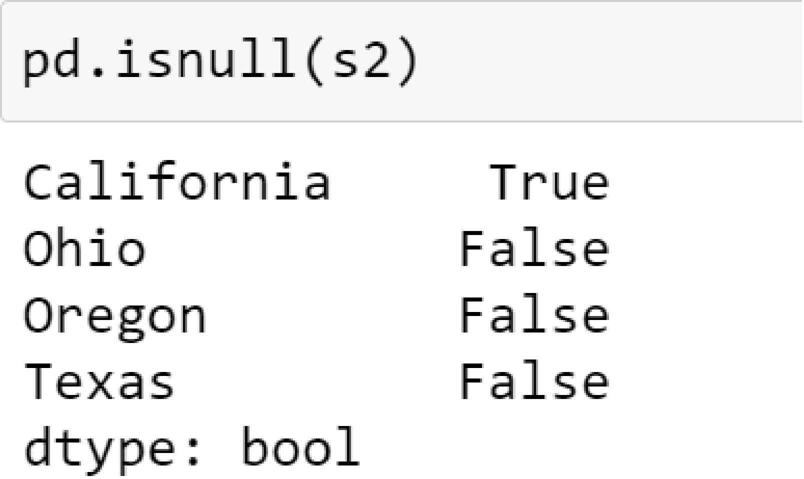
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

value unknown

Checking Null Values

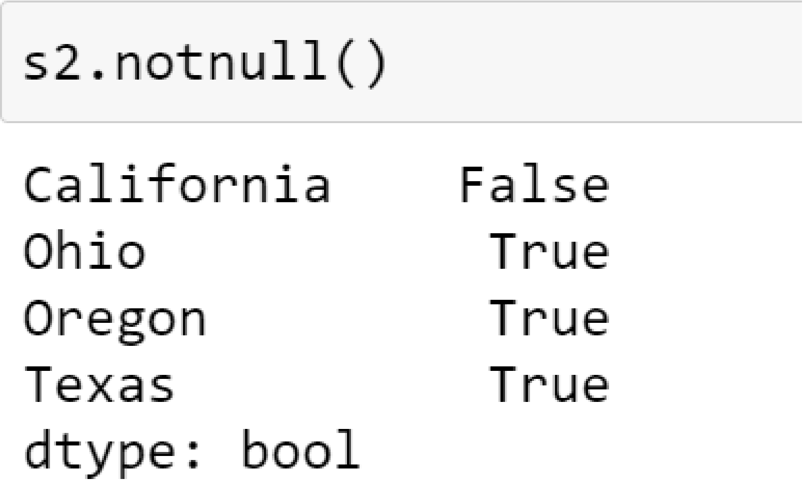
- *pd*.`isnull(`*obj*`)`
- *pd*.`isna(`*obj*`)`
  - Return an array of Boolean indicating whether the corresponding element is missing
  - Same as *obj*.`isnull()`

- *pd*.`notnull(`*obj*`)`
  - Detect non-missing values
  - Same as *obj*.`notnull()`



```
pd.isnull(s2)
```

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

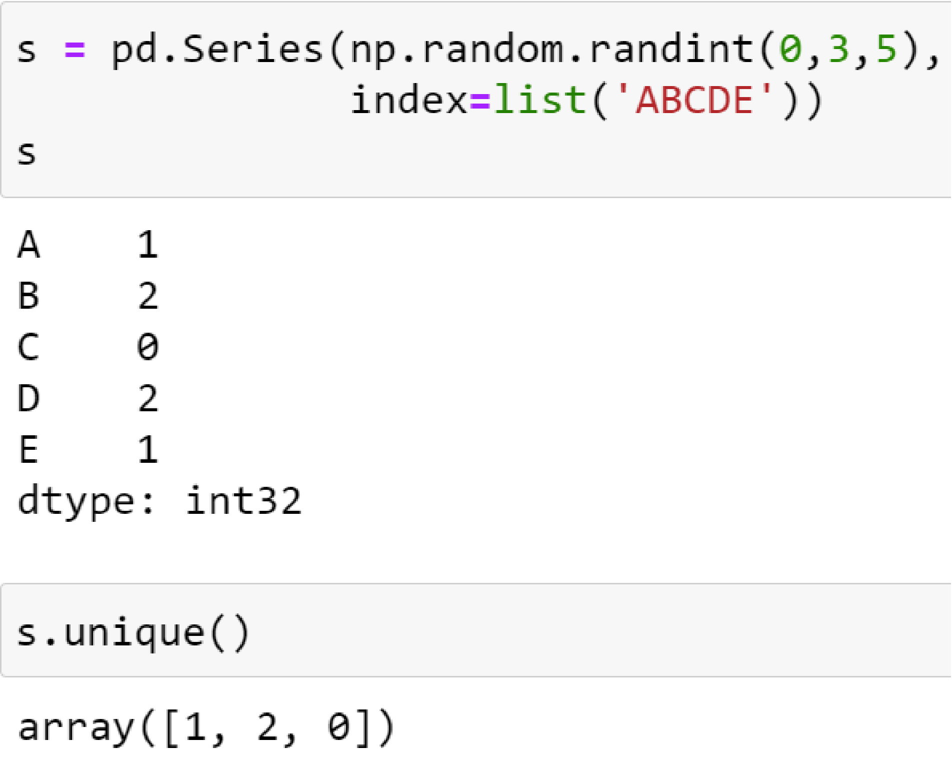


```
s2.notnull()
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Unique Values and Value Counts

- *series.*`unique()`
  - Return unique values of Series object



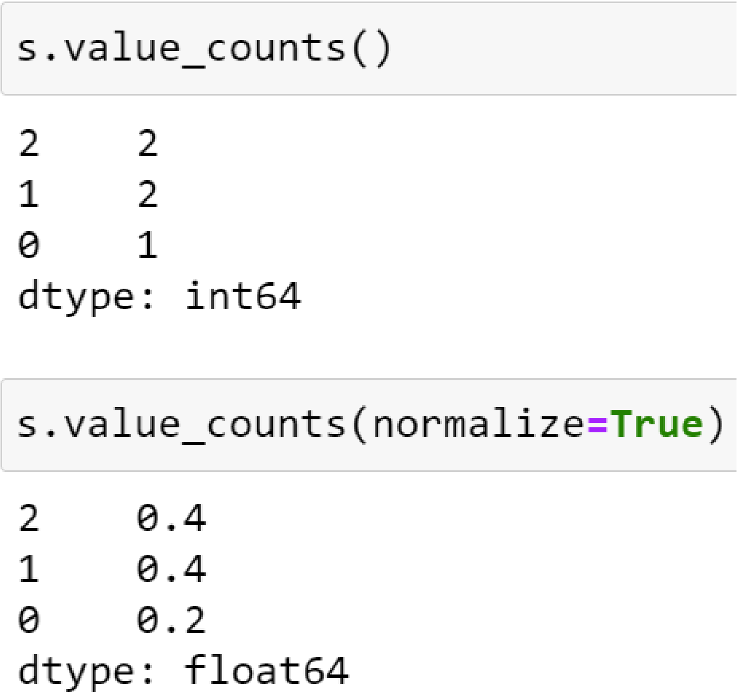
```
s = pd.Series(np.random.randint(0,3,5),
              index=list('ABCDE'))
s
```

```
A    1
B    2
C    0
D    2
E    1
dtype: int32
```

```
s.unique()
```

```
array([1, 2, 0])
```

- *series.*`value_counts(`*normalize=False, sort=True, ascending=False, …*`)`
  - Return a Series containing counts of unique values



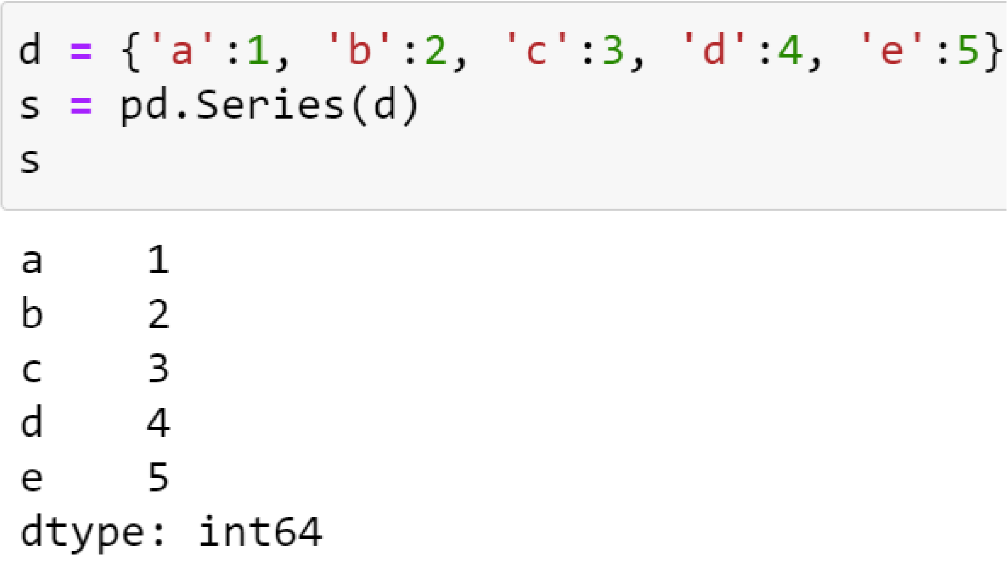
```
s.value_counts()
```

```
2    2
1    2
0    1
dtype: int64
```

```
s.value_counts(normalize=True)
```

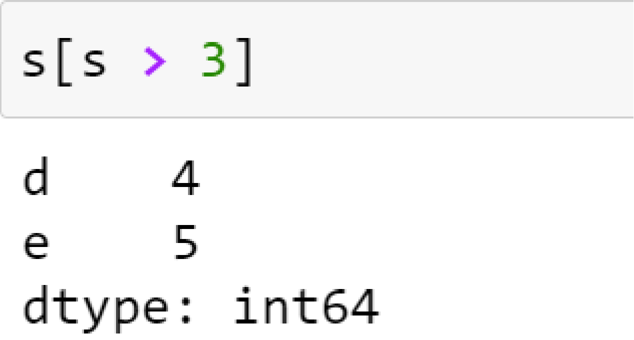
```
2    0.4
1    0.4
0    0.2
dtype: float64
```

Selecting Elements



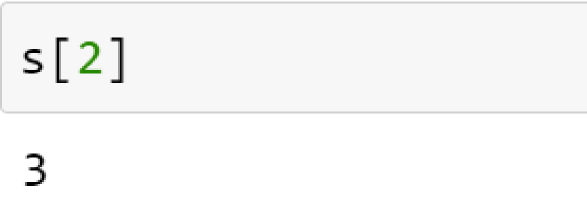
```
d = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}
s = pd.Series(d)
s
```

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



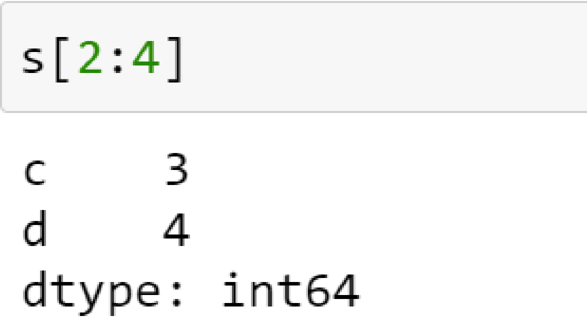
```
s[s > 3]
```

```
d    4
e    5
dtype: int64
```



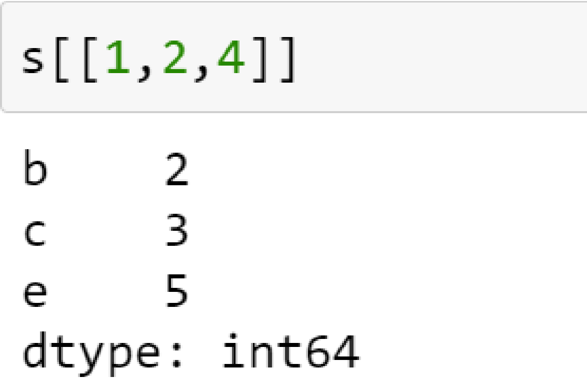
```
s[2]
```

```
3
```



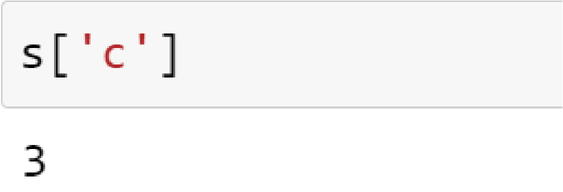
```
s[2:4]
```

```
c    3
d    4
dtype: int64
```



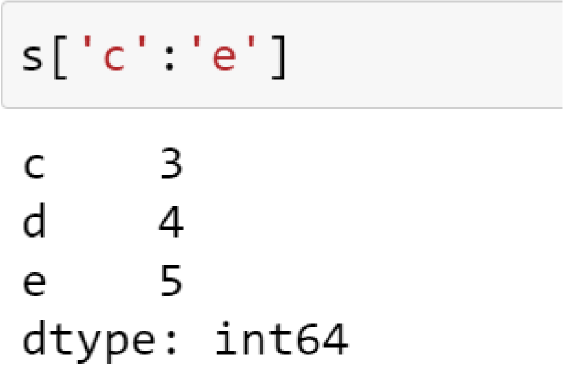
```
s[[1,2,4]]
```

```
b    2
c    3
e    5
dtype: int64
```



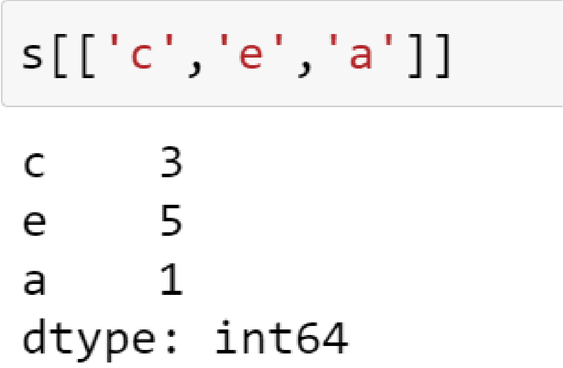
```
s['c']
```

```
3
```



```
s['c':'e']
```

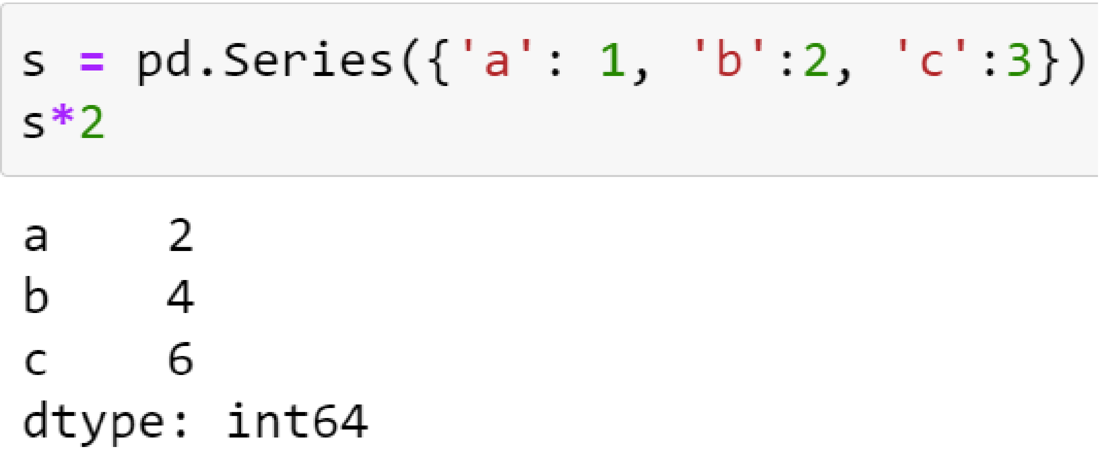
```
c    3
d    4
e    5
dtype: int64
```



```
s[['c','e','a']]
```

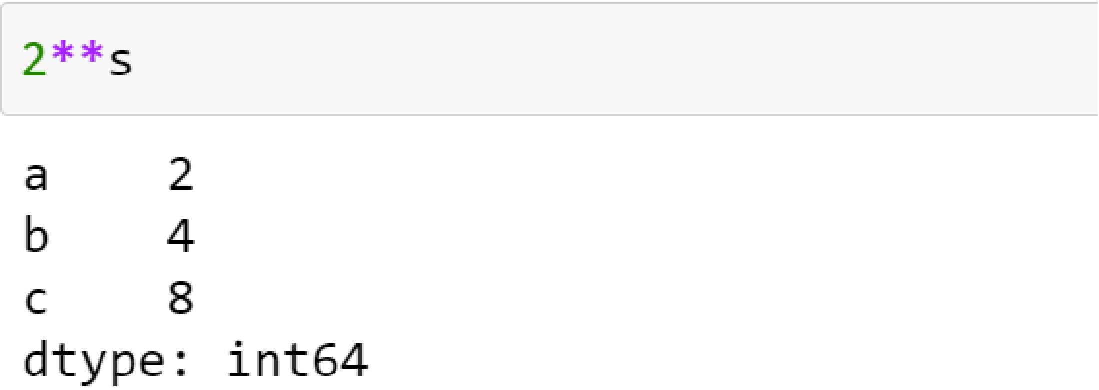
```
c    3
e    5
a    1
dtype: int64
```

Manipulating Series



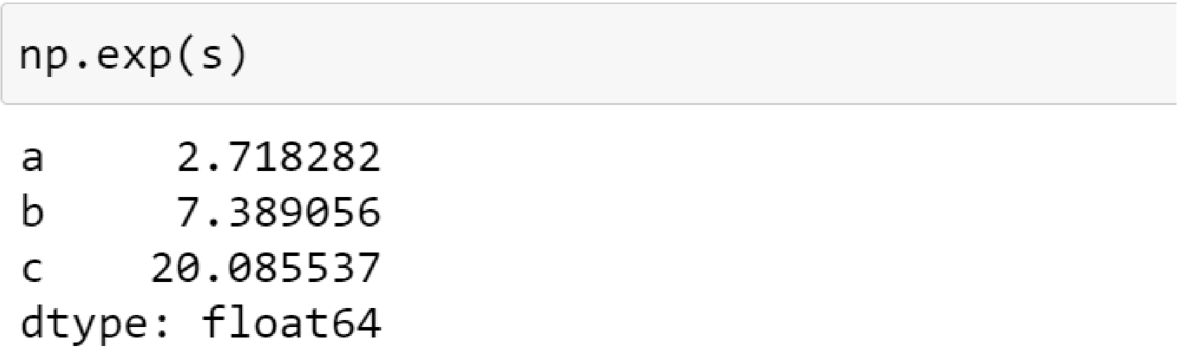
```
s = pd.Series({'a': 1, 'b':2, 'c':3})
s*2
```

```
a    2
b    4
c    6
dtype: int64
```



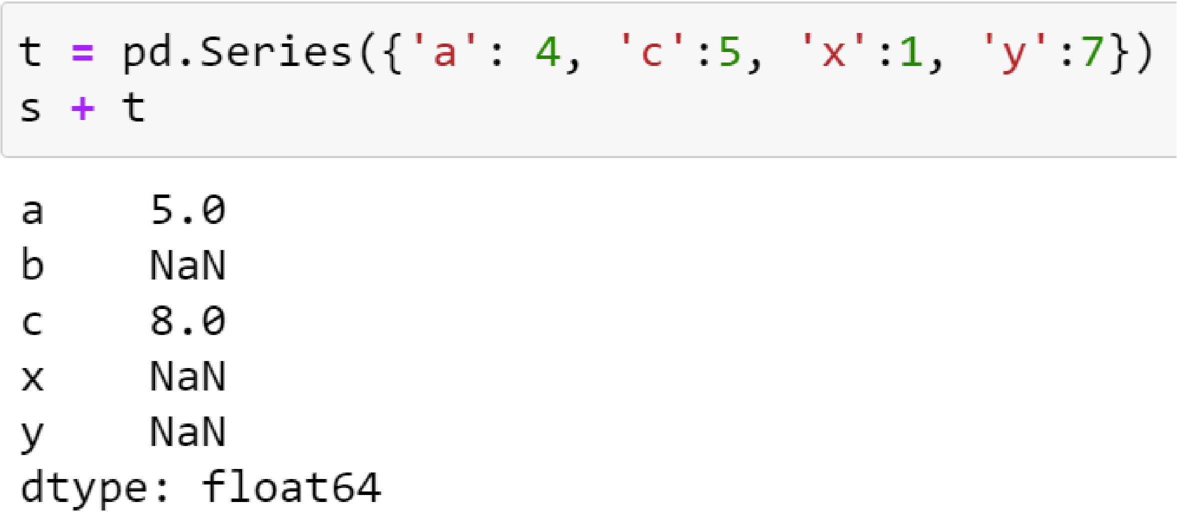
```
2**s
```

```
a    2
b    4
c    8
dtype: int64
```



```
np.exp(s)
```

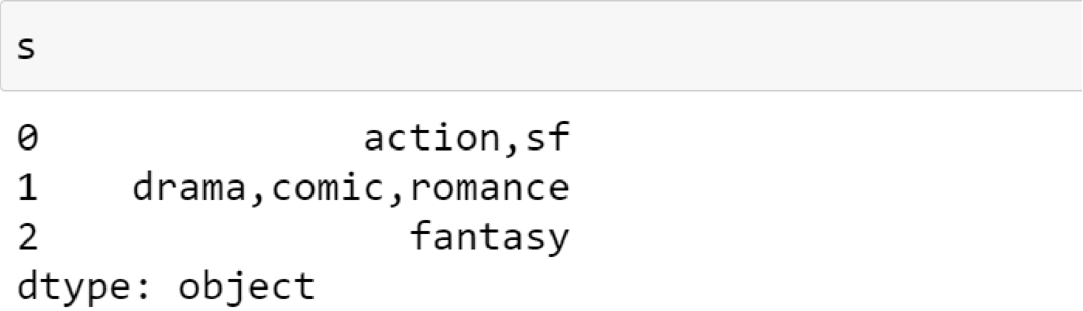
```
a     2.718282
b     7.389056
c    20.085537
dtype: float64
```



```
t = pd.Series({'a': 4, 'c':5, 'x':1, 'y':7})
s + t
```

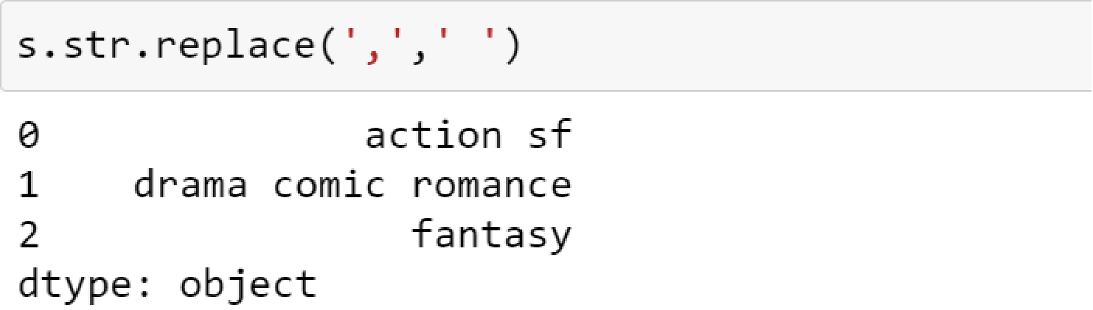
```
a    5.0
b    NaN
c    8.0
x    NaN
y    NaN
dtype: float64
```

String Functions and to_list()



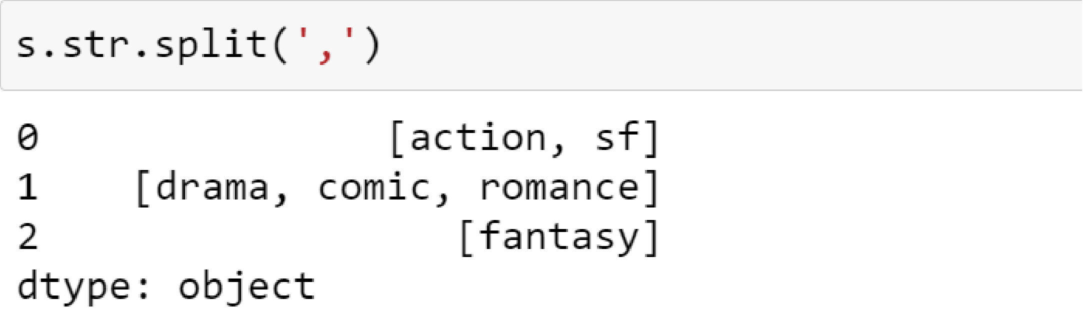
```
s
```

