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

Systems Software &
Architecture Lab.

Seoul National University

Jan. 11 – 15, 2021

Python for Data Analytics

Pandas II

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

Arithmetic Operations

Arithmetic (1)

```
df1 = pd.DataFrame(np.random.randint(0,10,size=(4,2)),
                   index=list('abcd'))
```

```
df2 = pd.DataFrame(np.random.randint(0,10,size=(4,2)),
                   index=list('abde'))
```

	0	1
a	8	1
b	9	3
c	1	7
d	9	4

df1

	0	1
a	9	8
b	9	0
d	2	0
e	4	7

df2

```
df1 + df2
```

	0	1
a	17.0	9.0
b	18.0	3.0
c	NaN	NaN
d	11.0	4.0
e	NaN	NaN

```
df1 - df2
```

	0	1
a	-1.0	-7.0
b	0.0	3.0
c	NaN	NaN
d	7.0	4.0
e	NaN	NaN

```
df1 * df2
```

	0	1
a	72.0	8.0
b	81.0	0.0
c	NaN	NaN
d	18.0	0.0
e	NaN	NaN

```
df1 / df2
```

	0	1
a	0.888889	0.125
b	1.000000	inf
c	NaN	NaN
d	4.500000	inf
e	NaN	NaN

```
df1 / 10
```

	0	1
a	0.8	0.1
b	0.9	0.3
c	0.1	0.7
d	0.9	0.4

```
df2 + 0.5
```

	0	1
a	9.5	8.5
b	9.5	0.5
d	2.5	0.5
e	4.5	7.5

Arithmetic (2)

- Alignment is performed on both rows and columns
- The result will be all NANs if no column or row labels are in common

```
df1 = pd.DataFrame(np.arange(9.0).reshape((3,3)),
                   columns=list('bcd'), index=list('xyz'))
df2 = pd.DataFrame(np.arange(12.0).reshape((4,3)),
                   columns=list('bde'), index=list('wxyu'))
df1 + df2
```

	b	c	d
x	0.0	1.0	2.0
y	3.0	4.0	5.0
z	6.0	7.0	8.0

+

	b	d	e
w	0.0	1.0	2.0
x	3.0	4.0	5.0
y	6.0	7.0	8.0
u	9.0	10.0	11.0

=

	b	c	d	e
u	NaN	NaN	NaN	NaN
w	NaN	NaN	NaN	NaN
x	3.0	NaN	6.0	NaN
y	9.0	NaN	12.0	NaN
z	NaN	NaN	NaN	NaN

```
df1 = pd.DataFrame(np.arange(4.0).reshape((2,2)),
                   columns=list('ab'), index=list('mn'))
df2 = pd.DataFrame(np.arange(4.0).reshape((2,2)),
                   columns=list('cd'), index=list('op'))
df1 + df2
```

	a	b
m	0.0	1.0
n	2.0	3.0

+

	c	d
o	0.0	1.0
p	2.0	3.0

=

	a	b	c	d
m	NaN	NaN	NaN	NaN
n	NaN	NaN	NaN	NaN
o	NaN	NaN	NaN	NaN
p	NaN	NaN	NaN	NaN

Arithmetic Methods

Operation	Meaning
`df1.add(df2)`	`df1 + df2`
`df1.sub(df2)`	`df1 - df2`
`df1.div(df2)`	`df1 / df2`
`df1.floordiv(df2)`	`df1 // df2`
`df1.mul(df2)`	`df1 * df2`
`df1.pow(df2)`	`df1 ** df2`
`df1.radd(df2)`	`df2 + df1`
`df1.rsub(df2)`	`df2 – df1`
`df1.rdiv(df2)`	`df2 / df1`
`df1.rfloordiv(df2)`	`df2 // df1`
`df1.rmul(df2)`	`df2 * df1`
`df1.rpow(df2)`	`df2 ** df1`