```
0                action,sf
1    drama,comic,romance
2                  fantasy
dtype: object
```



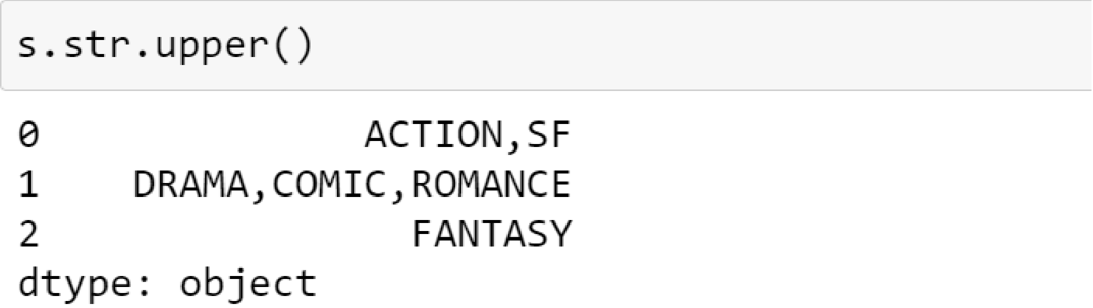
```
s.str.replace(',',' ')
```

```
0                action sf
1    drama comic romance
2                  fantasy
dtype: object
```



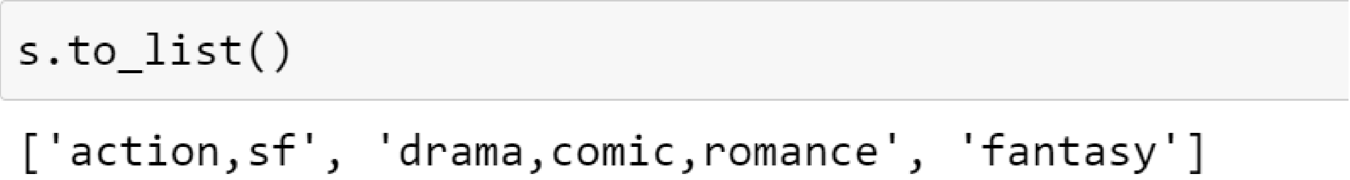
```
s.str.split(',')
```

```
0                [action, sf]
1    [drama, comic, romance]
2                   [fantasy]
dtype: object
```



```
s.str.upper()
```

```
0                ACTION,SF
1    DRAMA,COMIC,ROMANCE
2                  FANTASY
dtype: object
```



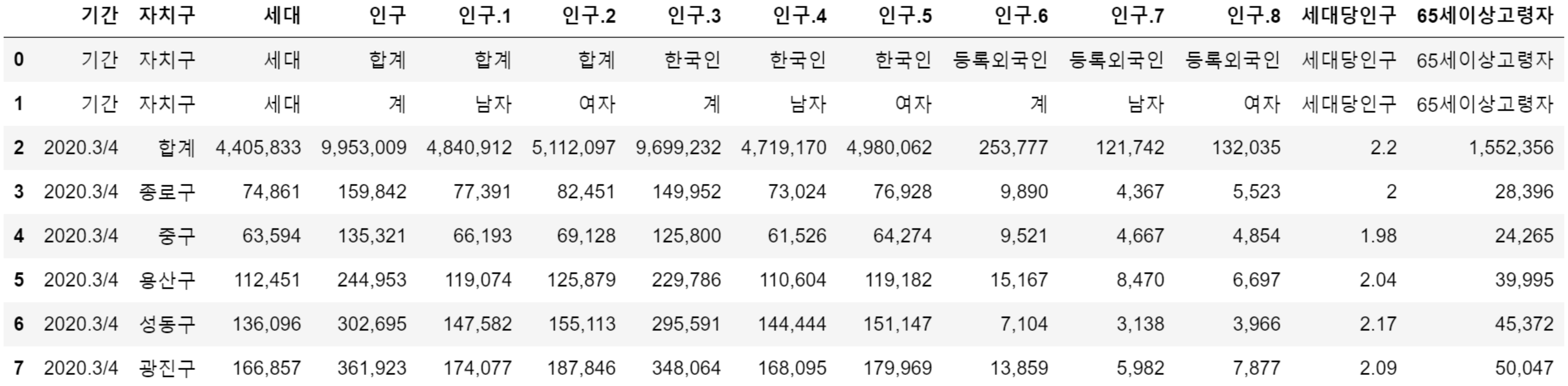
```
s.to_list()
```

```
['action,sf', 'drama,comic,romance', 'fantasy']
```

Creating DataFrame

DataFrame

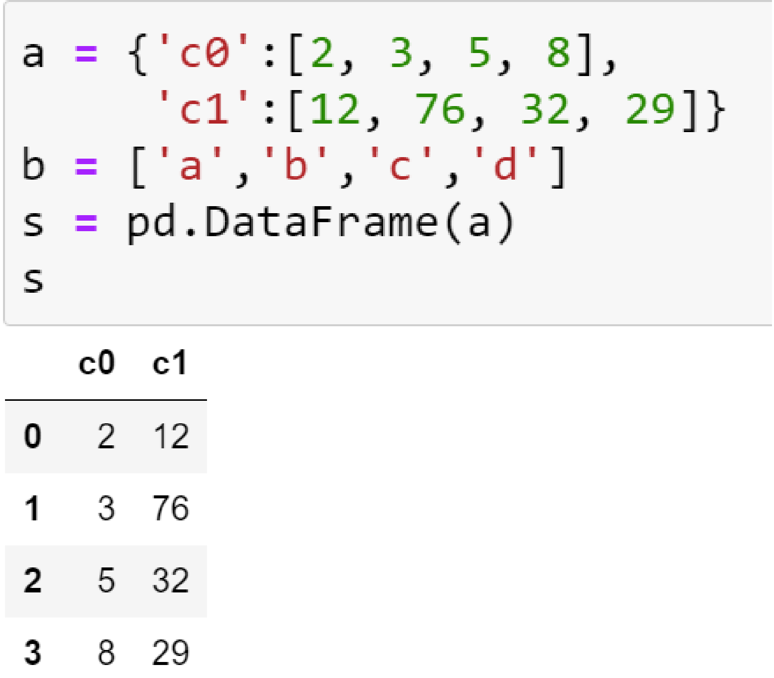
- A 2-D array of indexed data
  - Similar to a spreadsheet or SQL table
- DataFrame is the most commonly used pandas object



	기간	자치구	세대	인구	인구.1	인구.2	인구.3	인구.4	인구.5	인구.6	인구.7	인구.8	세대당인구	65세이상고령자
0	기간	자치구	세대	합계	합계	합계	한국인	한국인	한국인	등록외국인	등록외국인	등록외국인	세대당인구	65세이상고령자
1	기간	자치구	세대	계	남자	여자	계	남자	여자	계	남자	여자	세대당인구	65세이상고령자
2	2020.3/4	합계	4,405,833	9,953,009	4,840,912	5,112,097	9,699,232	4,719,170	4,980,062	253,777	121,742	132,035	2.2	1,552,356
3	2020.3/4	종로구	74,861	159,842	77,391	82,451	149,952	73,024	76,928	9,890	4,367	5,523	2	28,396
4	2020.3/4	중구	63,594	135,321	66,193	69,128	125,800	61,526	64,274	9,521	4,667	4,854	1.98	24,265
5	2020.3/4	용산구	112,451	244,953	119,074	125,879	229,786	110,604	119,182	15,167	8,470	6,697	2.04	39,995
6	2020.3/4	성동구	136,096	302,695	147,582	155,113	295,591	144,444	151,147	7,104	3,138	3,966	2.17	45,372
7	2020.3/4	광진구	166,857	361,923	174,077	187,846	348,064	168,095	179,969	13,859	5,982	7,877	2.09	50,047

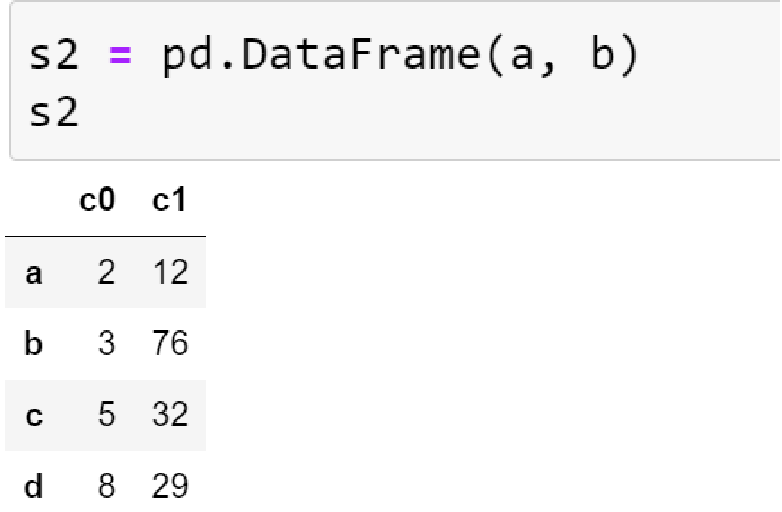
pandas.DataFrame()

- *pd*.DataFrame([*data*], [*index*], [*columns*], [*dtype*], ...)
  - The primary pandas data structure
  - Two-dimensional size-mutable, potentially heterogenous tabular data structure with labeled axes (rows and columns)
  - *data*: ndarray, list, dictionary, or dataframe
  - *index*: index to use for resulting frame. (default: np.arange(len(*data*))
  - *columns*: column labels to use for resulting frame



```
a = {'c0':[2, 3, 5, 8],
     'c1':[12, 76, 32, 29]}
b = ['a','b','c','d']
s = pd.DataFrame(a)
s
```

	c0	c1
0	2	12
1	3	76
2	5	32
3	8	29

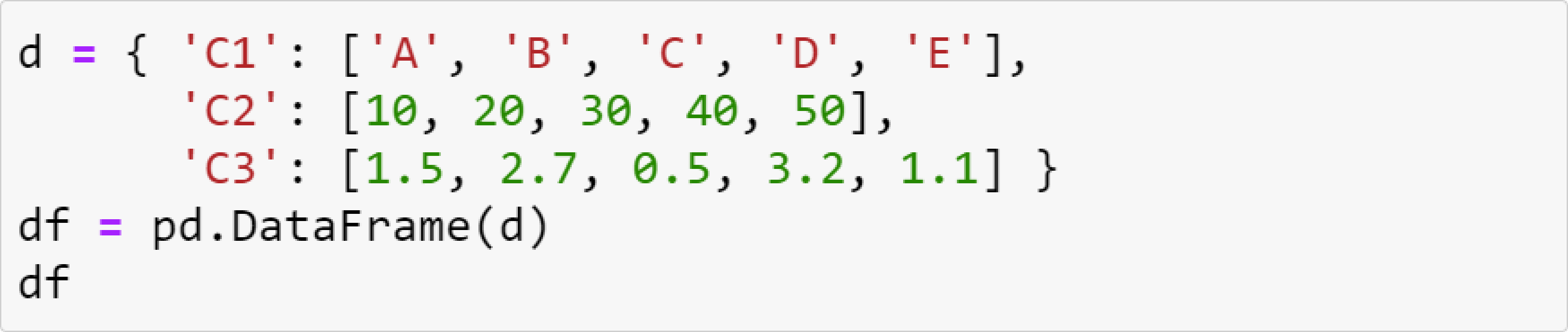


```
s2 = pd.DataFrame(a, b)
s2
```

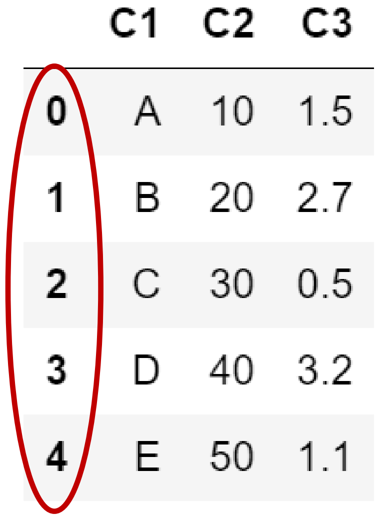
	c0	c1
a	2	12
b	3	76
c	5	32
d	8	29

Dict → DataFrame

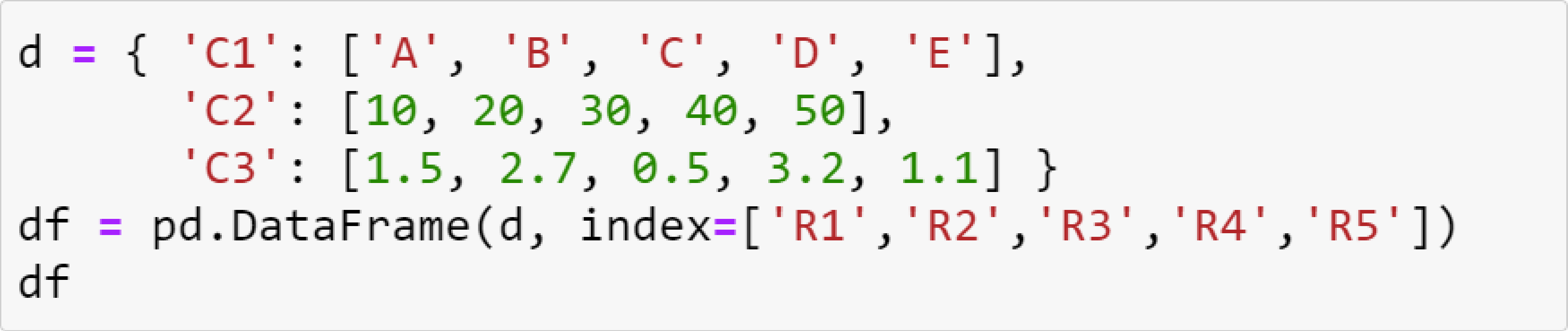
- From a Python dictionary of equal-length lists



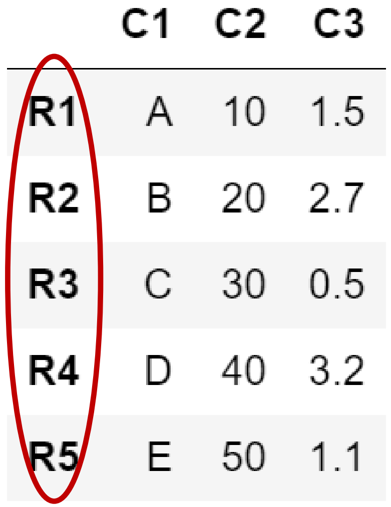
```
d = { 'C1': ['A', 'B', 'C', 'D', 'E'],
      'C2': [10, 20, 30, 40, 50],
      'C3': [1.5, 2.7, 0.5, 3.2, 1.1] }
df = pd.DataFrame(d)
df
```



	C1	C2	C3
0	A	10	1.5
1	B	20	2.7
2	C	30	0.5
3	D	40	3.2
4	E	50	1.1



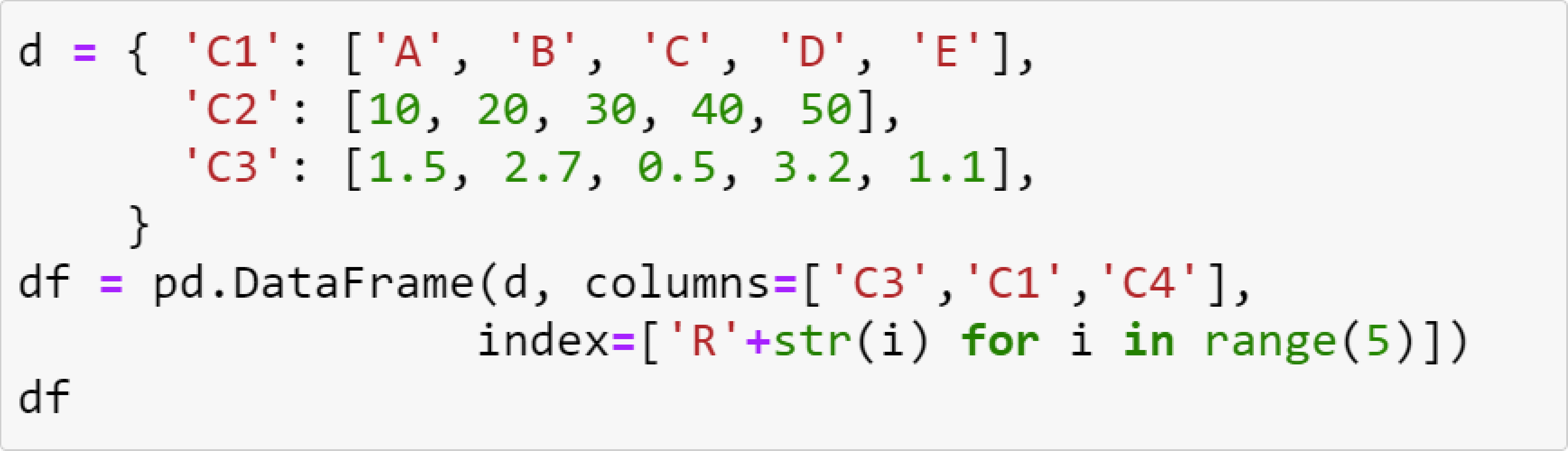
```
d = { 'C1': ['A', 'B', 'C', 'D', 'E'],
      'C2': [10, 20, 30, 40, 50],
      'C3': [1.5, 2.7, 0.5, 3.2, 1.1] }
df = pd.DataFrame(d, index=['R1','R2','R3','R4','R5'])
df
```



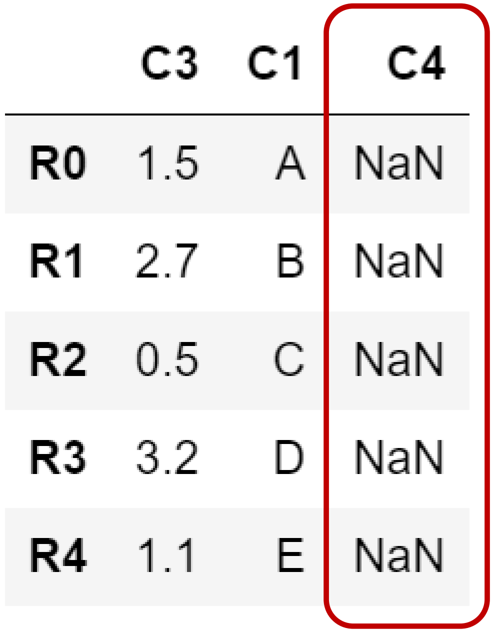
	C1	C2	C3
R1	A	10	1.5
R2	B	20	2.7
R3	C	30	0.5
R4	D	40	3.2
R5	E	50	1.1

Arranging Columns

- Part of columns can be selected
- Order of columns can be changed
- New columns can be added
  - Missing values are set to NaN



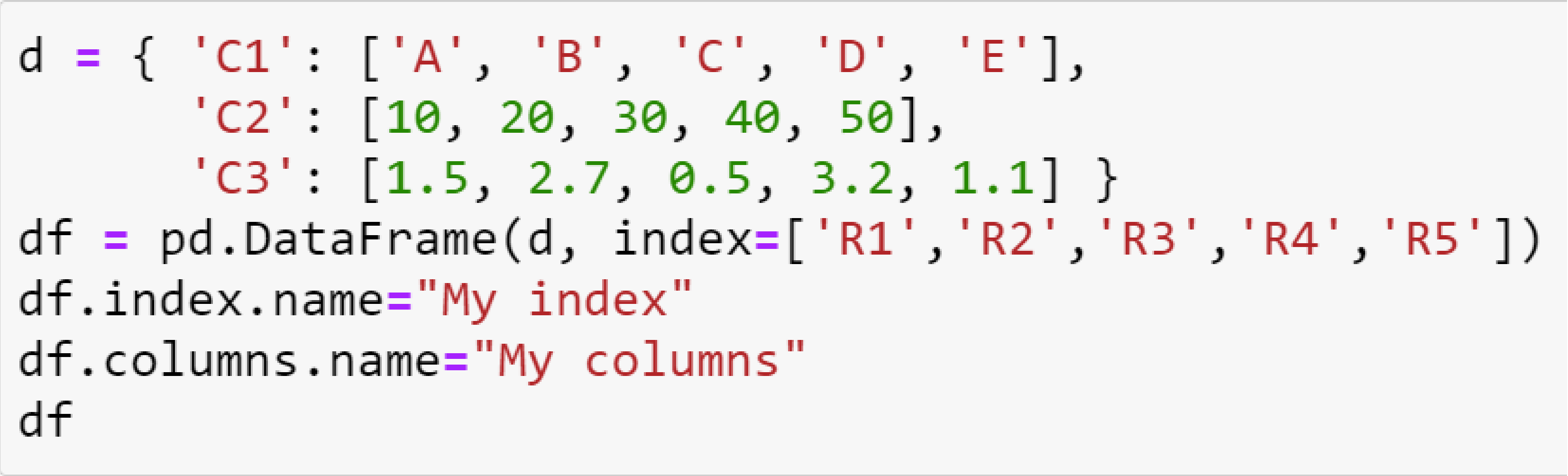
```
d = { 'C1': ['A', 'B', 'C', 'D', 'E'],
      'C2': [10, 20, 30, 40, 50],
      'C3': [1.5, 2.7, 0.5, 3.2, 1.1],
    }
df = pd.DataFrame(d, columns=['C3','C1','C4'],
                     index=['R'+str(i) for i in range(5)])
df
```



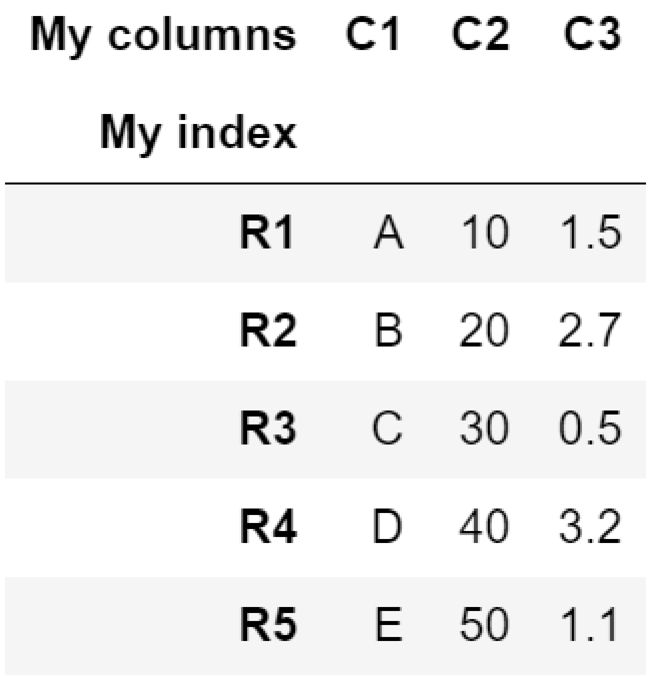
	C3	C1	C4
R0	1.5	A	NaN
R1	2.7	B	NaN
R2	0.5	C	NaN
R3	3.2	D	NaN
R4	1.1	E	NaN

Index and Column Names

- *df*.`index.name` and *df*.`columns.name`



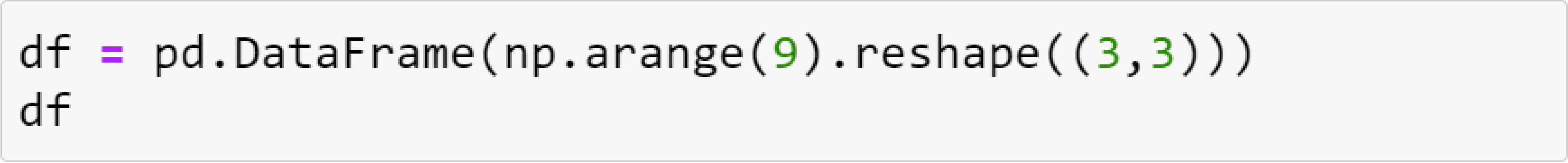
```
d = { 'C1': ['A', 'B', 'C', 'D', 'E'],
      'C2': [10, 20, 30, 40, 50],
      'C3': [1.5, 2.7, 0.5, 3.2, 1.1] }
df = pd.DataFrame(d, index=['R1','R2','R3','R4','R5'])
df.index.name="My index"
df.columns.name="My columns"
df
```



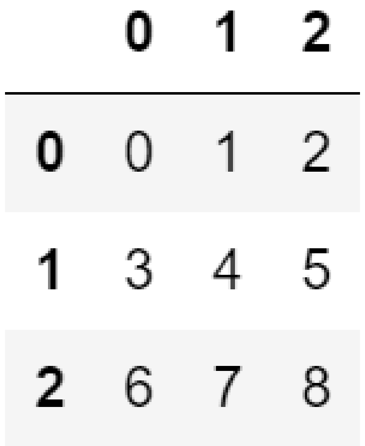
My columns	C1	C2	C3
My index			
R1	A	10	1.5
R2	B	20	2.7
R3	C	30	0.5
R4	D	40	3.2
R5	E	50	1.1

NumPy ndarray → DataFrame

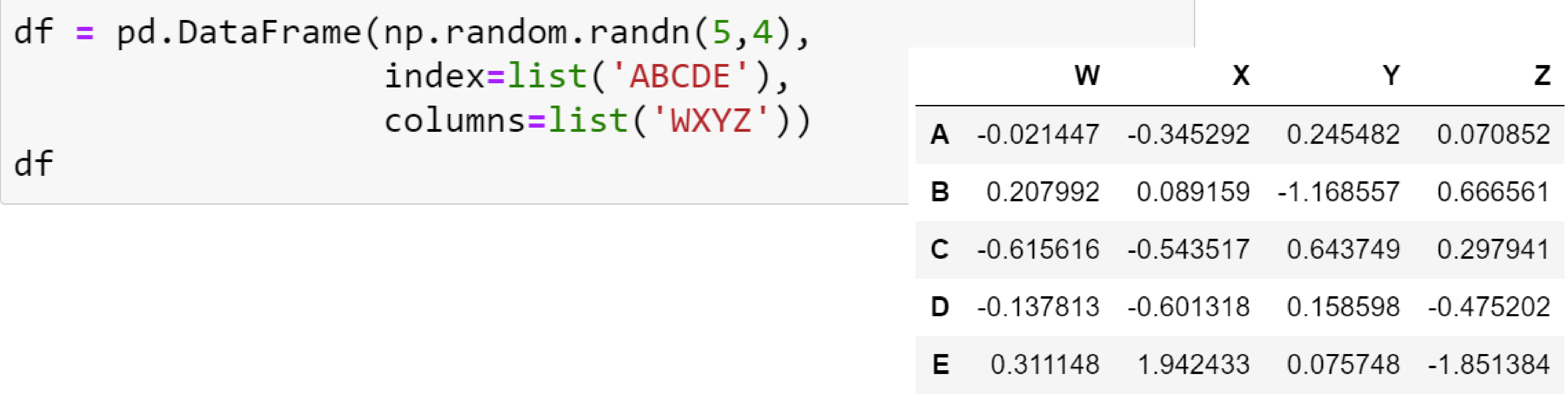
- From a NumPy ndarray



```
df = pd.DataFrame(np.arange(9).reshape((3,3)))
df
```



	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8



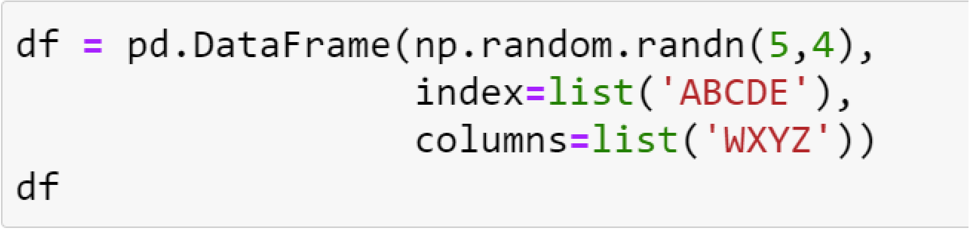
```
df = pd.DataFrame(np.random.randn(5,4),
                  index=list('ABCDE'),
                  columns=list('WXYZ'))

df
```

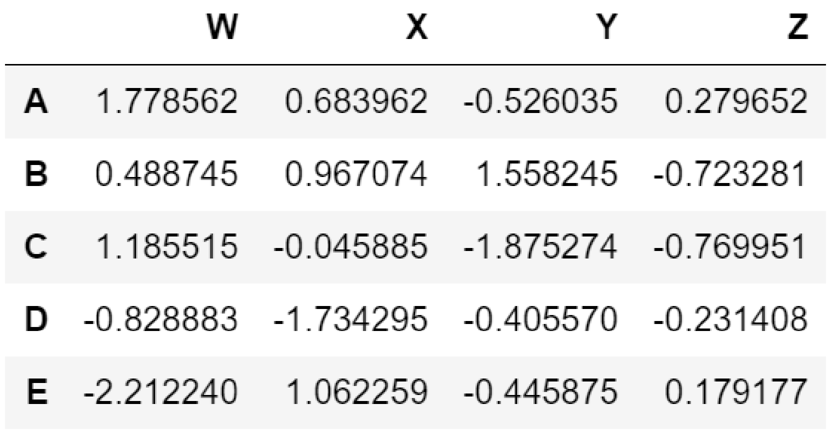
	W	X	Y	Z
A	-0.021447	-0.345292	0.245482	0.070852
B	0.207992	0.089159	-1.168557	0.666561
C	-0.615616	-0.543517	0.643749	0.297941
D	-0.137813	-0.601318	0.158598	-0.475202
E	0.311148	1.942433	0.075748	-1.851384

DataFrame → NumPy ndarray

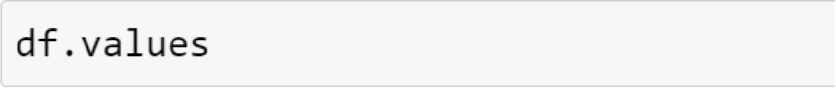
- *df*.values
  - Return a Numpy representation of the DataFrame



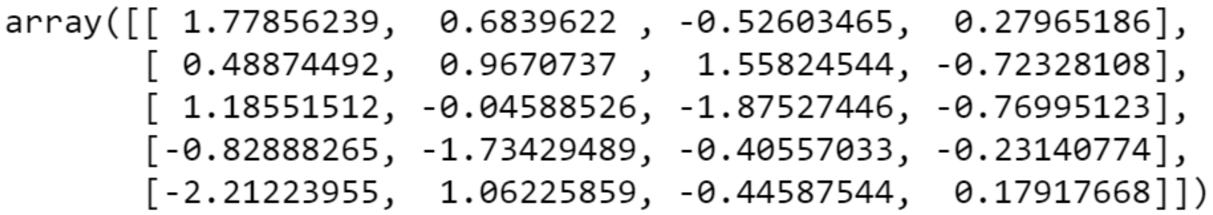
```
df = pd.DataFrame(np.random.randn(5,4),
                  index=list('ABCDE'),
                  columns=list('WXYZ'))
df
```



	W	X	Y	Z
A	1.778562	0.683962	-0.526035	0.279652
B	0.488745	0.967074	1.558245	-0.723281
C	1.185515	-0.045885	-1.875274	-0.769951
D	-0.828883	-1.734295	-0.405570	-0.231408
E	-2.212240	1.062259	-0.445875	0.179177



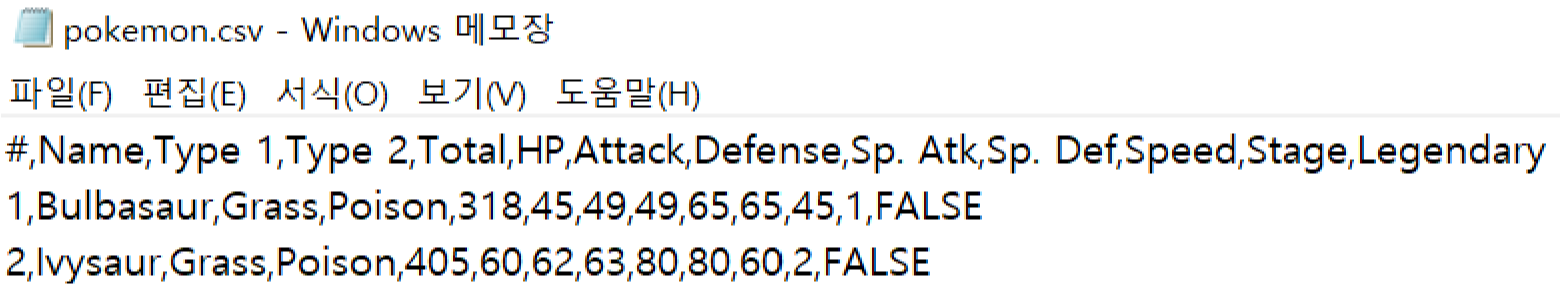
```
df.values
```



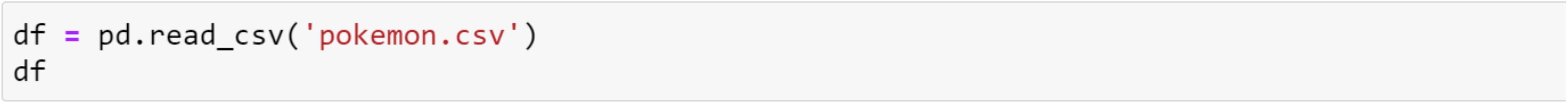
```
array([[ 1.77856239,  0.6839622 , -0.52603465,  0.27965186],
       [ 0.48874492,  0.9670737 ,  1.55824544, -0.72328108],
       [ 1.18551512, -0.04588526, -1.87527446, -0.76995123],
       [-0.82888265, -1.73429489, -0.40557033, -0.23140774],
       [-2.21223955,  1.06225859, -0.44587544,  0.17917668]])
```

CSV File → DataFrame

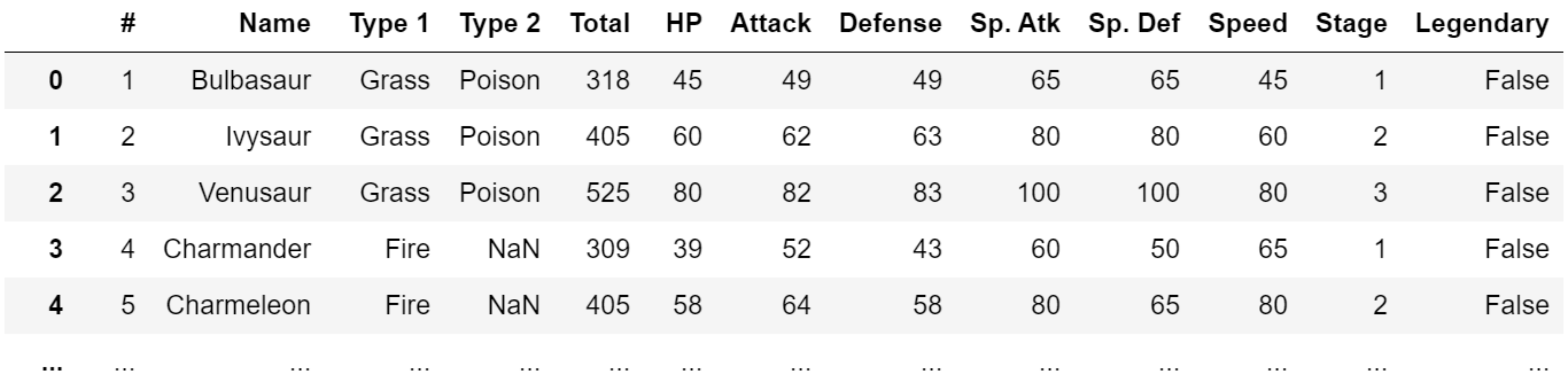
- From a CSV(Comma-Separated Values) file


pokemon.csv - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
#,Name,Type 1,Type 2,Total,HP,Attack,Defense,Sp. Atk,Sp. Def,Speed,Stage,Legendary
1,Bulbasaur,Grass,Poison,318,45,49,49,65,65,45,1,FALSE
2,Ivysaur,Grass,Poison,405,60,62,63,80,80,60,2,FALSE
```



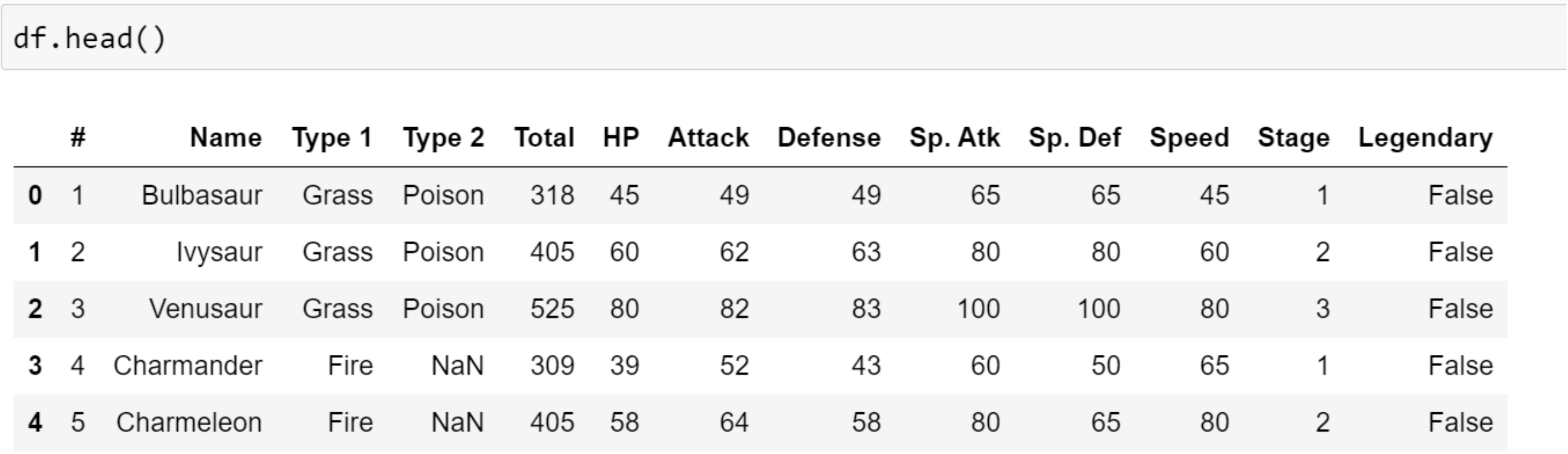
```
df = pd.read_csv('pokemon.csv')
df
```



	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Stage	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	2	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	3	False
3	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False
4	5	Charmeleon	Fire	NaN	405	58	64	58	80	65	80	2	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...

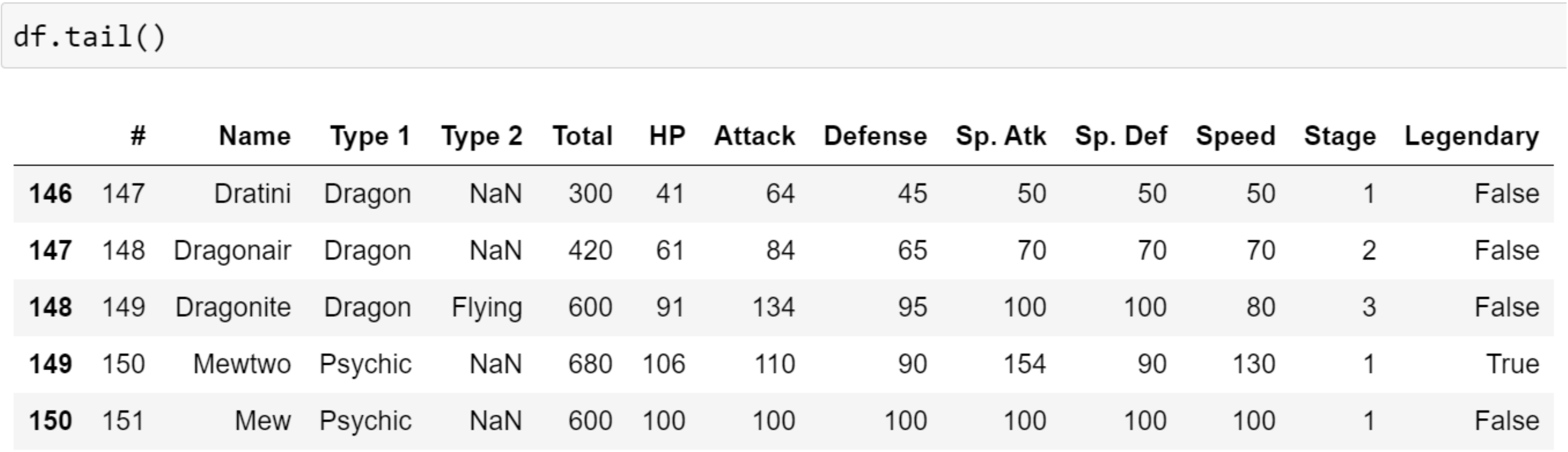
Getting a Glimpse of Your Data

- *df*.`head()` and *df*.`tail()`



```
df.head()
```

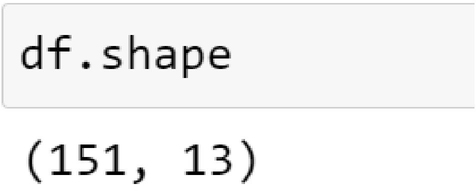
	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Stage	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	2	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	3	False
3	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False
4	5	Charmeleon	Fire	NaN	405	58	64	58	80	65	80	2	False



```
df.tail()
```

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Stage	Legendary
146	147	Dratini	Dragon	NaN	300	41	64	45	50	50	50	1	False
147	148	Dragonair	Dragon	NaN	420	61	84	65	70	70	70	2	False
148	149	Dragonite	Dragon	Flying	600	91	134	95	100	100	80	3	False
149	150	Mewtwo	Psychic	NaN	680	106	110	90	154	90	130	1	True
150	151	Mew	Psychic	NaN	600	100	100	100	100	100	100	1	False

- *df*.`shape`

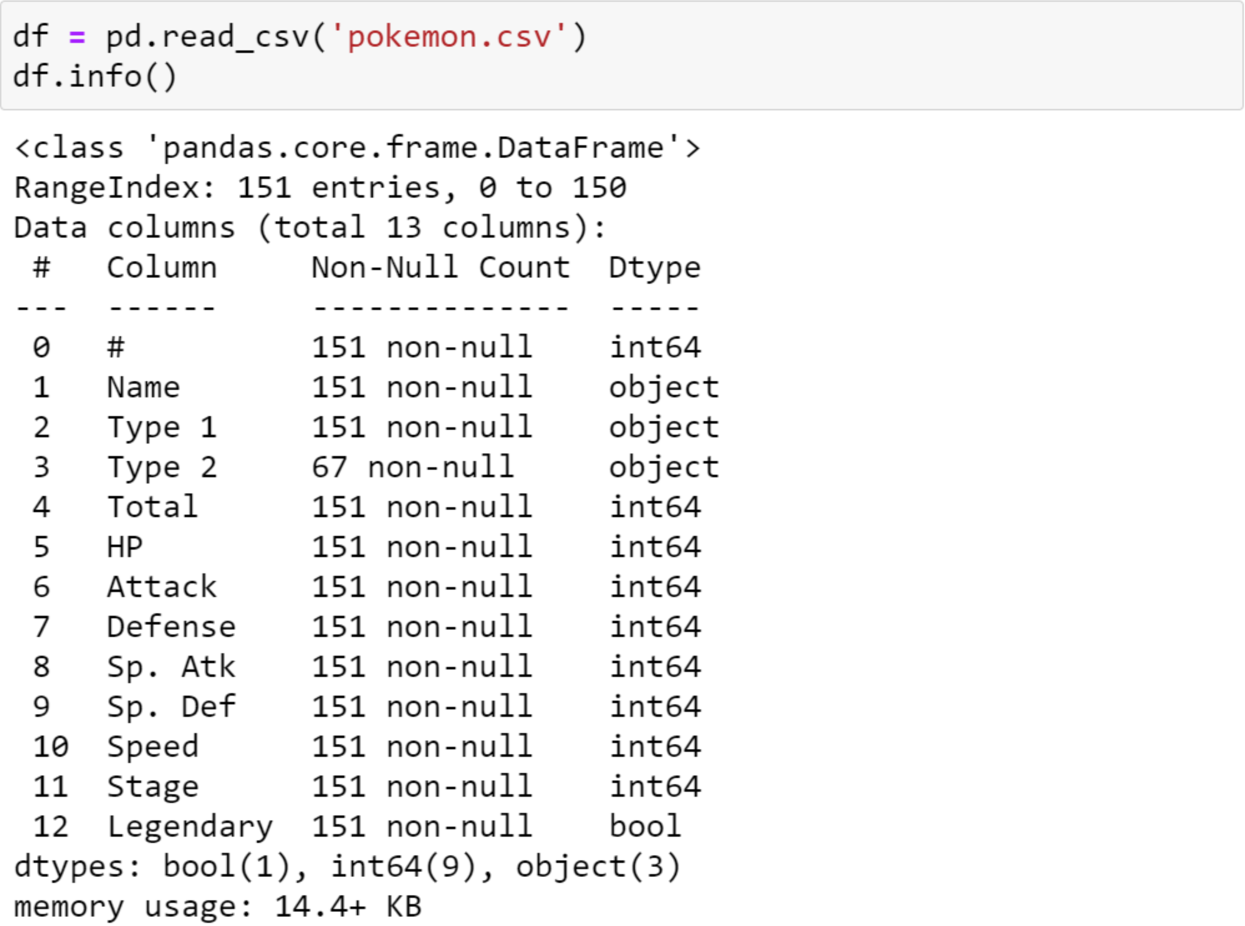


```
df.shape
```

```
(151, 13)
```

DataFrame Info.