- *fill_value* can be specified
  - Existing missing values and any new elements is set to this value
  - If data in both corresponding DataFrame locations is missing, the result will be NaN

```
df1.add(df2, fill_value=0)
```

	b	c	d
x	0.0	1.0	2.0
y	3.0	4.0	5.0
z	6.0	7.0	8.0

+

	b	d	e
w	0.0	1.0	2.0
x	3.0	4.0	5.0
y	6.0	7.0	8.0
u	9.0	10.0	11.0

=

	b	c	d	e
u	9.0	NaN	10.0	11.0
w	0.0	NaN	1.0	2.0
x	3.0	1.0	6.0	5.0
y	9.0	4.0	12.0	8.0
z	6.0	7.0	8.0	NaN

Applying NumPy Universal Function

- A universal function (ufunc) operates on ndarrays in an element-by-element fashion
  - `np.fabs()`, `np.exp()`, `np.sqrt()`, `np.log()`, ...

	a	b	c
R0	0.0	1.0	2.0
R1	3.0	4.0	5.0
R2	6.0	7.0	8.0
R3	9.0	10.0	11.0

```
np.sqrt(df)
```

	a	b	c
R0	0.000000	1.000000	1.414214
R1	1.732051	2.000000	2.236068
R2	2.449490	2.645751	2.828427
R3	3.000000	3.162278	3.316625

```
df.apply(np.exp)
```

	a	b	c
R0	1.000000	2.718282	7.389056
R1	20.085537	54.598150	148.413159
R2	403.428793	1096.633158	2980.957987
R3	8103.083928	22026.465795	59874.141715

Applying a Function to Row/Column

- *df*.apply(*func*, *axis=0*, ...)
  - Apply a function along an axis of the DataFrame
  - *axis=0* or *axis='index'*: apply *func* to each column (default)
  - *axis=1* or *axis='columns'*: apply *func* to each row

	a	b	c
R0	0.0	1.0	2.0
R1	3.0	4.0	5.0
R2	6.0	7.0	8.0
R3	9.0	10.0	11.0

```
df.apply(np.sum)
```

```
a     18.0
b     22.0
c     26.0
dtype: float64
```

```
df.apply(np.average, axis=1)
```

```
R0     1.0
R1     4.0
R2     7.0
R3    10.0
dtype: float64
```

Common Statistical Functions

Method	Description
`count()`	Number of non-null observations
`sum()`	Sum of values
`mean()`	Mean of values
`median()`	Arithmetic median of values
`min()`	Minimum
`max()`	Maximum
`std()`	Bessel-corrected sample standard deviation
`var()`	Unbiased variance
`skew()`	Sample skewness (3rd moment)
`kurt()`	Sample kurtosis (4th moment)
`quantile()`	Sample quantile (value at %)
`mad()`	Mean absolute deviation from mean value
`cov()`	Unbiased covariance
`corr()`	Correlation

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

```
df.Household.mean()
```

```
213570.0
```

```
df.Male.max()
```

```
326602
```

```
df.loc['Songpa'].sum()
```

```
958090
```

```
df['Female'].std()
```

```
120431.30086035219
```

Sorting DataFrame by Index

- *df*.sort_index(*axis=0, level=None, ascending=True, ...)*
  - Sort by labels along an axis
  - *axis=0* or *axis= 'index'*: sort along the rows (default)
  - *axis=1* or *axis='columns'*: sort along the columns
  - *level*: sort on values in specified index level

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

```
df.sort_index(axis=0, ascending=False)
```

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

```
df.sort_index(axis=0, level=1)
```

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

Sorting DataFrame by Values

- *df*.`sort_values`(*by, axis=0, ascending=True, ...)*
  - Sort by the values along either axis
  - *by:* name or list of names to sort by
  - *axis:* axis to be sorted

```
df.sort_values('Male')
```

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

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

```
df.sort_values('Songpa', axis=1)
```

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

Ranking DataFrame

- *df.*`rank(`*axis=0, method='average', ascending=True, ...)*
  - Compute numerical data ranks (1 through n) along axis
  - *axis:* index to direct ranking
  - *method:* how to rank the records that have the same value (tie-breaking rule) -- 'average', 'min', 'max', 'first' or 'dense'

```
df['average'] =
   df.score.rank(method='average',
                 ascending=False
```

	name	score	average	min	max	first	dense
0	kim	100	1.0	1.0	1.0	1.0	1.0
1	lee	80	6.0	5.0	7.0	5.0	4.0
2	park	95	2.5	2.0	3.0	2.0	2.0
3	choi	80	6.0	5.0	7.0	6.0	4.0
4	seo	80	6.0	5.0	7.0	7.0	4.0
5	hong	90	4.0	4.0	4.0	4.0	3.0
6	min	95	2.5	2.0	3.0	3.0	2.0