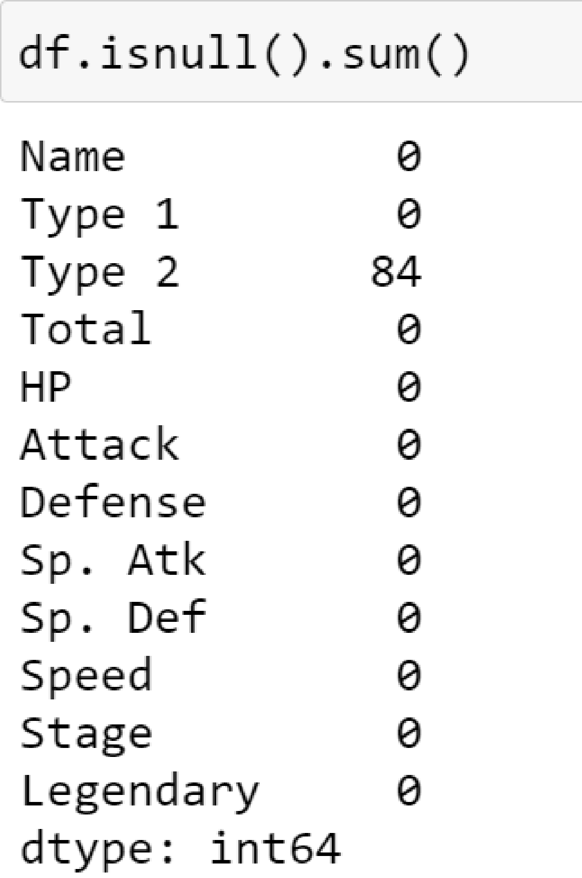
- *df*.info()



```
df = pd.read_csv('pokemon.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 151 entries, 0 to 150
Data columns (total 13 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   #          151 non-null    int64
 1   Name       151 non-null    object
 2   Type 1     151 non-null    object
 3   Type 2     67 non-null     object
 4   Total      151 non-null    int64
 5   HP         151 non-null    int64
 6   Attack     151 non-null    int64
 7   Defense    151 non-null    int64
 8   Sp. Atk    151 non-null    int64
 9   Sp. Def    151 non-null    int64
 10  Speed      151 non-null    int64
 11  Stage      151 non-null    int64
 12  Legendary  151 non-null    bool
dtypes: bool(1), int64(9), object(3)
memory usage: 14.4+ KB
```

- *df*.isnull().sum()

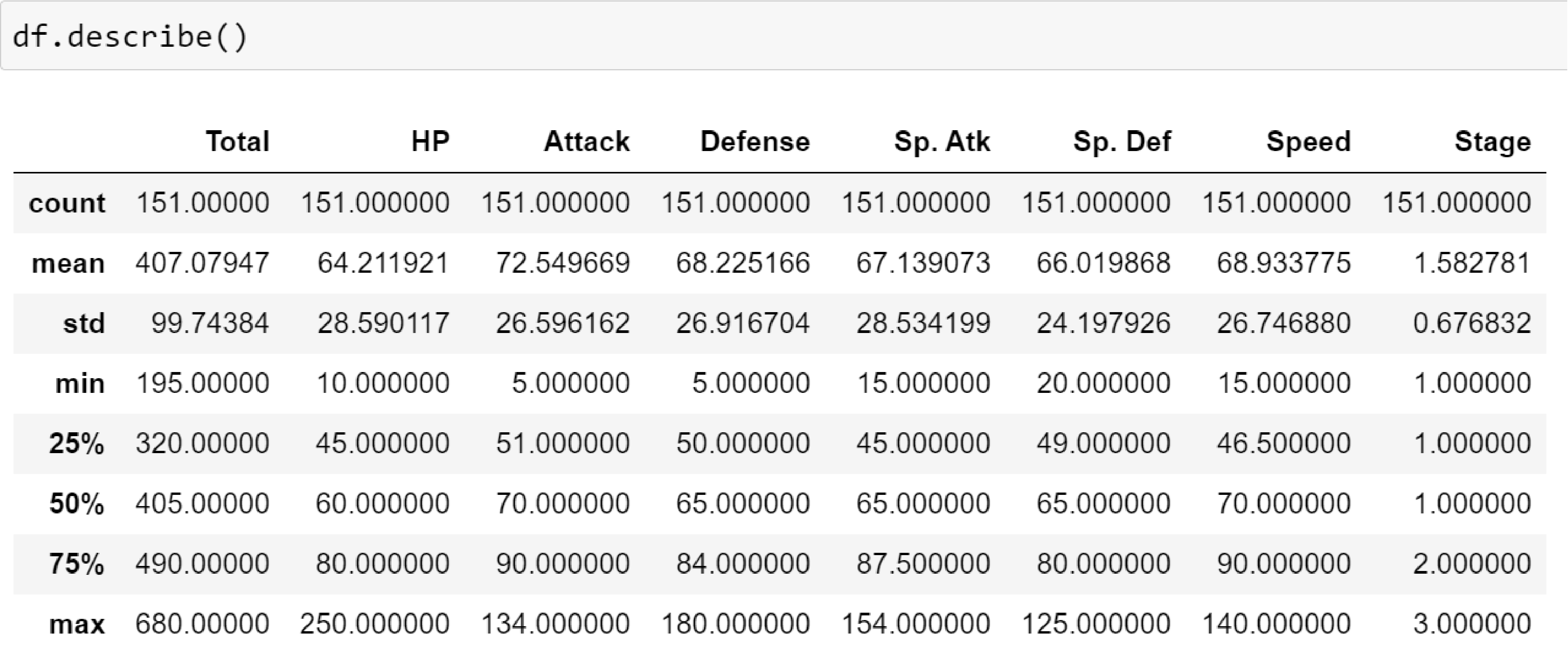


```
df.isnull().sum()
```

```
Name          0
Type 1        0
Type 2       84
Total         0
HP            0
Attack        0
Defense       0
Sp. Atk       0
Sp. Def       0
Speed         0
Stage         0
Legendary     0
dtype: int64
```

Descriptive Statistics

- *df*.describe()



```
df.describe()
```

	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Stage
count	151.00000	151.000000	151.000000	151.000000	151.000000	151.000000	151.000000	151.000000
mean	407.07947	64.211921	72.549669	68.225166	67.139073	66.019868	68.933775	1.582781
std	99.74384	28.590117	26.596162	26.916704	28.534199	24.197926	26.746880	0.676832
min	195.00000	10.000000	5.000000	5.000000	15.000000	20.000000	15.000000	1.000000
25%	320.00000	45.000000	51.000000	50.000000	45.000000	49.000000	46.500000	1.000000
50%	405.00000	60.000000	70.000000	65.000000	65.000000	65.000000	70.000000	1.000000
75%	490.00000	80.000000	90.000000	84.000000	87.500000	80.000000	90.000000	2.000000
max	680.00000	250.000000	134.000000	180.000000	154.000000	125.000000	140.000000	3.000000

Indexing

- Row id = key = label = record id = index
- Used for
  - Accessing individual/multiple rows
  - Aligning multiple DataFrames and Series

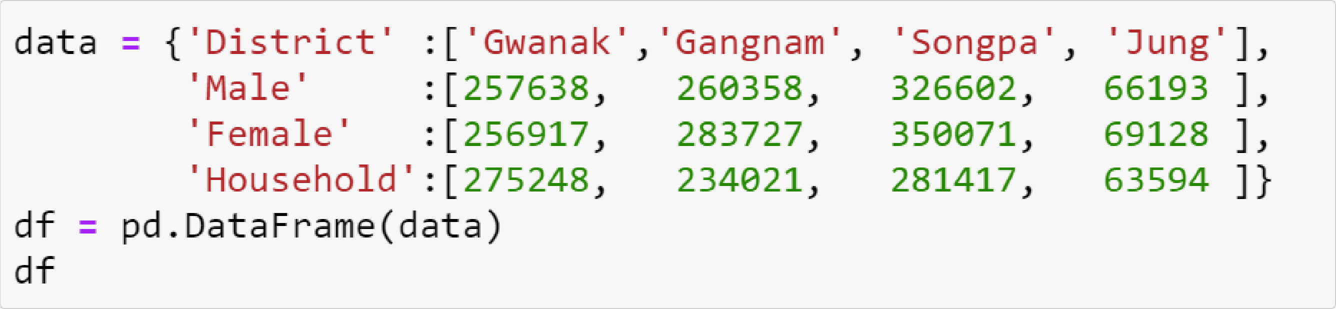
- *df*.`set_index(`*keys*, …`)`
  - Set the DataFrame index using existing column(s)
  - Return a new DataFrame with changed row labels (not in-place update)
  - *keys*: label or array/list of labels

    e.g., `df = df.set_index(['Day','Time'])`

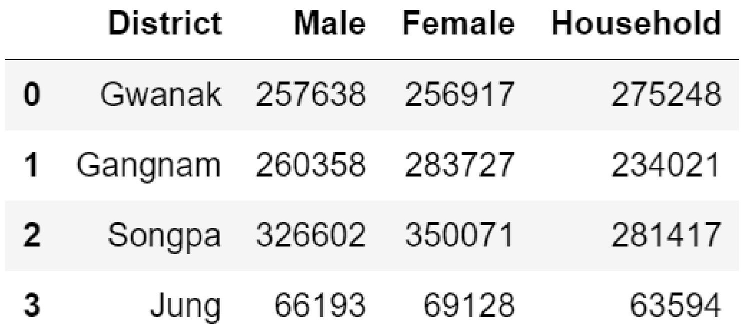
- *df*.`reset_index()` : go back to integer index

Position-based vs. Label-based Index

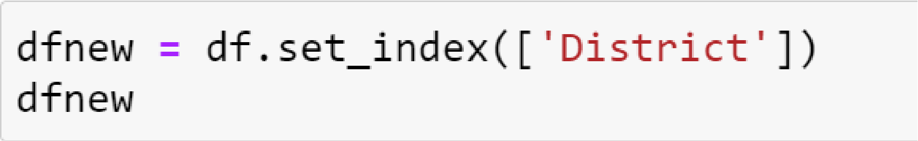
- Use *df*.`set_index()` to set the index using existing column(s)



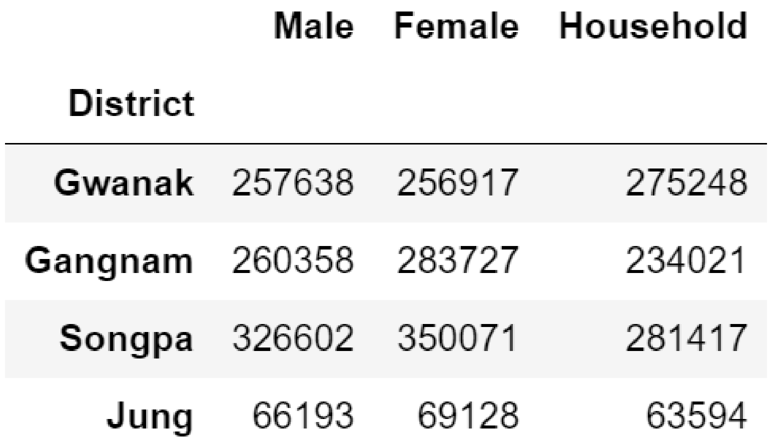
```
data = {'District'  :['Gwanak','Gangnam', 'Songpa', 'Jung'],
        'Male'      :[257638,   260358,   326602,   66193 ],
        'Female'    :[256917,   283727,   350071,   69128 ],
        'Household':[275248,   234021,   281417,   63594 ]}
df = pd.DataFrame(data)
df
```



	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594



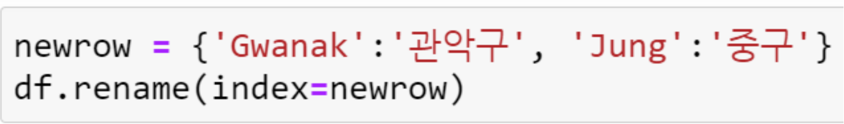
```
dfnew = df.set_index(['District'])
dfnew
```



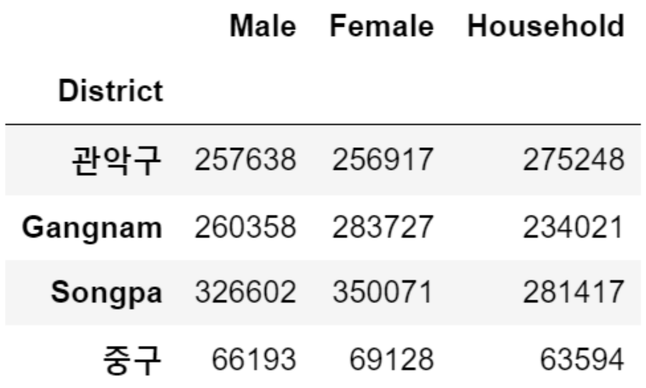
	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594

Renaming Rows/Columns

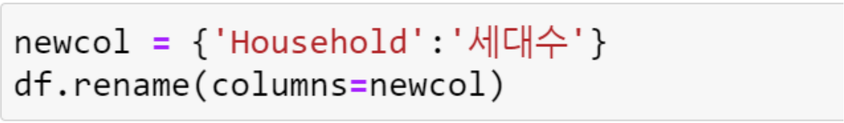
- *df*.rename([*index*], [*columns*], [*inplace*], …)
  - Rename any index, row or column
  - A part of rows or columns can be altered
  - *index*: dict. for changing row indexes
  - *columns*: dict. for changing column indexes
  - *inplace*: If True, *df* is updated in place. Otherwise, return a new *df* (default: False)



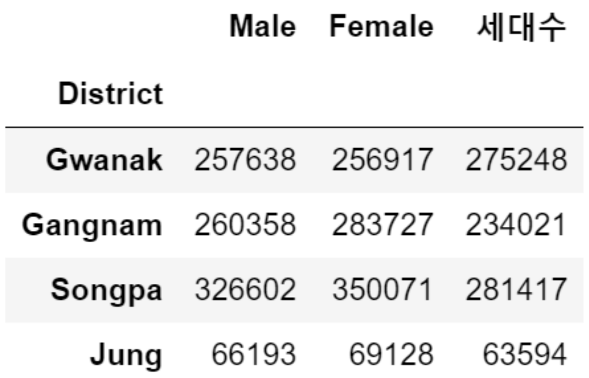
```
newrow = {'Gwanak':'관악구', 'Jung':'중구'}
df.rename(index=newrow)
```



	Male	Female	Household
District			
관악구	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
중구	66193	69128	63594



```
newcol = {'Household':'세대수'}
df.rename(columns=newcol)
```

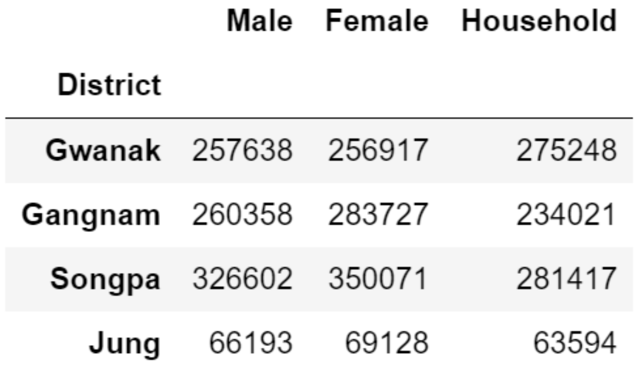


	Male	Female	세대수
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594

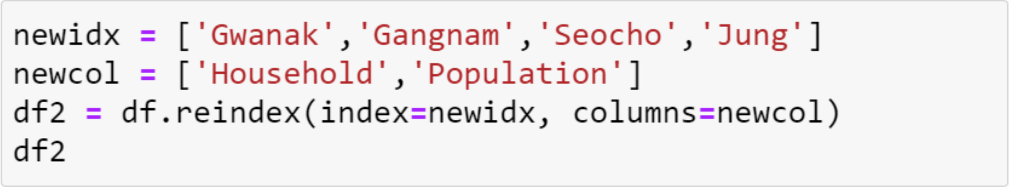
Reindexing

- *df*.reindex([*index*], [*columns*], [*fill_value*], …)
  - Reindex the dataframe with optional filling logic
  - *index*: list for changing row indexes
  - *columns*: list for changing column indexes
  - *fill_value*: value to use for missing values (default: NaN)

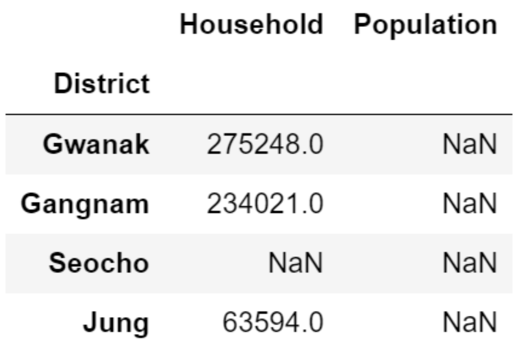
df



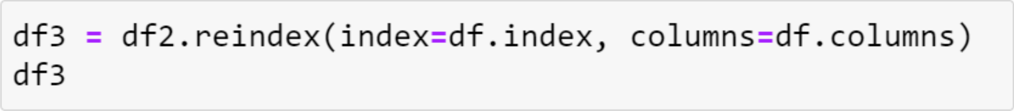
	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



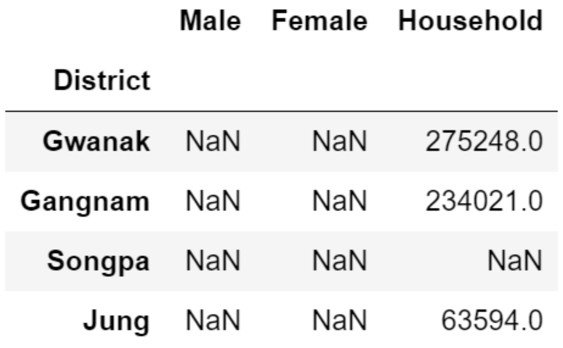
```
newidx = ['Gwanak','Gangnam','Seocho','Jung']
newcol = ['Household','Population']
df2 = df.reindex(index=newidx, columns=newcol)
df2
```



	Household	Population
District		
Gwanak	275248.0	NaN
Gangnam	234021.0	NaN
Seocho	NaN	NaN
Jung	63594.0	NaN



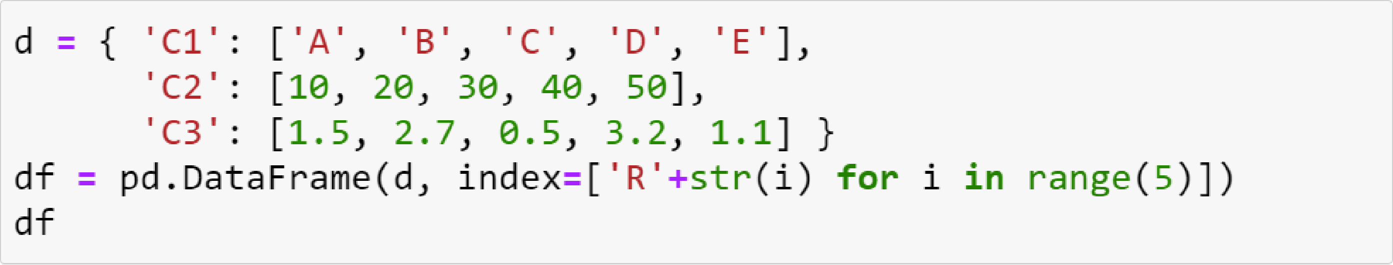
```
df3 = df2.reindex(index=df.index, columns=df.columns)
df3
```



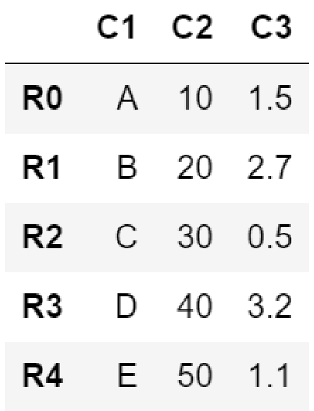
	Male	Female	Household
District			
Gwanak	NaN	NaN	275248.0
Gangnam	NaN	NaN	234021.0
Songpa	NaN	NaN	NaN
Jung	NaN	NaN	63594.0

Transposing a DataFrame

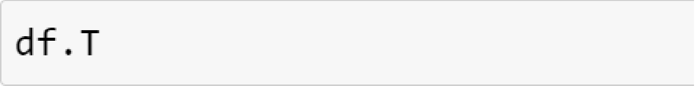
- *df*.transpose() or *df*.T



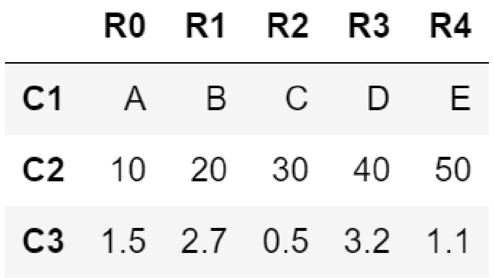
```
d = { 'C1': ['A', 'B', 'C', 'D', 'E'],
      'C2': [10, 20, 30, 40, 50],
      'C3': [1.5, 2.7, 0.5, 3.2, 1.1] }
df = pd.DataFrame(d, index=['R'+str(i) for i in range(5)])
df
```



	C1	C2	C3
R0	A	10	1.5
R1	B	20	2.7
R2	C	30	0.5
R3	D	40	3.2
R4	E	50	1.1



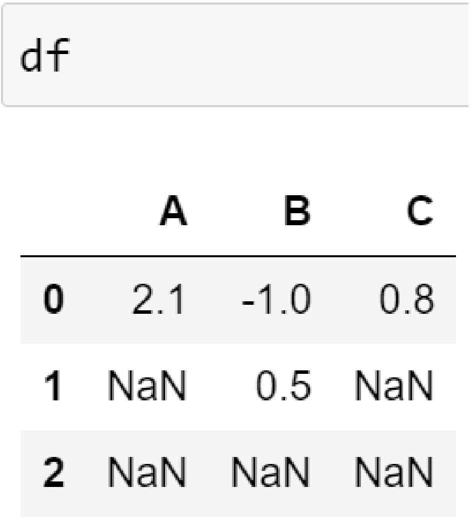
```
df.T
```



	R0	R1	R2	R3	R4
C1	A	B	C	D	E
C2	10	20	30	40	50
C3	1.5	2.7	0.5	3.2	1.1

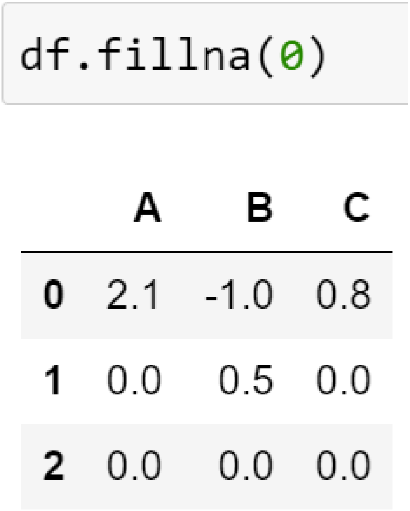
Filling / Dropping NANs

- *df*.`fillna(`*value, method, axis, ...*`)`
  - *value*: value to use to fill holes
  - *method*: 'bfill', 'ffill', or None (default: None)
  - *axis*: axis along which to fill missing values



```
df
```

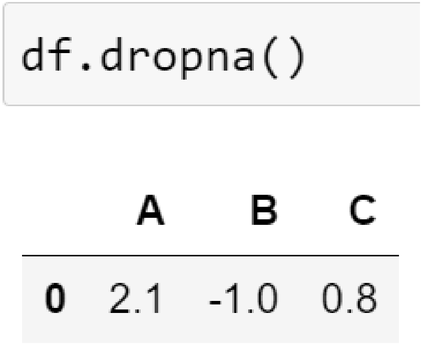
	A	B	C
0	2.1	-1.0	0.8
1	NaN	0.5	NaN
2	NaN	NaN	NaN



```
df.fillna(0)
```

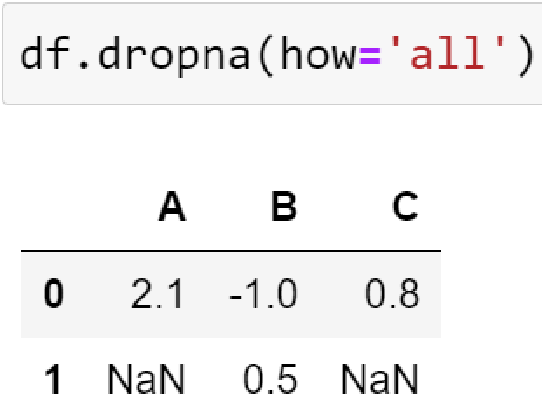
	A	B	C
0	2.1	-1.0	0.8
1	0.0	0.5	0.0
2	0.0	0.0	0.0

- *df*.`dropna(`*axis, how, ...*`)`
  - *axis*: drop rows (0) or columns (1) which contain missing values (default: 0)
  - *how*: 'any' or 'all'. If any (or all) values are NAN, drop that row or column. (default: 'any'



```
df.dropna()
```