Group Aggregation

Example DataFrame Object

```
df = pd.DataFrame({'A':['foo','bar','foo','bar','foo','bar','foo','foo'],
                   'B':['one','one','two','three','two','two','one','three'],
                   'C':np.random.randn(8), \
                   'D':np.random.randn(8)})
df
```

Two categorical values: A, B
Two numerical values: C, D

	A	B	C	D
0	foo	one	-1.307901	1.174486
1	bar	one	-0.409657	0.872421
2	foo	two	-1.278086	0.028901
3	bar	three	0.315299	-2.273053
4	foo	two	-0.147411	-0.892378
5	bar	two	0.218316	-0.693276
6	foo	one	-1.024522	-0.358686
7	foo	three	1.996648	-0.273449

```
df[df.A == 'foo']
```

	A	B	C	D
0	foo	one	-1.307901	1.174486
2	foo	two	-1.278086	0.028901
4	foo	two	-0.147411	-0.892378
6	foo	one	-1.024522	-0.358686
7	foo	three	1.996648	-0.273449

```
df[df.A == 'bar']
```

	A	B	C	D
1	bar	one	-0.409657	0.872421
3	bar	three	0.315299	-2.273053
5	bar	two	0.218316	-0.693276

Groupby and Aggregation

- *df*.groupby(*by*, *axis=0*, ...)
  - Used to group large amounts of data and compute operations on these groups
  - *by*: label, function, a list of labels, ... (Used to determine the groups)
  - *axis*: 0 or '*index*' for rows, 1 or '*columns*' for columns (default: 0)

- Aggregation stat functions after grouping
  - `mean()`, `sum()`, `median()`, `var()`, etc.

```
g = df.groupby('A')
g.mean()
```

A	C	D
bar	0.041319	-0.697969
foo	-0.352255	-0.064225

```
g.corr()
```

A		C	D
bar	C	1.000000	-0.920023
	D	-0.920023	1.000000
foo	C	1.000000	-0.404475
	D	-0.404475	1.000000

Iterating over Groups

- Get a group's contents

```
g.get_group('bar')
```

	B	C	D
1	one	-0.409657	0.872421
3	three	0.315299	-2.273053
5	two	0.218316	-0.693276

- Printing the groups

```
for key, items in g:
    print('%s:' % key)
    print(g.get_group(key))
```

```
bar:
        B         C         D
1     one -0.409657  0.872421
3   three  0.315299 -2.273053
5     two  0.218316 -0.693276
foo:
        B         C         D
0     one -1.307901  1.174486
2     two -1.278086  0.028901
4     two -0.147411 -0.892378
6     one -1.024522 -0.358686
7   three  1.996648 -0.273449
```

Describing a Group

```
g.describe()
```

	C							
	count	mean	std	min	25%	50%	75%	max
A								
bar	3.0	0.041319	0.393556	-0.409657	-0.095670	0.218316	0.266807	0.315299
foo	5.0	-0.352255	1.394782	-1.307901	-1.278086	-1.024522	-0.147411	1.996648

	D							
	count	mean	std	min	25%	50%	75%	max
	3.0	-0.697969	1.572742	-2.273053	-1.483164	-0.693276	0.089573	0.872421
	5.0	-0.064225	0.768016	-0.892378	-0.358686	-0.273449	0.028901	1.174486

Grouping by Multiple Columns

```
gm = df.groupby(['A','B'])
gm.mean()
```

		C	D
A	B		
bar	one	-0.409657	0.872421
	three	0.315299	-2.273053
	two	0.218316	-0.693276
foo	one	-1.166211	0.407900
	three	1.996648	-0.273449
	two	-0.712749	-0.431739

```
gm.mean().unstack()
```

	C			D		
B	one	three	two	one	three	two
A						
bar	-0.409657	0.315299	0.218316	0.872421	-2.273053	-0.693276
foo	-1.166211	1.996648	-0.712749	0.407900	-0.273449	-0.431739

Grouping by List/Dict

```
years = [2019,2020,2020,2018,2018,2020,2020,2018]
df[['C','D']].groupby(years).mean()
```

	C	D
2018	0.721512	-1.146293
2019	-1.307901	1.174486
2020	-0.623487	-0.037660

	A	B	C	D
0	foo	one	-1.307901	1.174486
1	bar	one	-0.409657	0.872421
2	foo	two	-1.278086	0.028901
3	bar	three	0.315299	-2.273053
4	foo	two	-0.147411	-0.892378
5	bar	two	0.218316	-0.693276
6	foo	one	-1.024522	-0.358686
7	foo	three	1.996648	-0.273449

```
d = {'세대수':'세대수','한국인남자':'남자',
     '한국인여자':'여자','외국인남자':'남자',
     '외국인여자':'여자'}
df.groupby(d, axis=1).sum()
```

기간	자치구	세대수	한국인남자	한국인여자	외국인남자	외국인여자
2020.1/4	중구	63045	61839	64336	4930	5364
	관악구	270760	250743	248631	8239	9049
	송파구	279301	325859	348236	3199	3589
2020.2/4	중구	63354	61697	64395	4848	5090
	관악구	273715	250829	248911	7911	8667
	송파구	280135	324317	347195	3066	3489
2020.3/4	중구	63594	61526	64274	4667	4854
	관악구	275248	250084	248490	7554	8427
	송파구	281417	323646	346685	2956	3386

		남자	세대수	여자
기간	자치구			
2020.1/4	중구	66769	63045	69700
	관악구	258982	270760	257680
	송파구	329058	279301	351825
2020.2/4	중구	66545	63354	69485
	관악구	258740	273715	257578
	송파구	327383	280135	350684
2020.3/4	중구	66193	63594	69128
	관악구	257638	275248	256917
	송파구	326602	281417	350071

Hierarchical Indexing

Reading a Sample Dataset (1)

- `seoul2020.txt`: 서울시 주민등록인구 (구별) 통계 (2020년 1-3분기)
  - https://data.seoul.go.kr/dataList/419/S/2/datasetView.do