	A	B	C
0	2.1	-1.0	0.8

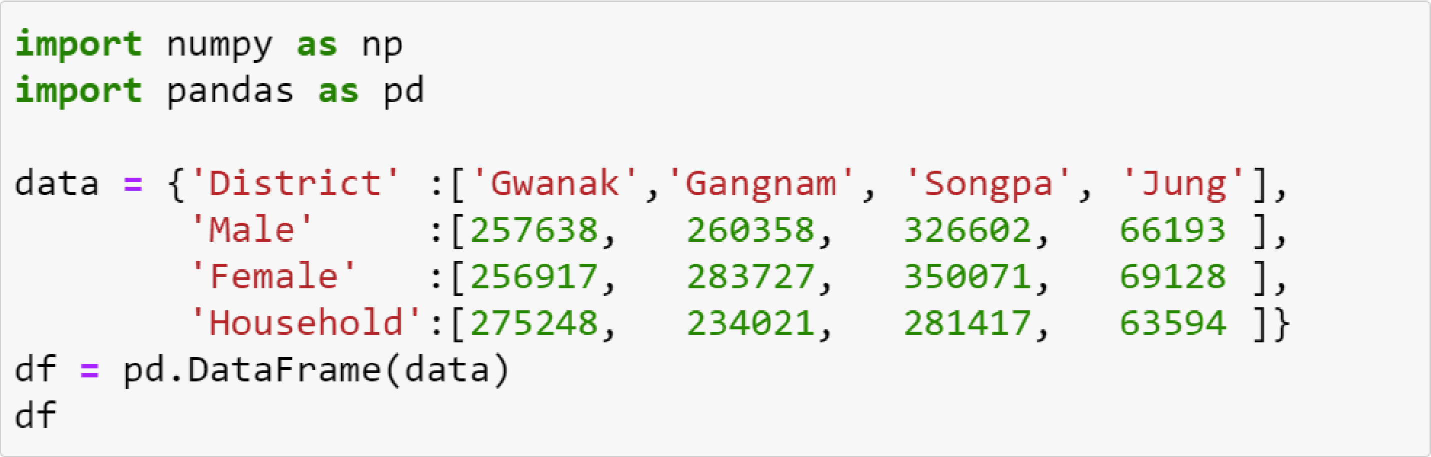


```
df.dropna(how='all')
```

	A	B	C
0	2.1	-1.0	0.8
1	NaN	0.5	NaN

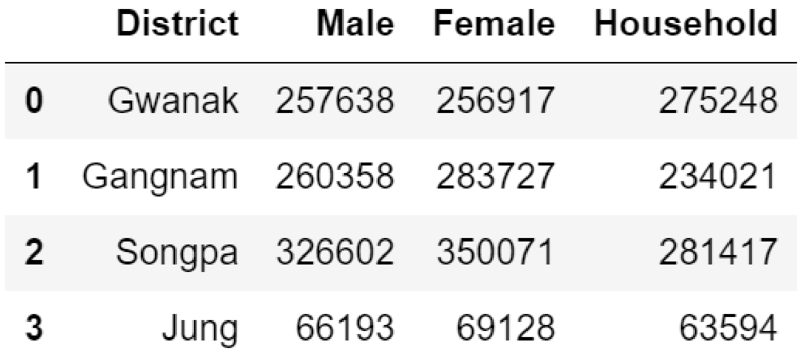
Manipulating Columns

Selecting Columns

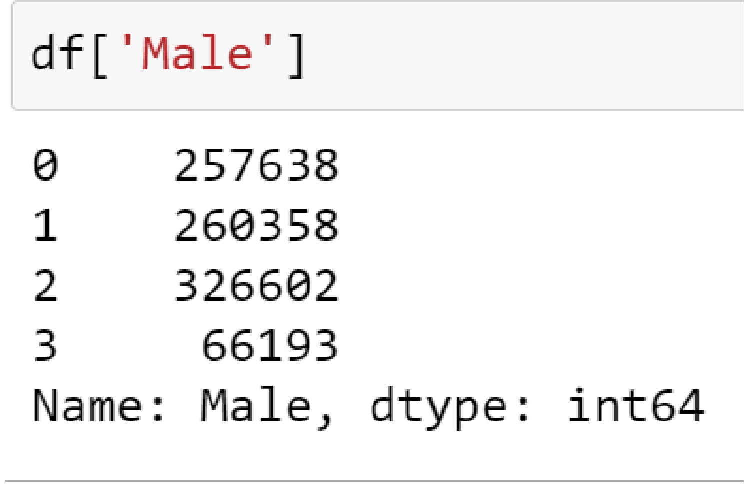


```
import numpy as np
import pandas as pd

data = {'District'  :['Gwanak','Gangnam', 'Songpa', 'Jung'],
        'Male'      :[257638,   260358,   326602,   66193 ],
        'Female'    :[256917,   283727,   350071,   69128 ],
        'Household':[275248,   234021,   281417,   63594 ]}
df = pd.DataFrame(data)
df
```

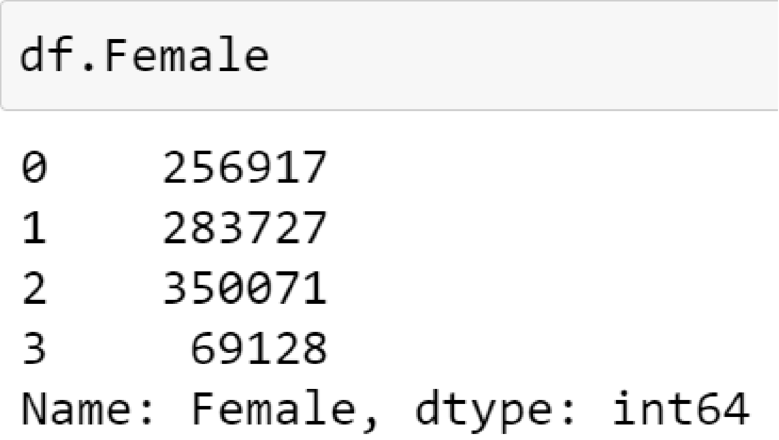


	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594



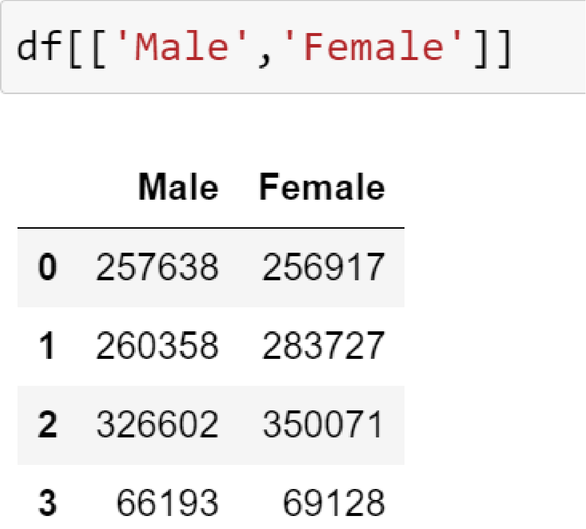
```
df['Male']
```

```
0     257638
1     260358
2     326602
3      66193
Name: Male, dtype: int64
```



```
df.Female
```

```
0     256917
1     283727
2     350071
3      69128
Name: Female, dtype: int64
```

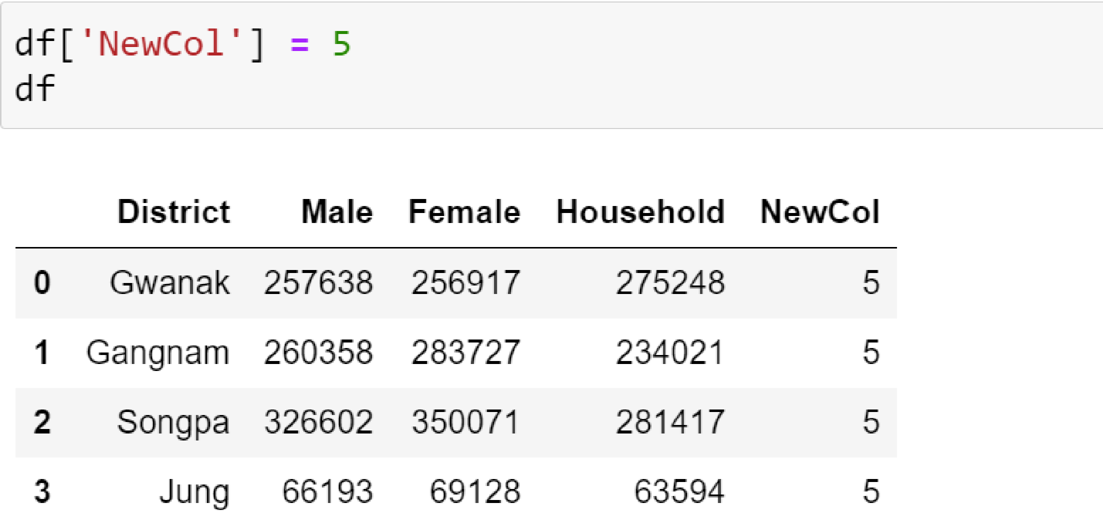


```
df[['Male','Female']]
```

	Male	Female
0	257638	256917
1	260358	283727
2	326602	350071
3	66193	69128

Adding a Column (1)

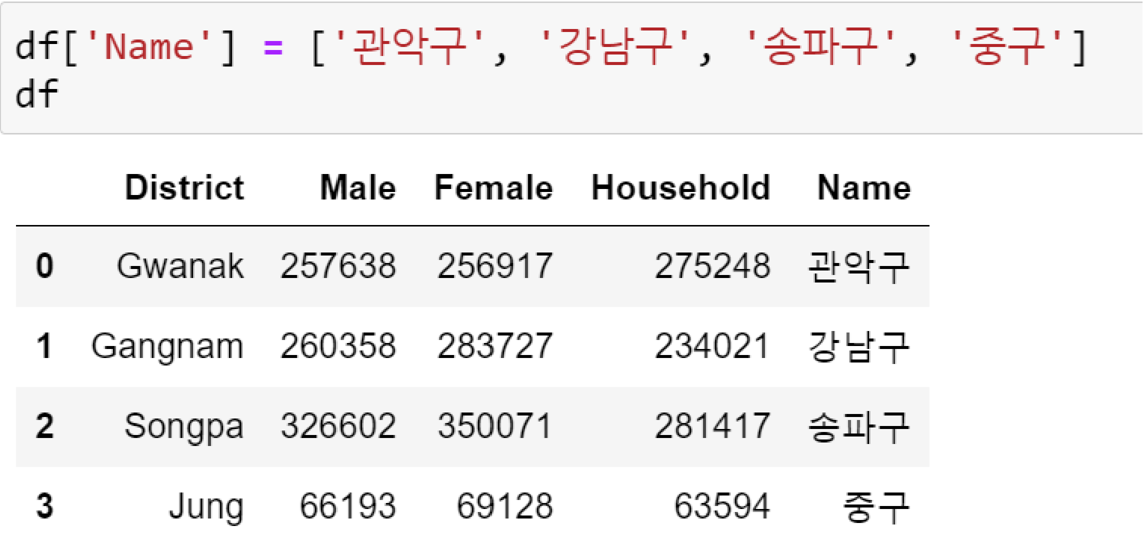
- With the same initial value



```
df['NewCol'] = 5
df
```

	District	Male	Female	Household	NewCol
0	Gwanak	257638	256917	275248	5
1	Gangnam	260358	283727	234021	5
2	Songpa	326602	350071	281417	5
3	Jung	66193	69128	63594	5

- With new values



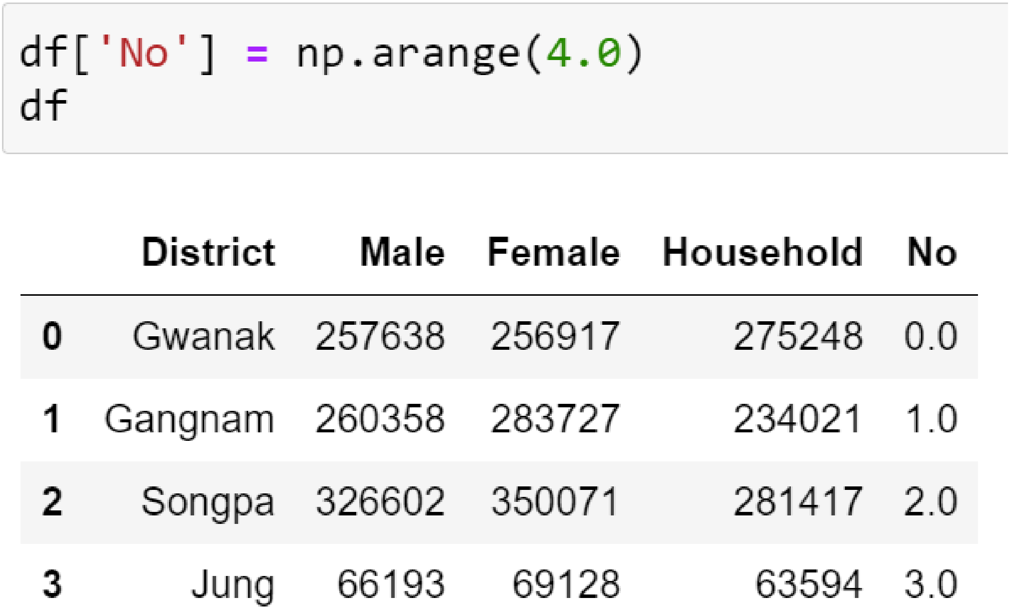
```
df['Name'] = ['관악구', '강남구', '송파구', '중구']
df
```

	District	Male	Female	Household	Name
0	Gwanak	257638	256917	275248	관악구
1	Gangnam	260358	283727	234021	강남구
2	Songpa	326602	350071	281417	송파구
3	Jung	66193	69128	63594	중구

Also works for the existing column (e.g., `df.Female = 1`)

Adding a Column (2)

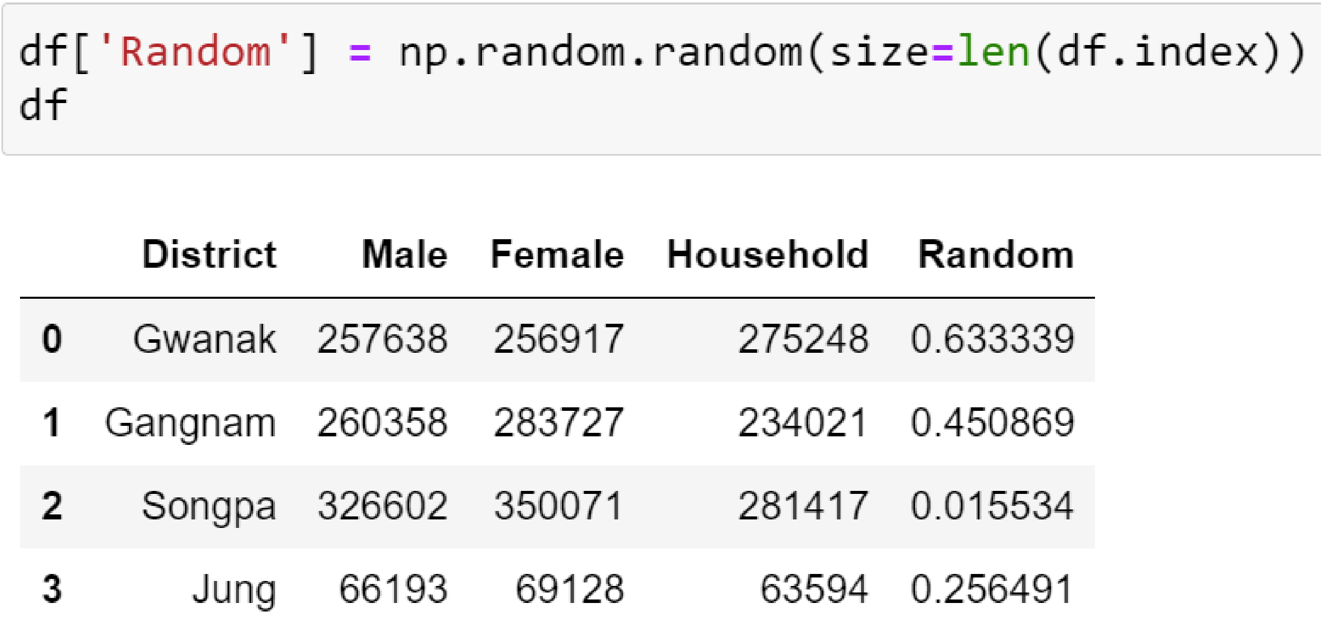
- With the sequential number



```
df['No'] = np.arange(4.0)
df
```

	District	Male	Female	Household	No
0	Gwanak	257638	256917	275248	0.0
1	Gangnam	260358	283727	234021	1.0
2	Songpa	326602	350071	281417	2.0
3	Jung	66193	69128	63594	3.0

- With random numbers

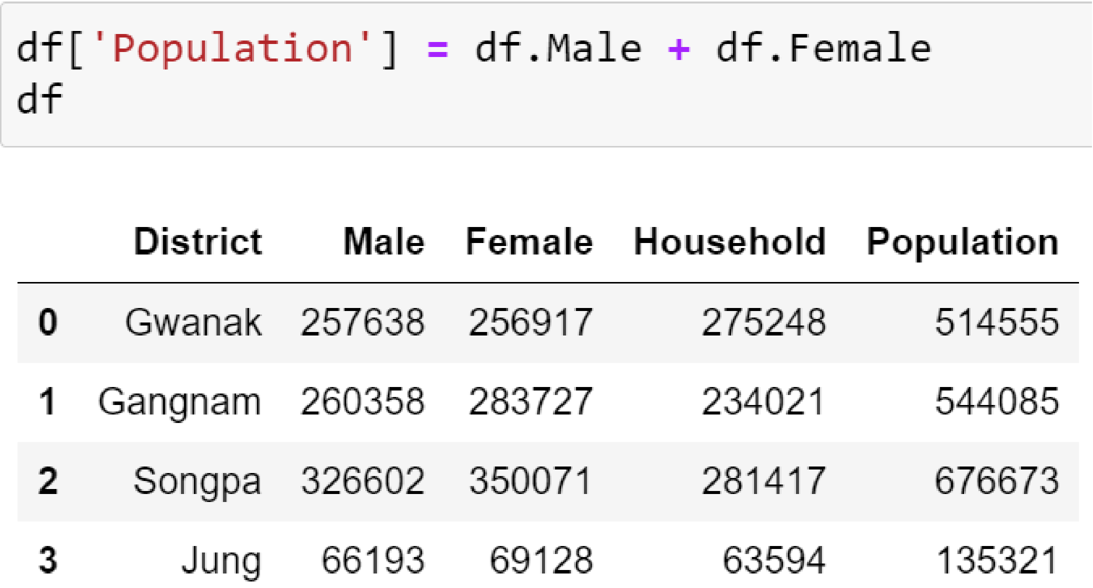


```
df['Random'] = np.random.random(size=len(df.index))
df
```

	District	Male	Female	Household	Random
0	Gwanak	257638	256917	275248	0.633339
1	Gangnam	260358	283727	234021	0.450869
2	Songpa	326602	350071	281417	0.015534
3	Jung	66193	69128	63594	0.256491

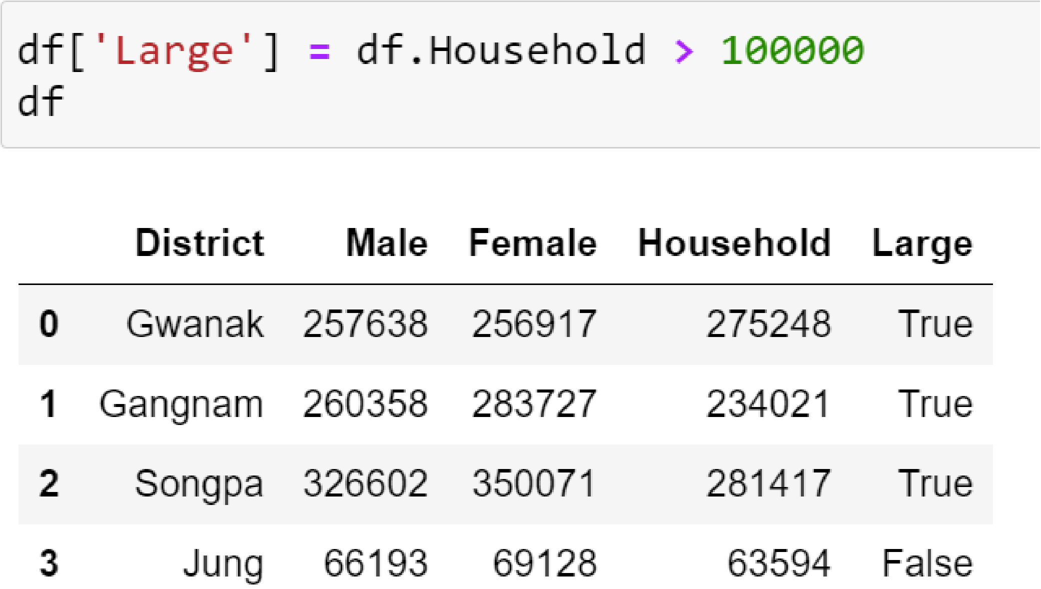
Adding a Column (3)

- With expressions



```
df['Population'] = df.Male + df.Female
df
```

	District	Male	Female	Household	Population
0	Gwanak	257638	256917	275248	514555
1	Gangnam	260358	283727	234021	544085
2	Songpa	326602	350071	281417	676673
3	Jung	66193	69128	63594	135321

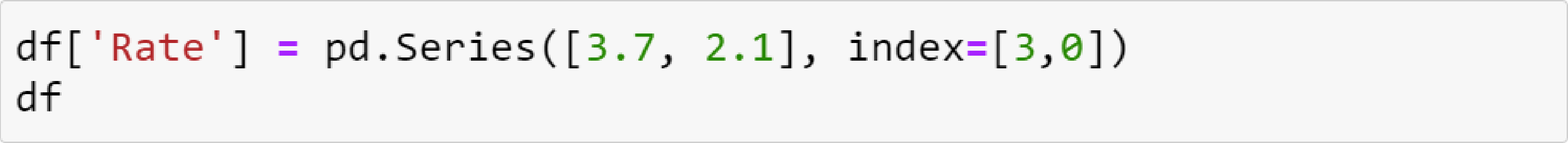


```
df['Large'] = df.Household > 100000
df
```

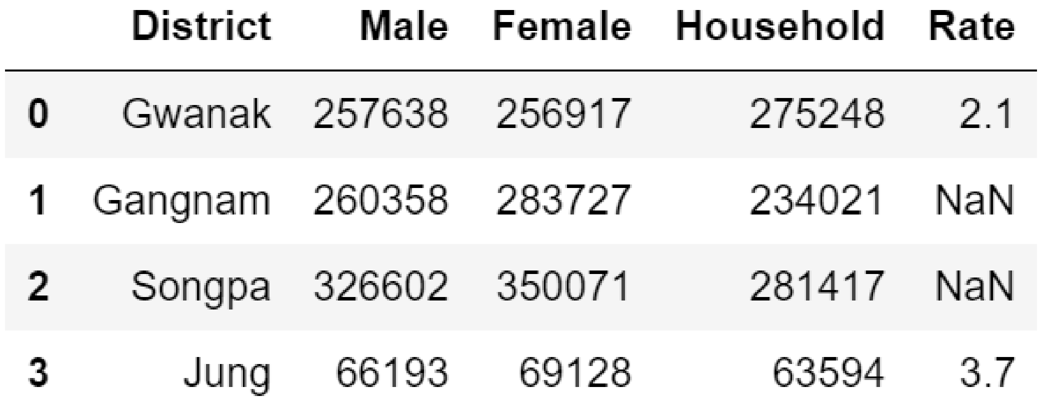
	District	Male	Female	Household	Large
0	Gwanak	257638	256917	275248	True
1	Gangnam	260358	283727	234021	True
2	Songpa	326602	350071	281417	True
3	Jung	66193	69128	63594	False

Adding a Column (4)

- With a Series
  - Unlike a list, the length can be smaller than the column size
  - Its labels will be realigned exactly to the DataFrame's index, insert missing values in any holes



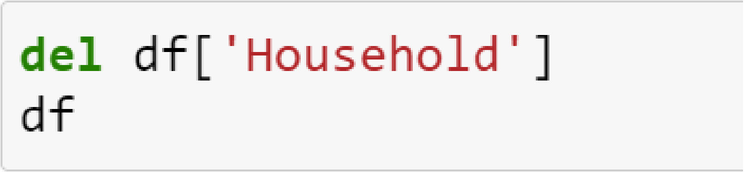
```
df['Rate'] = pd.Series([3.7, 2.1], index=[3,0])
df
```