기간	자치구	세대	인구	인구	인구	인구	인구	인구	인구	인구	인구	세대당인구	65세이상고령자						
기간	자치구	세대	합계	합계	합계	한국인	한국인	한국인	등록외국인	등록외국인	등록외국인	세대당인구	65세이상고령자						
기간	자치구	세대	계	남자	여자	계	남자	여자	계	남자	여자	세대당인구	65세이상고령자						
2020.1/4		합계	4,354,006		10,013,781		4,874,995		5,138,786		9,733,655		4,742,217	4,991,438	280,126	132,778	147,348	2.24	1,518,239
2020.1/4		종로구	74,151	161,984	78,271	83,713	151,217	73,704	77,513	10,767	4,567	6,200	2.04	28,073					
2020.1/4		중구	63,045	136,469	66,769	69,700	126,175	61,839	64,336	10,294	4,930	5,364	2	23,794					
2020.1/4		용산구	110,895	246,165	119,961	126,204	229,579	110,667	118,912	16,586	9,294	7,292	2.07	39,439					
2020.1/4		성동구	135,643	307,193	149,891	157,302	299,042	146,300	152,742	8,151	3,591	4,560	2.2	44,728					
2020.1/4		광진구	165,287	365,990	176,226	189,764	350,417	169,568	180,849	15,573	6,658	8,915	2.12	48,989					
2020.1/4		동대문구	165,279	362,793	178,202	184,591	346,156	171,896	174,260	16,637	6,306	10,331	2.09	60,367					
2020.1/4		중랑구	182,220	400,678	198,122	202,556	395,619	196,076	199,543	5,059	2,046	3,013	2.17	66,764					
2020.1/4		성북구	193,801	454,532	218,561	235,971	442,494	213,926	228,568	12,038	4,635	7,403	2.28	72,172					
2020.1/4		강북구	144,805	316,750	154,141	162,609	312,985	152,747	160,238	3,765	1,394	2,371	2.16	61,660					
2020.1/4		도봉구	138,595	333,495	162,774	170,721	331,238	161,879	169,359	2,257	895	1,362	2.39	60,023					
2020.1/4		노원구	217,148	535,495	258,696	276,799	531,037	256,726	274,311	4,458	1,970	2,488	2.45	82,682					
2020.1/4		은평구	208,209	482,509	231,953	250,556	478,019	230,147	247,872	4,490	1,806	2,684	2.3	82,245					
2020.1/4		서대문구	142,109	325,875	154,502	171,373	312,642	150,051	162,591	13,233	4,451	8,782	2.2	53,038					
2020.1/4		마포구	176,133	386,086	181,204	204,882	374,570	176,943	197,627	11,516	4,261	7,255	2.13	53,283					
2020.1/4		양천구	177,436	460,532	226,109	234,423	456,339	224,241	232,098	4,193	1,868	2,325	2.57	62,761					
2020.1/4		강서구	263,645	595,703	288,134	307,569	589,302	285,085	304,217	6,401	3,049	3,352	2.24	85,992					
2020.1/4		구로구	177,275	438,308	218,970	219,338	405,837	200,558	205,279	32,471	18,412	14,059	2.29	67,432					
2020.1/4		금천구	111,542	251,370	128,643	122,727	232,583	118,044	114,539	18,787	10,599	8,188	2.09	38,508					
2020.1/4		영등포구	177,743	404,766	202,617	202,149	371,903	184,316	187,587	32,863	18,301	14,562	2.09	59,373					
2020.1/4		동작구	181,761	407,802	196,943	210,859	395,014	191,272	203,742	12,788	5,671	7,117	2.17	63,378					
2020.1/4		관악구	270,760	516,662	258,982	257,680	499,374	250,743	248,631	17,288	8,239	9,049	1.84	76,664					
2020.1/4		서초구	173,580	434,801	207,877	226,924	430,568	205,787	224,781	4,233	2,090	2,143	2.48	58,332					
2020.1/4		강남구	233,624	549,898	263,163	286,735	544,804	260,654	284,150	5,094	2,509	2,585	2.33	72,602					
2020.1/4		송파구	279,301	680,883	329,058	351,825	674,095	325,859	348,236	6,788	3,199	3,589	2.41	89,539					
2020.1/4		강동구	190,019	457,042	225,226	231,816	452,646	223,189	229,457	4,396	2,037	2,359	2.38	66,401					
2020.2/4		합계	4,384,076		9,985,652		4,859,501		5,126,151		9,720,846		4,732,275	4,988,571	264,806	127,226	137,580	2.22	1,534,957
2020.2/4		종로구	74,497	160,520	77,745	82,775	150,383	73,288	77,095	10,137	4,457	5,680	2.02	28,203					
2020.2/4		중구	63,354	136,030	66,545	69,485	126,092	61,697	64,395	9,938	4,848	5,090	1.99	24,035					
2020.2/4		용산구	111,586	245,362	119,494	125,868	229,431	110,527	118,904	15,931	8,967	6,964	2.06	39,650					

Reading a Sample Dataset (2)

- What's wrong?

```
import numpy as np
import pandas as pd
df = pd.read_csv('seoul2020.txt', sep='\t', thousands=',')
df.head()
```

	기간	자치구	세대	인구	인구.1	인구.2	인구.3	인구.4	인구.5	인
0	기간	자치구	세대	합계	합계	합계	한국인	한국인	한국인	등록외
1	기간	자치구	세대	계	남자	여자	계	남자	여자	
2	2020.1/4	합계	4,354,006	10,013,781	4,874,995	5,138,786	9,733,655	4,742,217	4,991,438	280,
3	2020.1/4	종로구	74,151	161,984	78,271	83,713	151,217	73,704	77,513	10,
4	2020.1/4	중구	63,045	136,469	66,769	69,700	126,175	61,839	64,336	10,

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 80 entries, 0 to 79
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   기간        80 non-null     object
 1   자치구       80 non-null     object
 2   세대        80 non-null     object
 3   인구        80 non-null     object
 4   인구.1      80 non-null     object
 5   인구.2      80 non-null     object
 6   인구.3      80 non-null     object
 7   인구.4      80 non-null     object
 8   인구.5      80 non-null     object
 9   인구.6      80 non-null     object
 10  인구.7      80 non-null     object
 11  인구.8      80 non-null     object
 12  세대당인구     80 non-null     object
 13  65세이상고령자  80 non-null     object
dtypes: object(14)
memory usage: 8.9+ KB
```

Reading a Sample Dataset (3)

- Ignoring first two rows

```
df = pd.read_csv('seoul2020.txt', sep='\t', thousands=',', skiprows=2)
df.head()
```

	기간	자치구	세대	계	남자	여자	계.1	남자.1	여자.1	
0	2020.1/4	합계	4354006	10013781	4874995	5138786	9733655	4742217	4991438	28
1	2020.1/4	종로구	74151	161984	78271	83713	151217	73704	77513	1
2	2020.1/4	중구	63045	136469	66769	69700	126175	61839	64336	1
3	2020.1/4	용산구	110895	246165	119961	126204	229579	110667	118912	1
4	2020.1/4	성동구	135643	307193	149891	157302	299042	146300	152742	

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9 entries, 4 to 78
Data columns (total 14 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   기간         9 non-null      object
 1   자치구        9 non-null      object
 2   세대         9 non-null      int64
 3   인구         9 non-null      int64
 4   인구.1       9 non-null      int64
 5   인구.2       9 non-null      int64
 6   인구.3       9 non-null      int64
 7   인구.4       9 non-null      int64
 8   인구.5       9 non-null      int64
 9   인구.6       9 non-null      int64
 10  인구.7       9 non-null      int64
 11  인구.8       9 non-null      int64
 12  세대당인구      9 non-null      object
 13  65세이상고령자  9 non-null      int64
dtypes: int64(11), object(3)
memory usage: 1.1+ KB
```

<Another way of changing types>

```
df = pd.read_csv('seoul2020.txt', sep='\t')
df = df[df.자치구.isin(['관악구','송파구','중구'])]
for col in df.columns[2:]:
    if (col != '세대당인구'):
        df[col] = df[col].apply(lambda x: x.replace(',','')).astype(np.int64)
```

Using Hierarchical Indexes

```
df = pd.read_csv('seoul2020.txt', thousands=',', sep='\t', skiprows=2)
df = df[df.자치구.isin(['관악구','송파구','중구'])]
df = df.set_index(['기간','자치구'])
df = df[['세대','남자.1','여자.1','남자.2','여자.2']]
df.columns=[['세대수','한국인','한국인','외국인','외국인'],
            ['세대',  '남자', '여자',  '남자',  '여자' ]]
df.columns.names = ['총계', '수']
df
```

	총계	세대수	한국인		외국인	
	수	세대	남자	여자	남자	여자
기간	자치구					
2020.1/4	중구	63045	61839	64336	4930	5364
	관악구	270760	250743	248631	8239	9049
	송파구	279301	325859	348236	3199	3589
2020.2/4	중구	63354	61697	64395	4848	5090
	관악구	273715	250829	248911	7911	8667
	송파구	280135	324317	347195	3066	3489
2020.3/4	중구	63594	61526	64274	4667	4854
	관악구	275248	250084	248490	7554	8427
	송파구	281417	323646	346685	2956	3386

Using Primary Row Index

```
df['2020.2/4':'2020.3/4']
```

	총계	세대수	한국인		외국인	
	수	세대	남자	여자	남자	여자
기간	자치구					
2020.2/4	중구	63354	61697	64395	4848	5090
	관악구	273715	250829	248911	7911	8667
	송파구	280135	324317	347195	3066	3489
2020.3/4	중구	63594	61526	64274	4667	4854
	관악구	275248	250084	248490	7554	8427
	송파구	281417	323646	346685	2956	3386

```
df.loc[['2020.1/4','2020.3/4']]
```