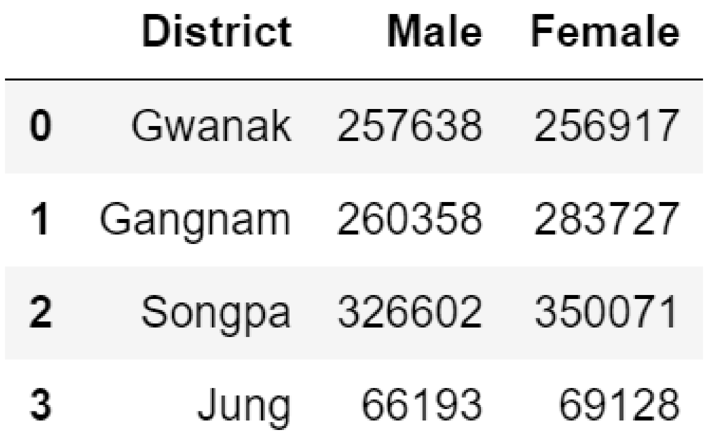
	District	Male	Female	Household	Rate
0	Gwanak	257638	256917	275248	2.1
1	Gangnam	260358	283727	234021	NaN
2	Songpa	326602	350071	281417	NaN
3	Jung	66193	69128	63594	3.7

Deleting Columns

- Using del
  - Delete in-place
  - Only one column at a time



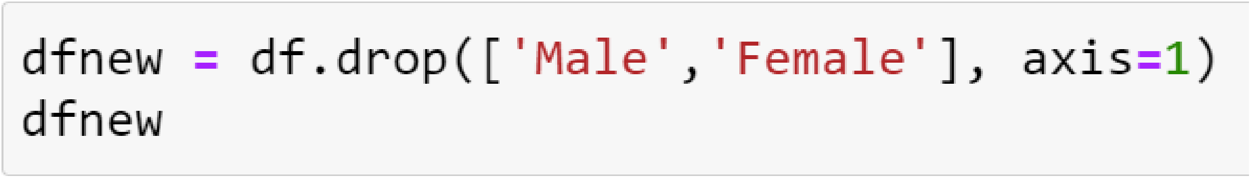
```
del df['Household']
df
```



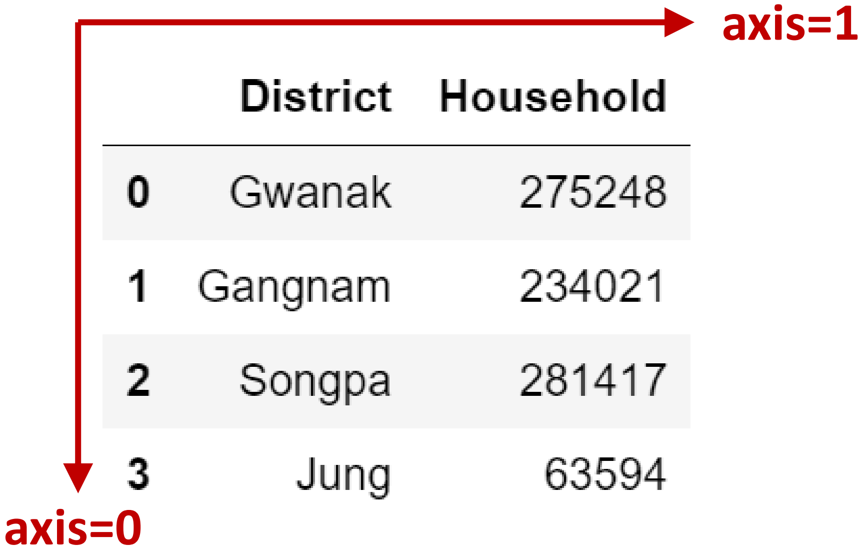
	District	Male	Female
0	Gwanak	257638	256917
1	Gangnam	260358	283727
2	Songpa	326602	350071
3	Jung	66193	69128

- Using df.drop()
  - Return a new DataFrame (*inplace*=False)
  - Also used to drop rows

or axis='columns'



```
dfnew = df.drop(['Male','Female'], axis=1)
dfnew
```

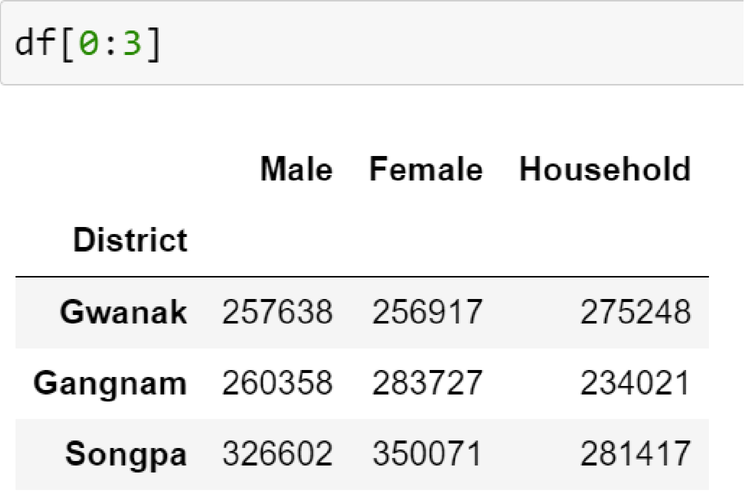


	District	Household
0	Gwanak	275248
1	Gangnam	234021
2	Songpa	281417
3	Jung	63594

Manipulating Rows

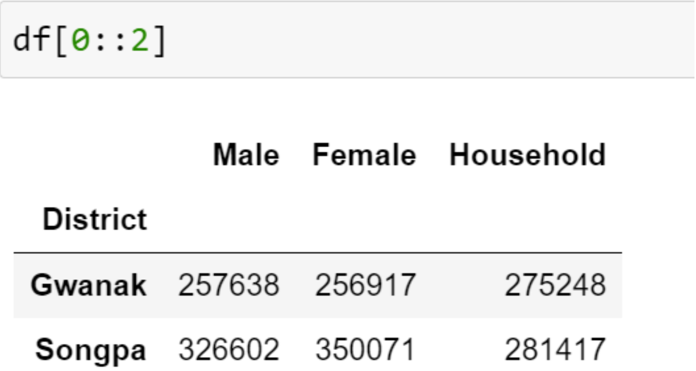
Slicing Rows Using [ ]

- *df*[*start* : *stop* : *step*]



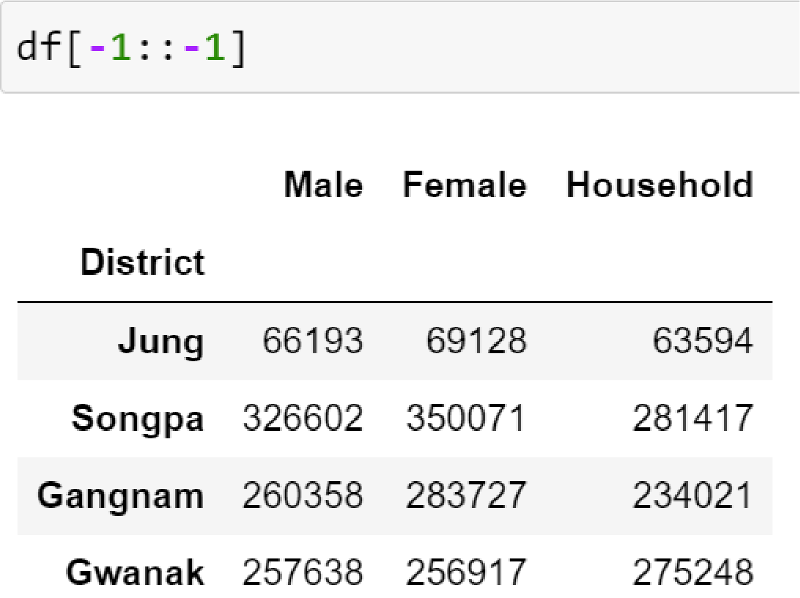
```
df[0:3]
```

	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417



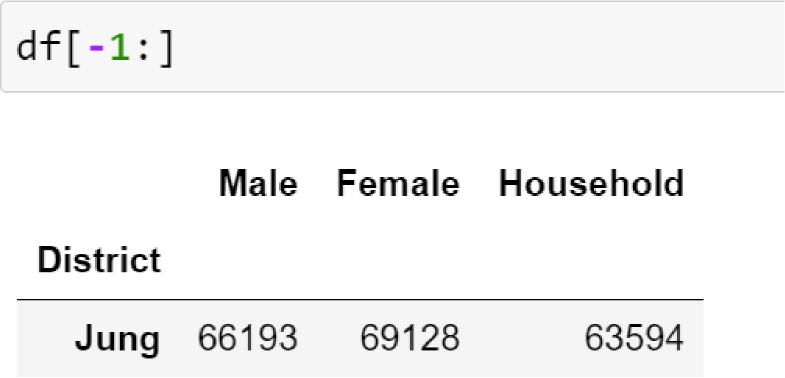
```
df[0::2]
```

	Male	Female	Household
District			
Gwanak	257638	256917	275248
Songpa	326602	350071	281417



```
df[-1::-1]
```

	Male	Female	Household
District			
Jung	66193	69128	63594
Songpa	326602	350071	281417
Gangnam	260358	283727	234021
Gwanak	257638	256917	275248

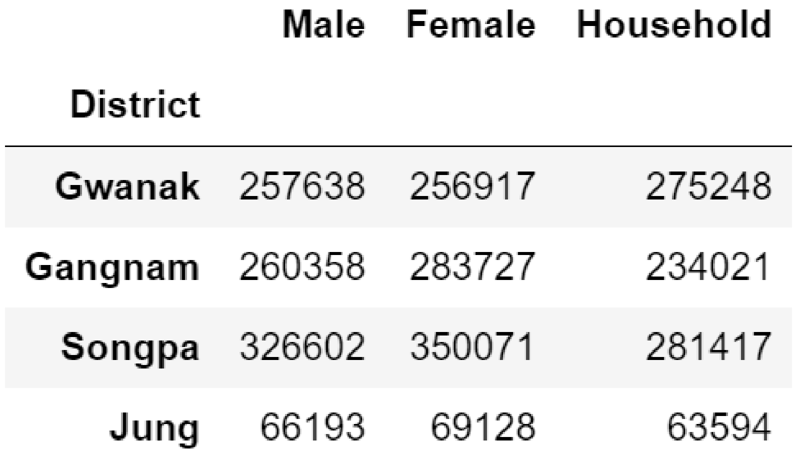


```
df[-1:]
```

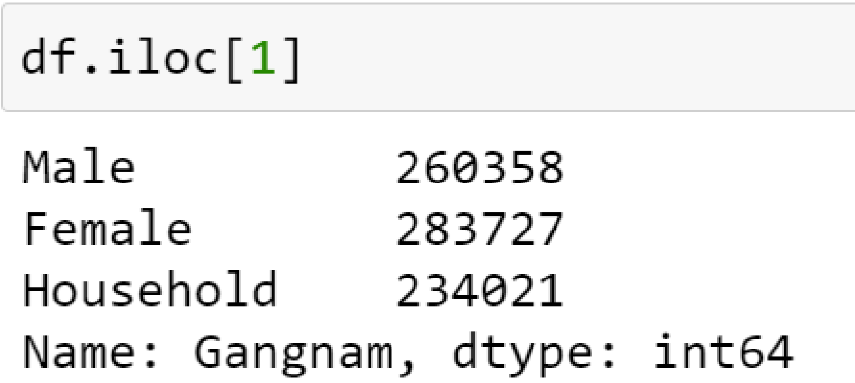
	Male	Female	Household
District			
Jung	66193	69128	63594

Selecting Rows by Integer Location

- *df*.iloc[*loc*]

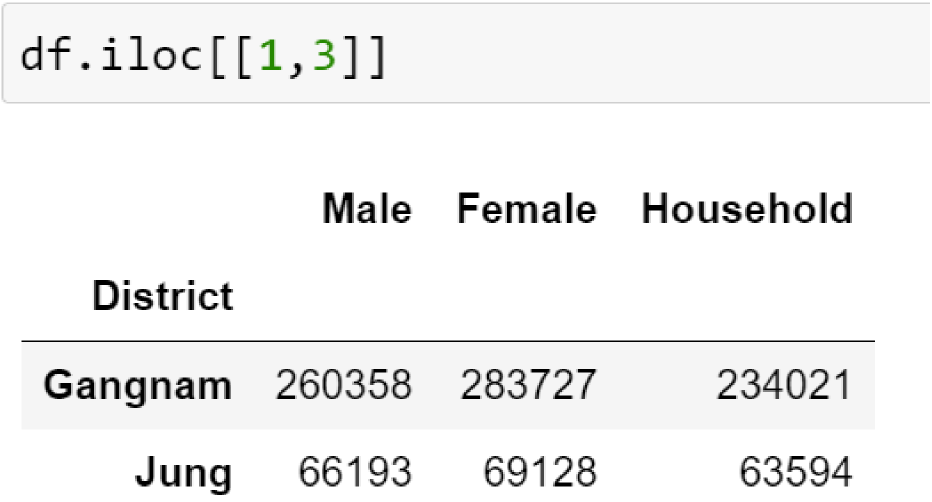


District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



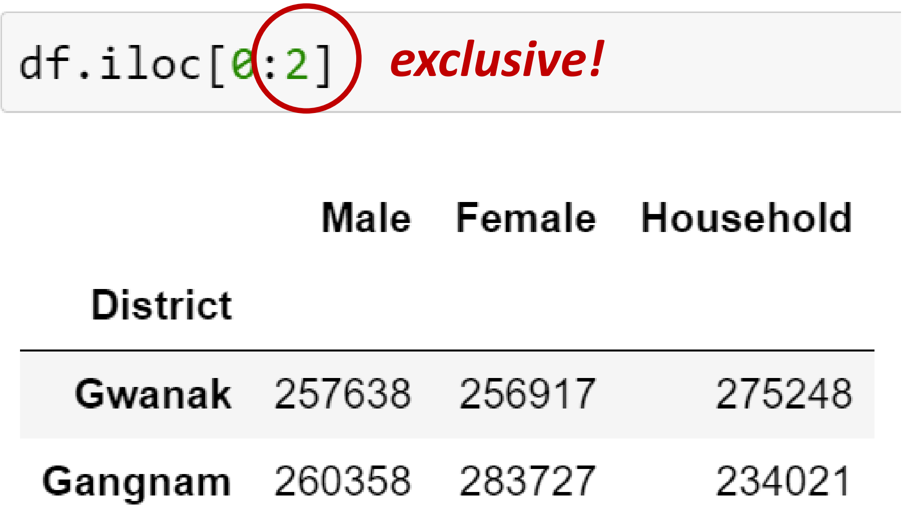
```
df.iloc[1]
```

```
Male         260358
Female       283727
Household    234021
Name: Gangnam, dtype: int64
```



```
df.iloc[[1,3]]
```

District	Male	Female	Household
Gangnam	260358	283727	234021
Jung	66193	69128	63594



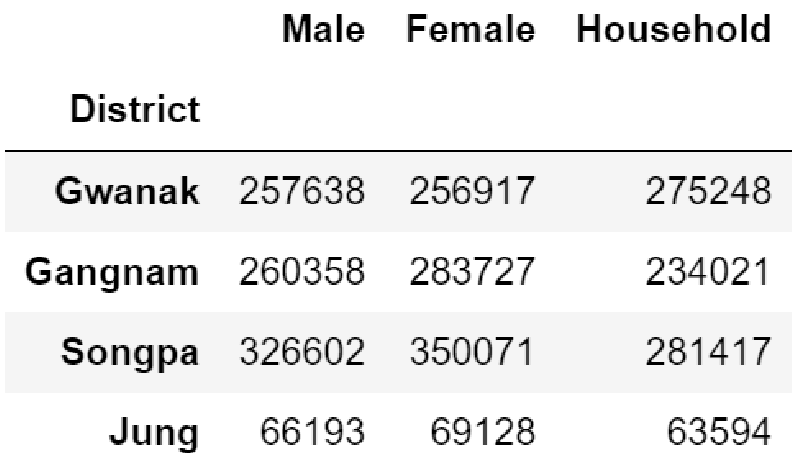
```
df.iloc[0:2]
```

exclusive!

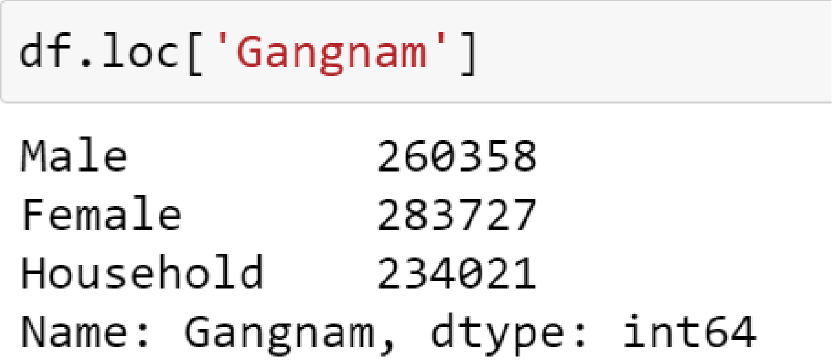
District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021

Selecting Rows by Label

- *df*.loc[*label*]

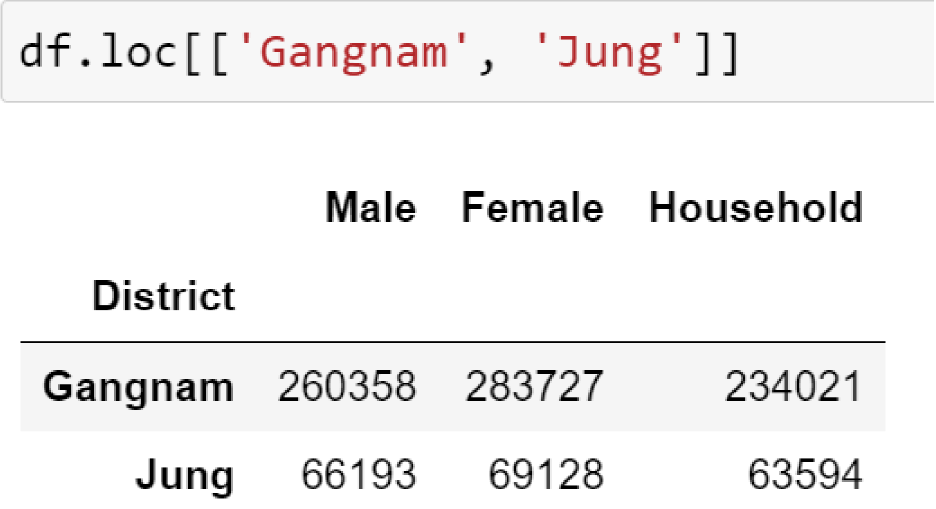


District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



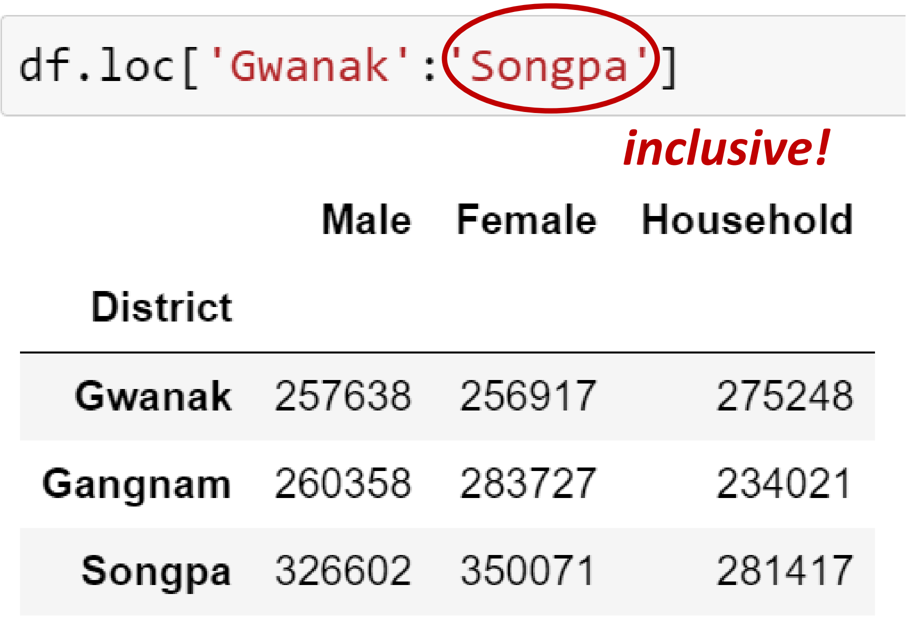
```
df.loc['Gangnam']
```

```
Male         260358
Female       283727
Household    234021
Name: Gangnam, dtype: int64
```



```
df.loc[['Gangnam', 'Jung']]
```

District	Male	Female	Household
Gangnam	260358	283727	234021
Jung	66193	69128	63594



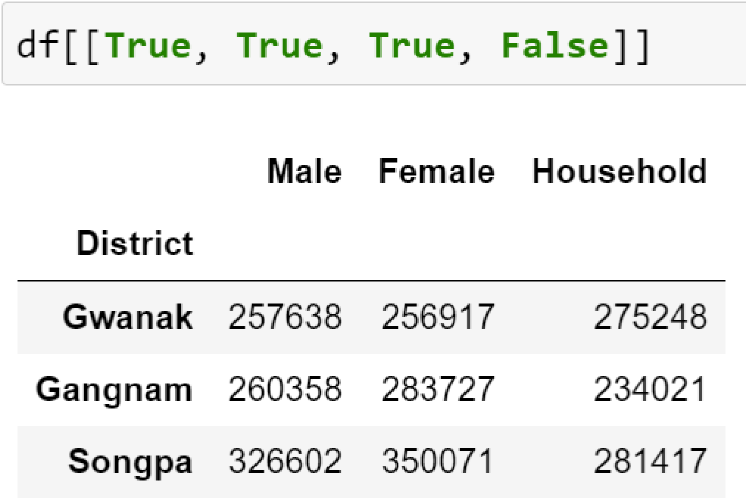
```
df.loc['Gwanak':'Songpa']
```

inclusive!

District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417

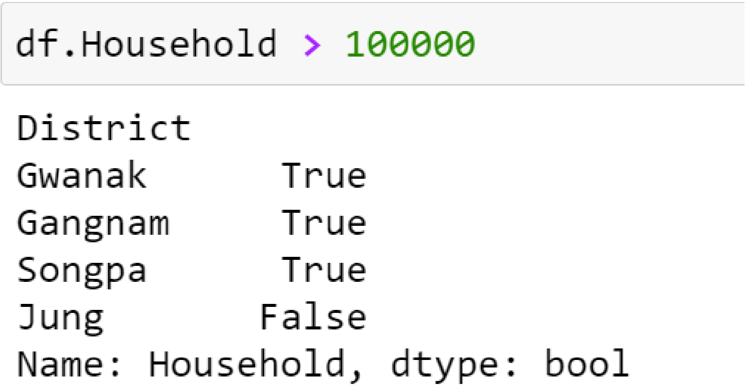
Selecting Rows by Boolean Vector (1)

- *df[bool_vec]*



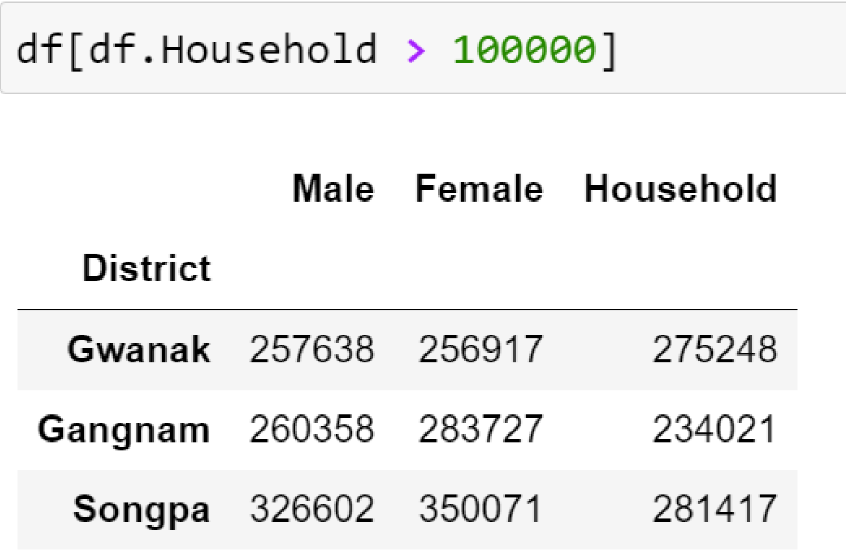
```
df[[True, True, True, False]]
```

District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417



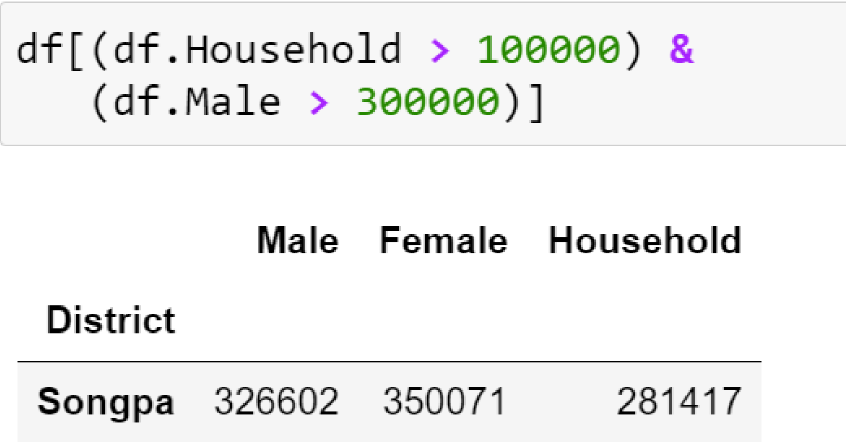
```
df.Household > 100000
```

```
District
Gwanak        True
Gangnam       True
Songpa        True
Jung         False
Name: Household, dtype: bool
```



```
df[df.Household > 100000]
```

District	Male	Female	Household
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417



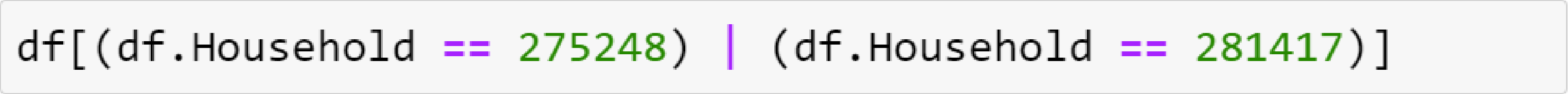
```
df[(df.Household > 100000) &
   (df.Male > 300000)]
```

District	Male	Female	Household
Songpa	326602	350071	281417

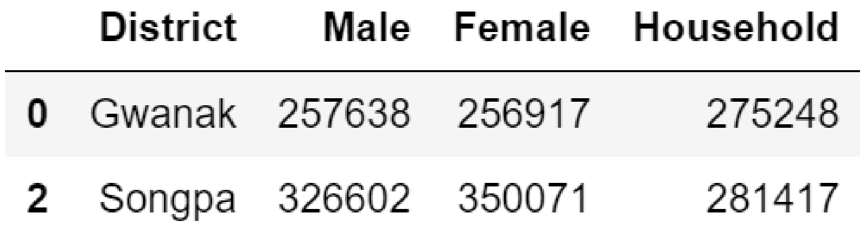
and	*&*
or	*|*
not	*~*
✗	✓

Selecting Rows by Boolean Vector (2)

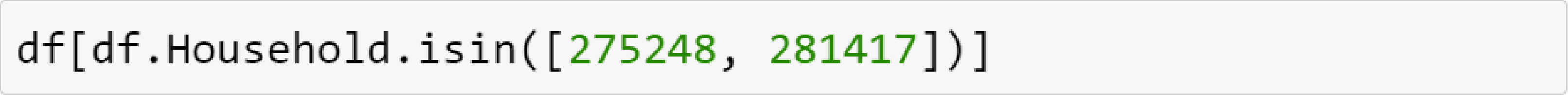
- Use `isin()` to select all rows whose values contain the specified value(s)



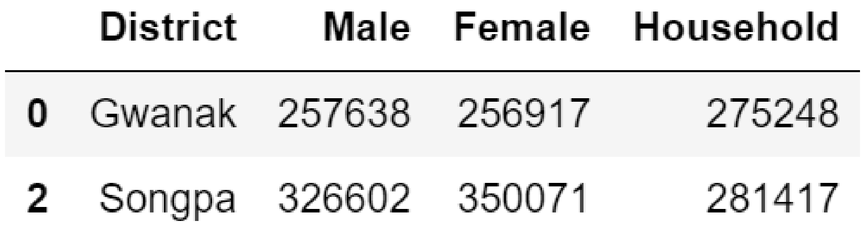
```
df[(df.Household == 275248) | (df.Household == 281417)]
```



	District	Male	Female	Household
0	Gwanak	257638	256917	275248
2	Songpa	326602	350071	281417

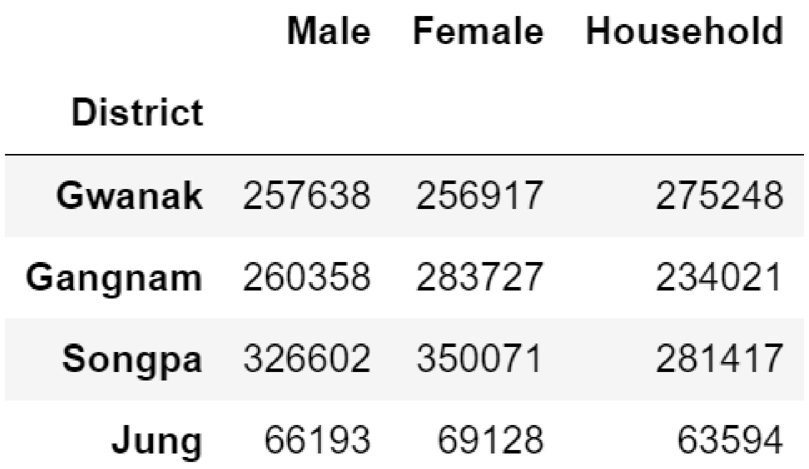


```
df[df.Household.isin([275248, 281417])]
```

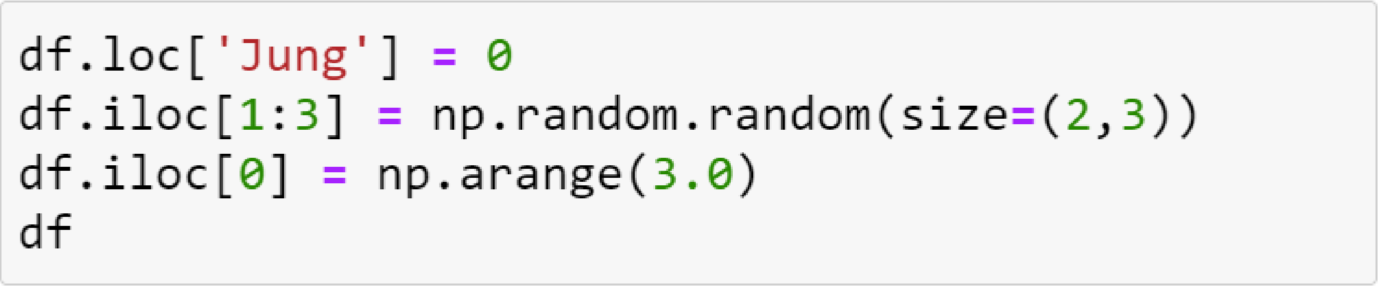


	District	Male	Female	Household
0	Gwanak	257638	256917	275248
2	Songpa	326602	350071	281417

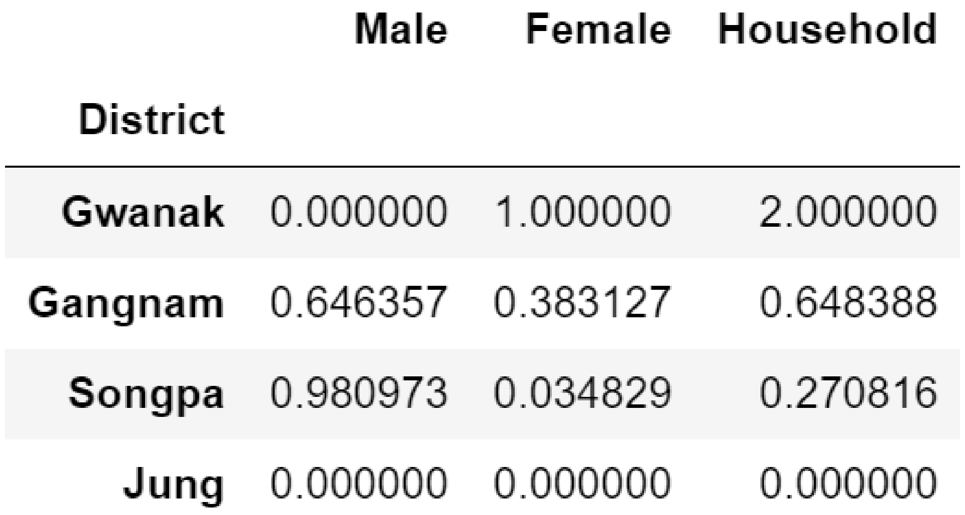
Changing Rows



	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



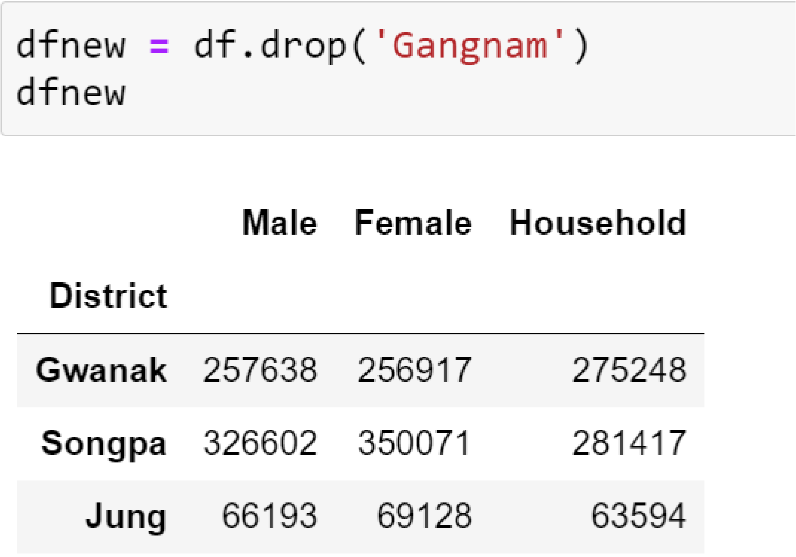
```
df.loc['Jung'] = 0
df.iloc[1:3] = np.random.random(size=(2,3))
df.iloc[0] = np.arange(3.0)
df
```



	Male	Female	Household
District			
Gwanak	0.000000	1.000000	2.000000
Gangnam	0.646357	0.383127	0.648388
Songpa	0.980973	0.034829	0.270816
Jung	0.000000	0.000000	0.000000

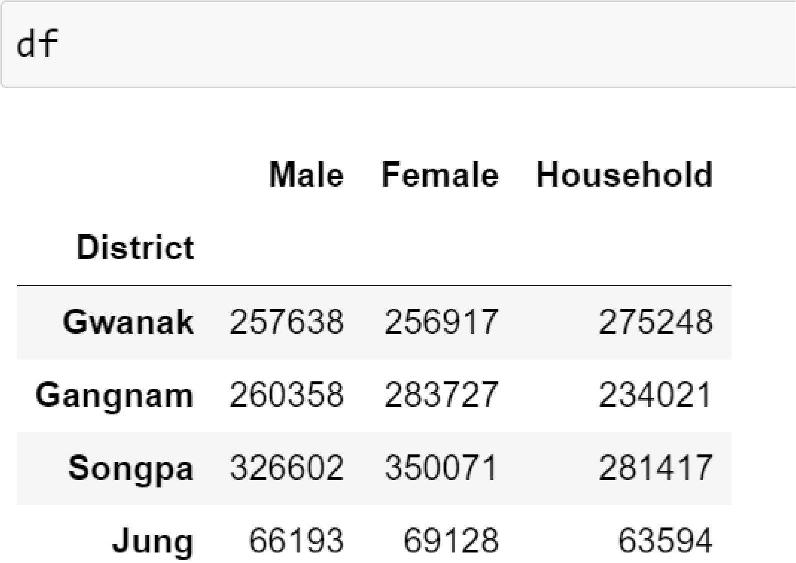
Deleting Rows

- Using `df.drop()`
  - Return a new DataFrame
  - Use *inplace*=True for in-place deletion



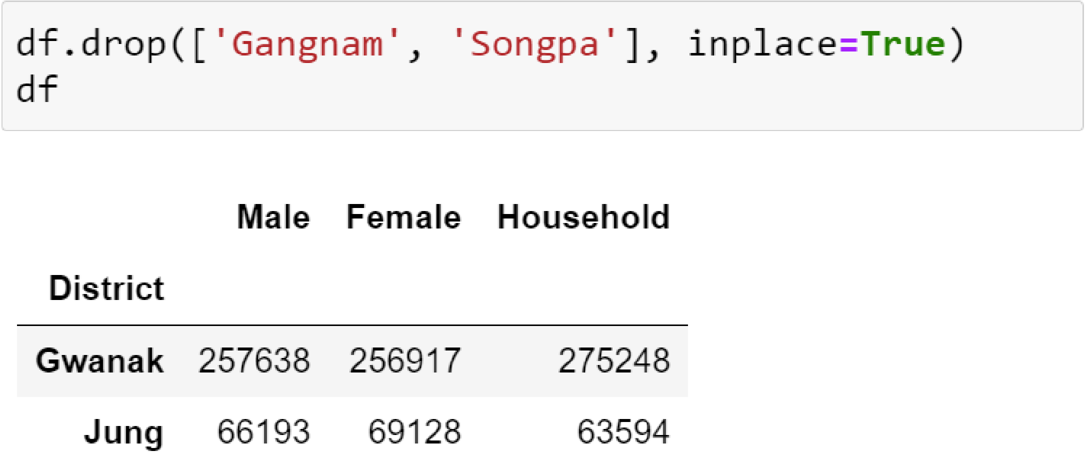
```
dfnew = df.drop('Gangnam')
dfnew
```

	Male	Female	Household
District			
Gwanak	257638	256917	275248
Songpa	326602	350071	281417
Jung	66193	69128	63594



```
df
```

	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594

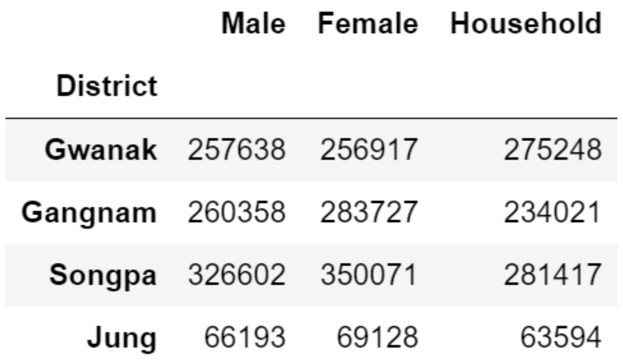


```
df.drop(['Gangnam', 'Songpa'], inplace=True)
df
```

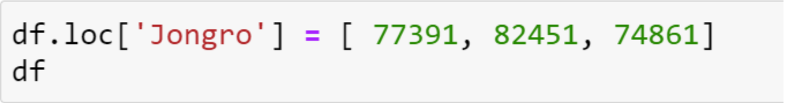
	Male	Female	Household
District			
Gwanak	257638	256917	275248
Jung	66193	69128	63594

Adding Rows

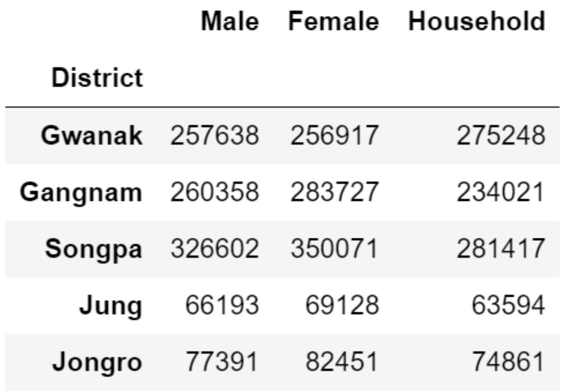
- Use *df*.loc[ ]



	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594

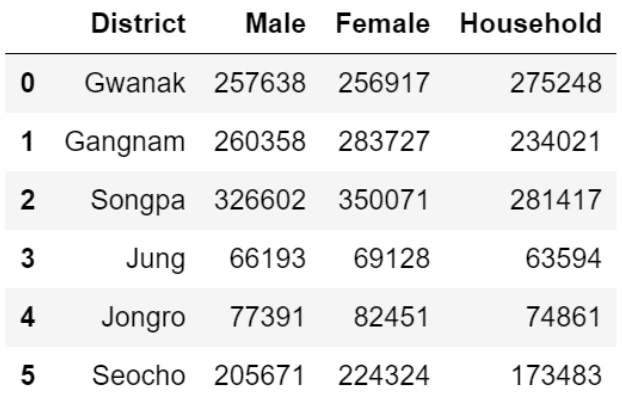


```
df.loc['Jongro'] = [ 77391, 82451, 74861]
df
```

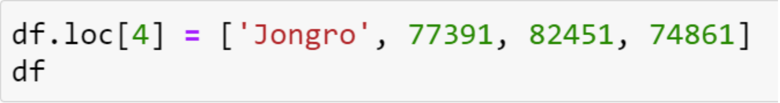


	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594
Jongro	77391	82451	74861

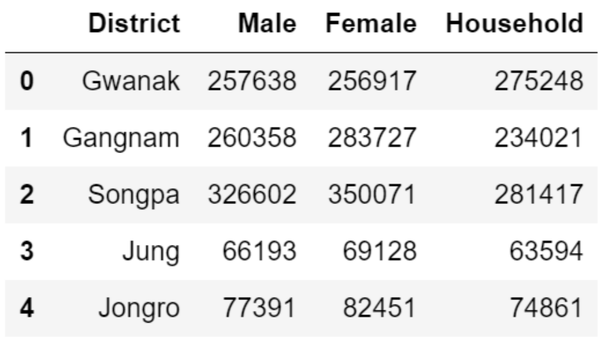
You can also use *pd*.concat() or *df*.append()



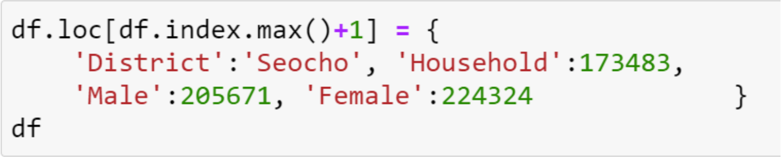
	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594
4	Jongro	77391	82451	74861
5	Seocho	205671	224324	173483



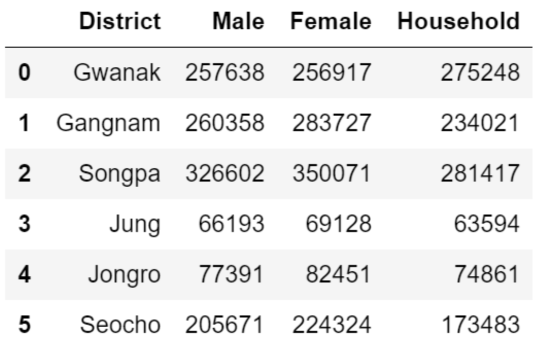
```
df.loc[4] = ['Jongro', 77391, 82451, 74861]
df
```



	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594
4	Jongro	77391	82451	74861

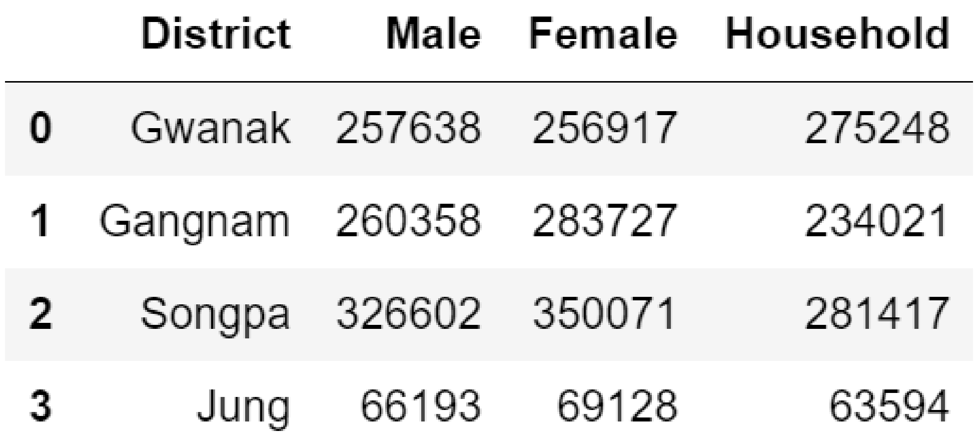


```
df.loc[df.index.max()+1] = {
    'District':'Seocho', 'Household':173483,
    'Male':205671, 'Female':224324                  }
df
```

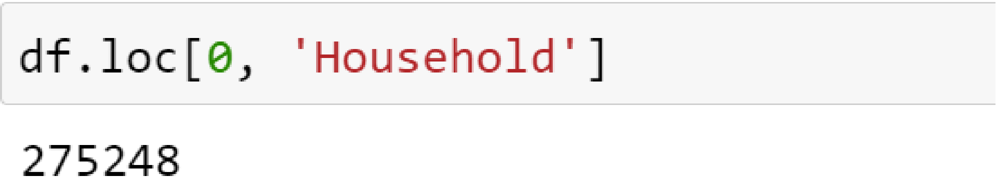


	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594
4	Jongro	77391	82451	74861
5	Seocho	205671	224324	173483

Accessing Values

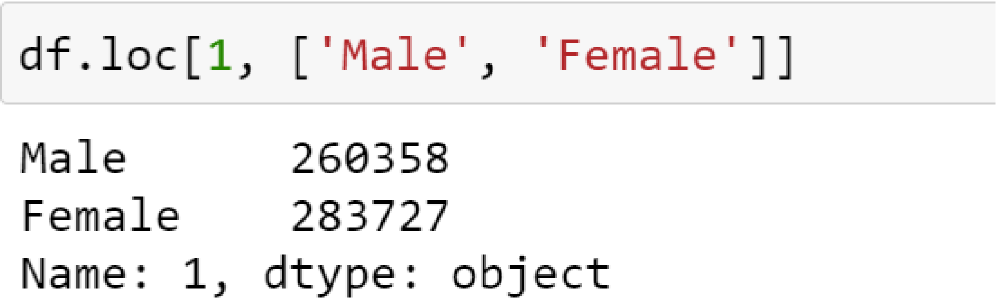


	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594



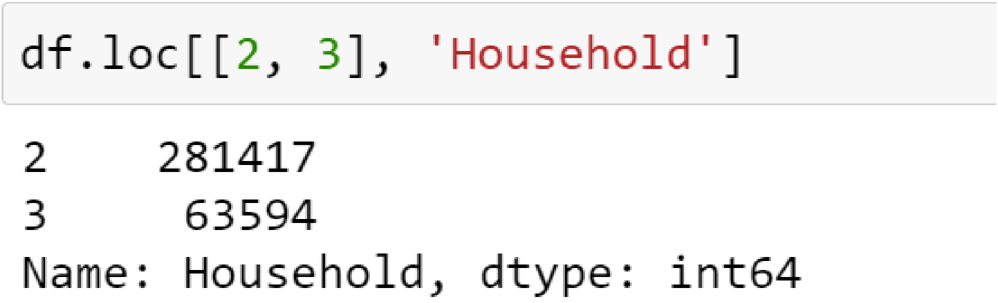
```
df.loc[0, 'Household']
```

```
275248
```



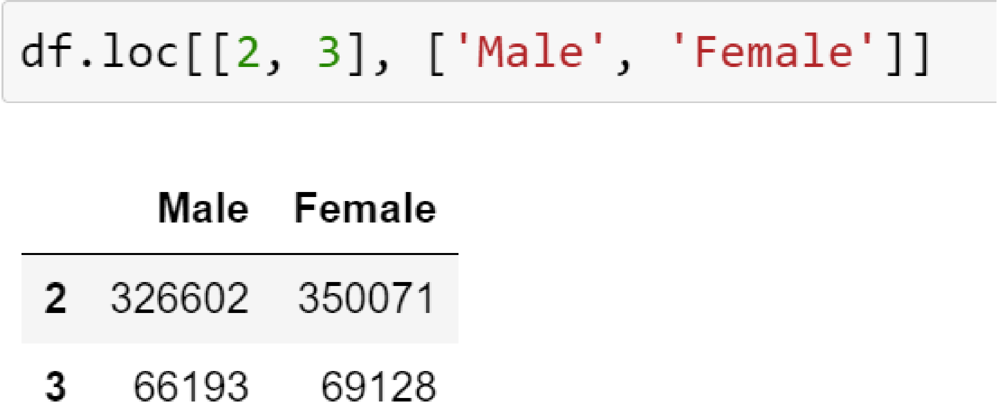
```
df.loc[1, ['Male', 'Female']]
```

```
Male      260358
Female    283727
Name: 1, dtype: object
```



```
df.loc[[2, 3], 'Household']
```

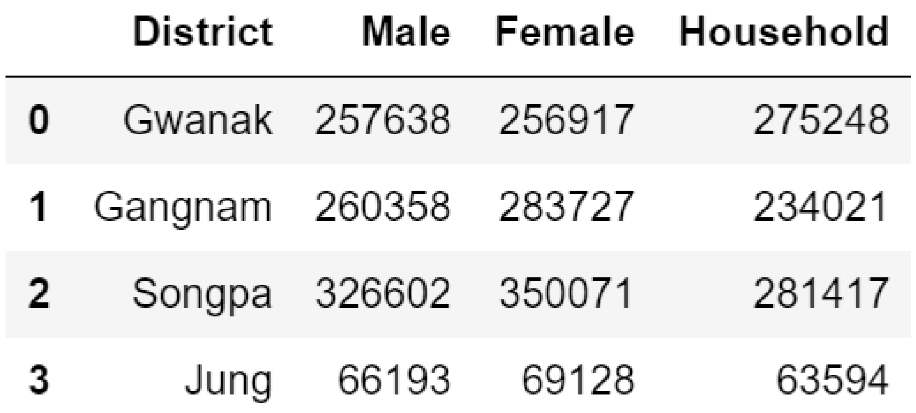
```
2    281417
3     63594
Name: Household, dtype: int64
```



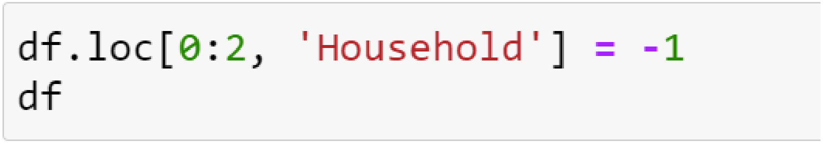
```
df.loc[[2, 3], ['Male', 'Female']]
```

	Male	Female
2	326602	350071
3	66193	69128

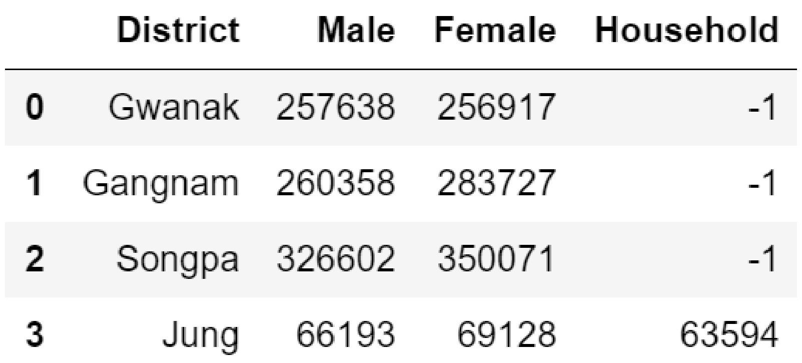
Changing Values



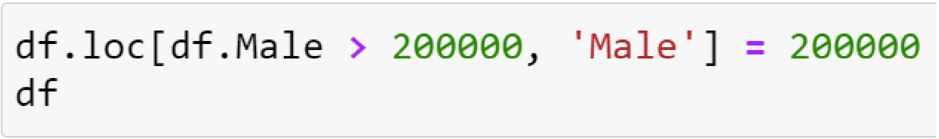
	District	Male	Female	Household
0	Gwanak	257638	256917	275248
1	Gangnam	260358	283727	234021
2	Songpa	326602	350071	281417
3	Jung	66193	69128	63594



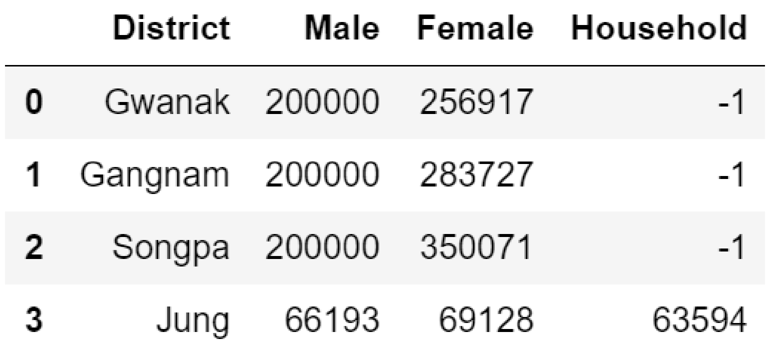
```
df.loc[0:2, 'Household'] = -1
df
```



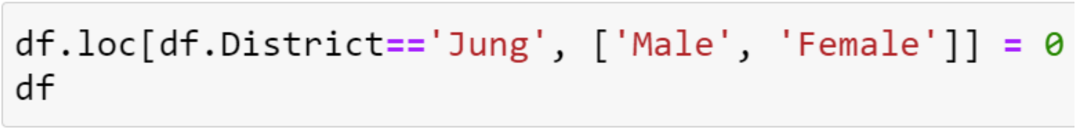
	District	Male	Female	Household
0	Gwanak	257638	256917	-1
1	Gangnam	260358	283727	-1
2	Songpa	326602	350071	-1
3	Jung	66193	69128	63594



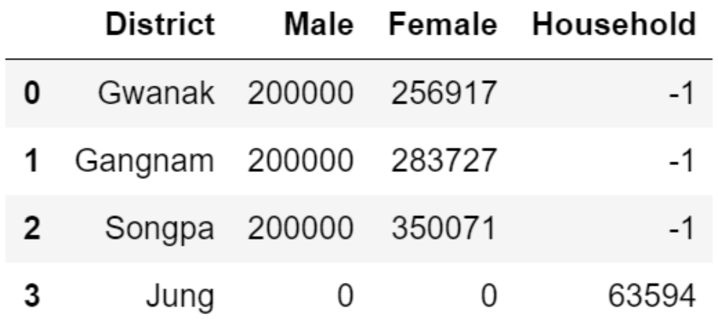
```
df.loc[df.Male > 200000, 'Male'] = 200000
df
```



	District	Male	Female	Household
0	Gwanak	200000	256917	-1
1	Gangnam	200000	283727	-1
2	Songpa	200000	350071	-1
3	Jung	66193	69128	63594



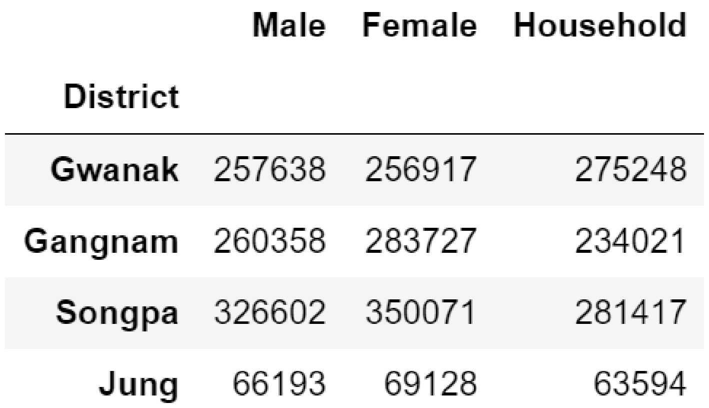
```
df.loc[df.District=='Jung', ['Male', 'Female']] = 0
df
```



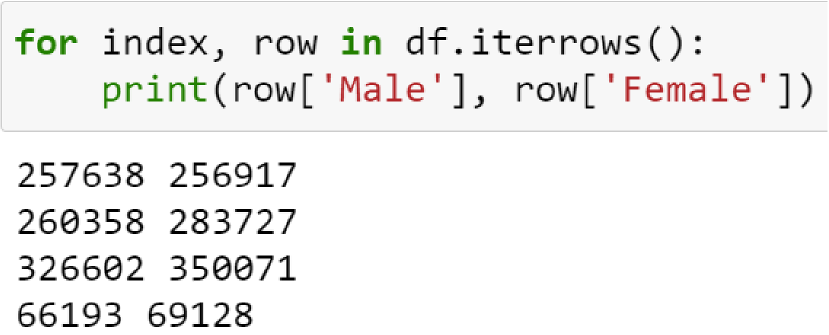
	District	Male	Female	Household
0	Gwanak	200000	256917	-1
1	Gangnam	200000	283727	-1
2	Songpa	200000	350071	-1
3	Jung	0	0	63594

Iteration over Rows

- *df*.`iterrows()`
  - Iterate over rows of DataFrame as (index, Series) pairs

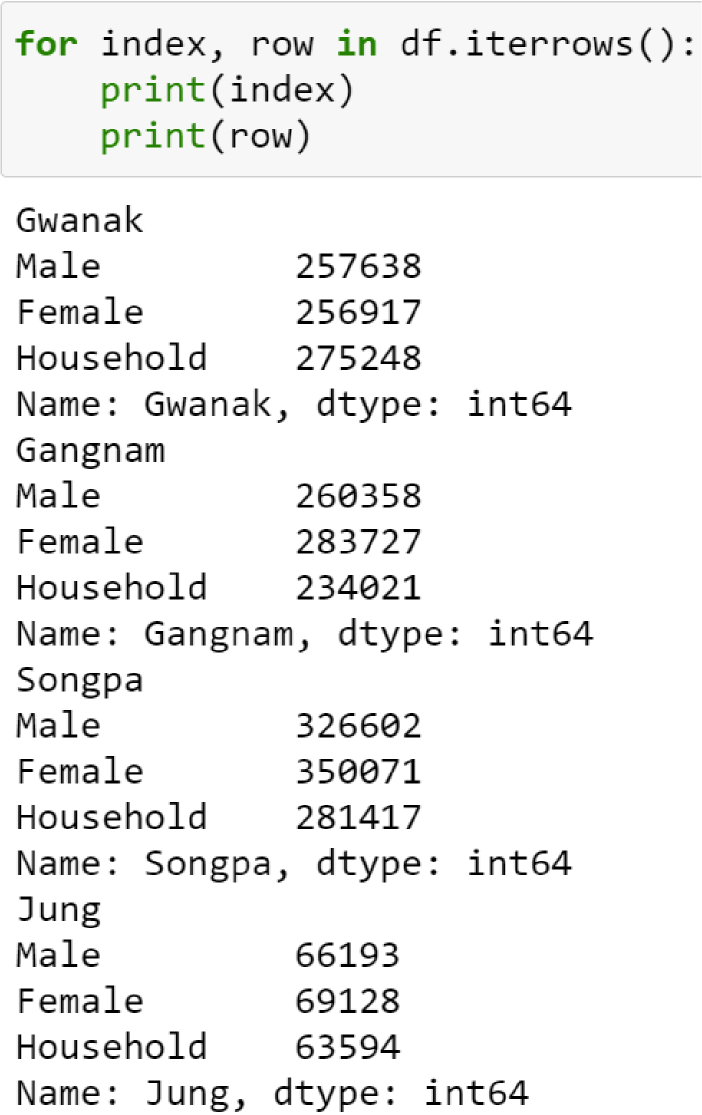


	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



```python
for index, row in df.iterrows():
    print(row['Male'], row['Female'])
```

```
257638 256917
260358 283727
326602 350071
66193 69128
```

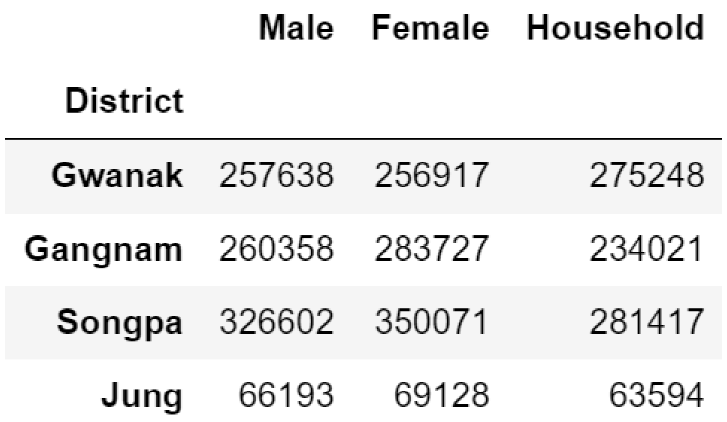


```python
for index, row in df.iterrows():
    print(index)
    print(row)
```

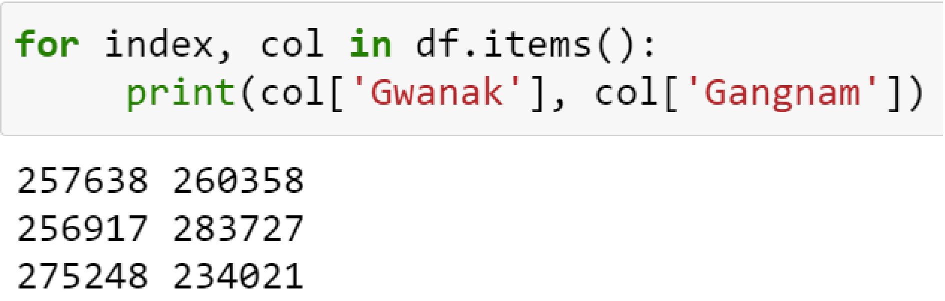
```
Gwanak
Male         257638
Female       256917
Household    275248
Name: Gwanak, dtype: int64
Gangnam
Male         260358
Female       283727
Household    234021
Name: Gangnam, dtype: int64
Songpa
Male         326602
Female       350071
Household    281417
Name: Songpa, dtype: int64
Jung
Male          66193
Female        69128
Household     63594
Name: Jung, dtype: int64
```

Iteration over Columns

- *df*.items()
  - Iterate over DataFrame columns as (index, Series) pairs

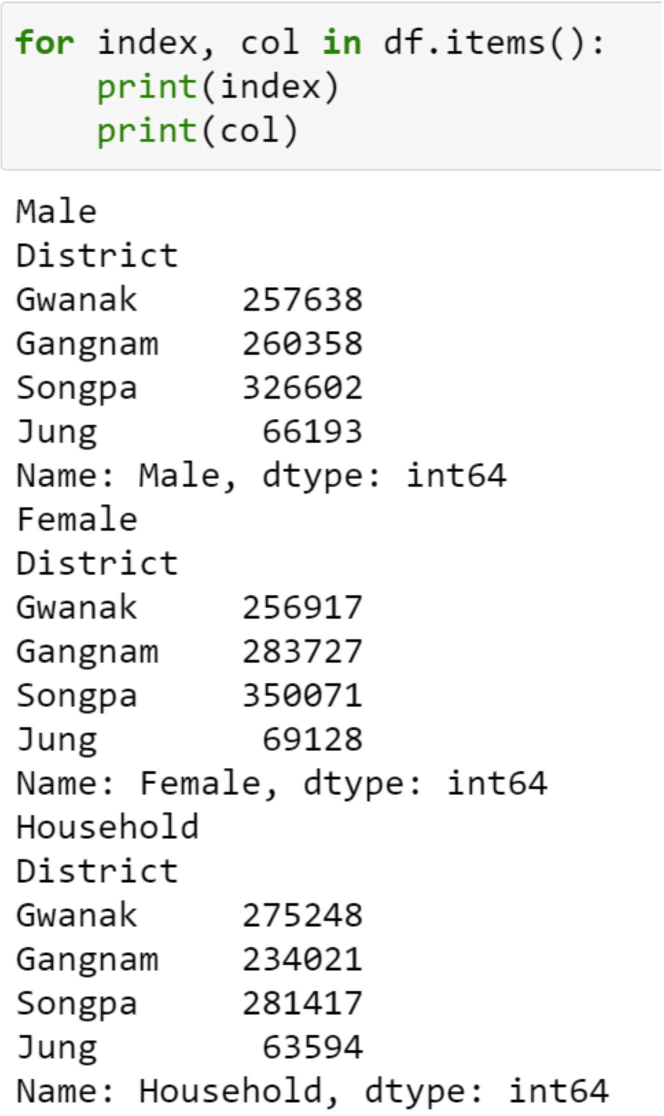


	Male	Female	Household
District			
Gwanak	257638	256917	275248
Gangnam	260358	283727	234021
Songpa	326602	350071	281417
Jung	66193	69128	63594



```python
for index, col in df.items():
    print(col['Gwanak'], col['Gangnam'])
```

```
257638 260358
256917 283727
275248 234021
```



```python
for index, col in df.items():
    print(index)
    print(col)
```

```
Male
District
Gwanak     257638
Gangnam    260358
Songpa     326602
Jung        66193
Name: Male, dtype: int64
Female
District
Gwanak     256917
Gangnam    283727
Songpa     350071
Jung        69128
Name: Female, dtype: int64
Household
District
Gwanak     275248
Gangnam    234021
Songpa     281417
Jung        63594
Name: Household, dtype: int64
```

Summary

- Selecting rows or columns

Operation	Syntax	Result
Select column	`df['Col'], df.Col, df[['Col1','Col2']]`	Series
Slice rows	`df[0:3], df[0:10:2], df[-5:]`	DataFrame
Select row by integer location	`df.iloc[1], df.iloc[0:3], df.iloc[[2,4]]`	Series or DataFrame

Select row by label	`df.loc['R0'], df.loc['R0':'R3'],`
	`df.loc[['R2','R4']]`
		Series orDataFrame

Select rows by Boolean vector	`df[df.Col1 > 10], df[df.Col1.isin([1,2])]`	DataFrame

- Deleting rows or columns
  - `df.drop(0), df.drop([0,2]), df.drop(['Col1','Col2'],axis=1)`

Pandas I/O

I/Os for Pandas DataFrame

- A collection of convenient I/O functions supporting various file formats

```
to_csv()     to_excel()     to_hdf()     to_sql()     to_json()     to_html()
read_csv()   read_excel()   read_hdf()   read_sql()   read_json()   read_html()
```

- (cf.) HDF (Hierarchical Data Format): Standardized file format for scientific data

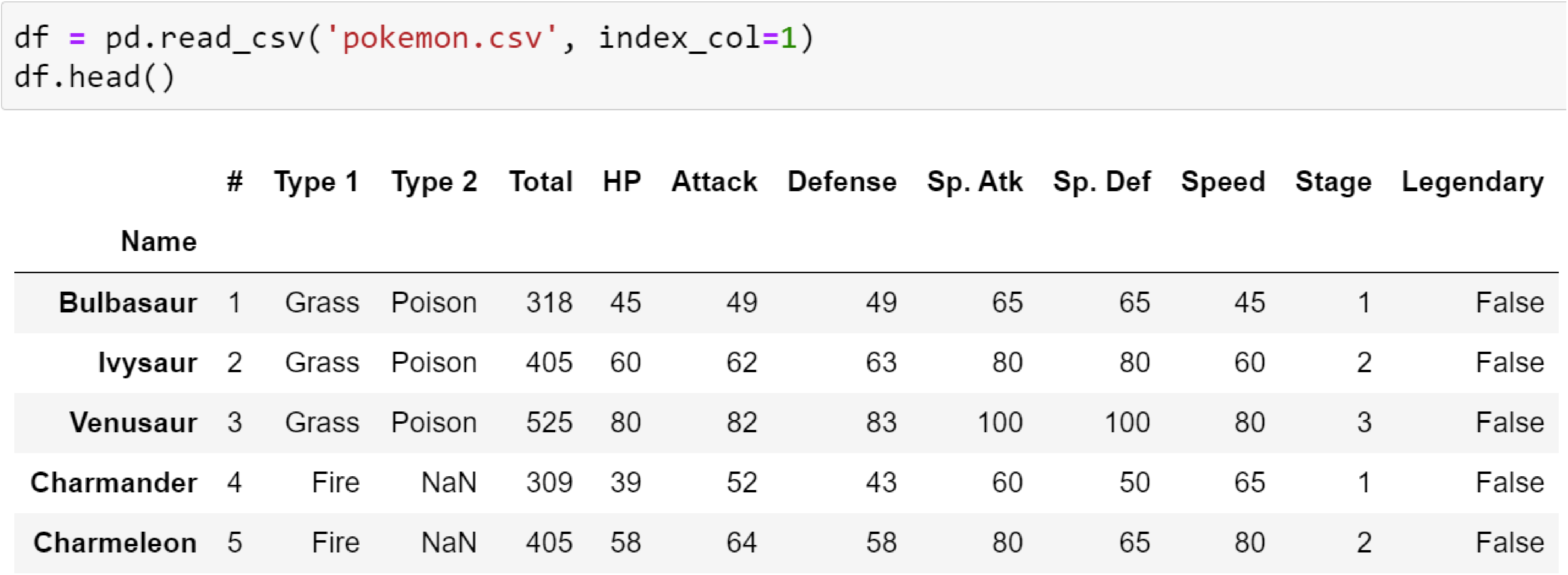
- From CSV file to Pandas DataFrame: *pd*.read_csv(*path*)

- From Pandas DataFrame to CSV file: *pd*.to_csv(*path*)

pandas.read_csv()

- *pd*.read_csv(*filepath, sep=',', header='infer', names=None, index_col=None, encoding='utf-8', skiprows=0, thousands=None, …*)
  - Read a comma-separated values (csv) file
  - *filepath*: any valid string path. The string could be a URL.
  - *sep* (or *delimiter*): delimiter to use
  - *header*: row number(s) to use as the column names
  - *names*: list of column names to use
  - *index_col*: column(s) to use as the row labels of the Data Frame
  - *encoding*: encoding to use
  - *skiprows*: line numbers to skip or number of lines to skip at the start of the file
  - *thousands*: thousands separator

Reading a CSV File



```
df = pd.read_csv('pokemon.csv', index_col=1)
df.head()
```

	#	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Stage	Legendary
Name												
Bulbasaur	1	Grass	Poison	318	45	49	49	65	65	45	1	False
Ivysaur	2	Grass	Poison	405	60	62	63	80	80	60	2	False
Venusaur	3	Grass	Poison	525	80	82	83	100	100	80	3	False
Charmander	4	Fire	NaN	309	39	52	43	60	50	65	1	False
Charmeleon	5	Fire	NaN	405	58	64	58	80	65	80	2	False

DataFrame.to_csv()

- *df*.`to_csv`(*filepath*, *sep*=',', *columns*=None, *header*=True, *index*=True, *encoding*=None, ...)
  - Write DataFrame to a comma-separated values (csv) file
  - *filepath*: any valid string path.
  - *sep* (or *delimiter*): delimiter to use
  - *columns*: columns to write
  - *header*: write out the column names
  - *index*: write row names
  - *encoding*: encoding to use (default: 'utf-8')

```
>>> df.to_csv('mydf.csv', sep='\t')
>>> df.to_csv('dataset.csv', sep='\t', encoding='utf-8')
```