	총계	세대수	한국인		외국인	
	수	세대	남자	여자	남자	여자
기간	자치구					
2020.1/4	중구	63045	61839	64336	4930	5364
	관악구	270760	250743	248631	8239	9049
	송파구	279301	325859	348236	3199	3589
2020.3/4	중구	63594	61526	64274	4667	4854
	관악구	275248	250084	248490	7554	8427
	송파구	281417	323646	346685	2956	3386

Using Secondary Row Index

```
df.loc[[('2020.1/4','중구'), ('2020.3/4', '관악구')]]
```

		세대수	한국인		외국인	
		세대	남자	여자	남자	여자
기간	자치구					
2020.1/4	중구	63045	66769	69700	4930	5364
2020.3/4	관악구	275248	257638	256917	7554	8427

```
df.loc[(slice(None), '중구'),:]
```

		총계	세대수	한국인		외국인	
		수	세대	남자	여자	남자	여자
기간	자치구						
2020.1/4	중구	63045	61839	64336	4930	5364	
2020.2/4	중구	63354	61697	64395	4848	5090	
2020.3/4	중구	63594	61526	64274	4667	4854	

Using Column Indexes

```
df.loc[(slice(None), '관악구'),
       (slice(None), '여자')]
```

기간	자치구	총계 수 여자	한국인 여자
2020.1/4	관악구	248631	9049
2020.2/4	관악구	248911	8667
2020.3/4	관악구	248490	8427

```
df['한국인']
```

기간	자치구	수	남자	여자
2020.1/4	중구		61839	64336
	관악구		250743	248631
	송파구		325859	348236
2020.2/4	중구		61697	64395
	관악구		250829	248911
	송파구		324317	347195
2020.3/4	중구		61526	64274
	관악구		250084	248490
	송파구		323646	346685

```
df.loc[:, (slice(None), '여자')]
```

기간	자치구	총계 수 여자	한국인 여자	외국인 여자
2020.1/4	중구	64336	5364	
	관악구	248631	9049	
	송파구	348236	3589	
2020.2/4	중구	64395	5090	
	관악구	248911	8667	
	송파구	347195	3489	
2020.3/4	중구	64274	4854	
	관악구	248490	8427	
	송파구	346685	3386	

Unstacking / Stacking

```
df2 = df['한국인']
df2
```

기간	자치구	남자	여자
2020.1/4	중구	66769	69700
	관악구	258982	257680
	송파구	329058	351825
2020.2/4	중구	66545	69485
	관악구	258740	257578
	송파구	327383	350684
2020.3/4	중구	66193	69128
	관악구	257638	256917
	송파구	326602	350071

```
df2.unstack()
```

	남자			여자		
자치구	관악구	송파구	중구	관악구	송파구	중구
기간						
2020.1/4	258982	329058	66769	257680	351825	69700
2020.2/4	258740	327383	66545	257578	350684	69485
2020.3/4	257638	326602	66193	256917	350071	69128

```
df2.unstack().stack()
```

		남자	여자
기간	자치구		
2020.1/4	관악구	258982	257680
	송파구	329058	351825
	중구	66769	69700
2020.2/4	관악구	258740	257578
	송파구	327383	350684
	중구	66545	69485
2020.3/4	관악구	257638	256917
	송파구	326602	350071
	중구	66193	69128

Swapping and Sorting

```
df = df.swaplevel('기간','자치구')
df
```

		세대수	한국인		외국인	
		세대	남자	여자	남자	여자
자치구	기간					
중구	2020.1/4	63045	66769	69700	4930	5364
관악구	2020.1/4	270760	258982	257680	8239	9049
송파구	2020.1/4	279301	329058	351825	3199	3589
중구	2020.2/4	63354	66545	69485	4848	5090
관악구	2020.2/4	273715	258740	257578	7911	8667
송파구	2020.2/4	280135	327383	350684	3066	3489
중구	2020.3/4	63594	66193	69128	4667	4854
관악구	2020.3/4	275248	257638	256917	7554	8427
송파구	2020.3/4	281417	326602	350071	2956	3386

```
df.sort_index(level=0)
```

		세대수	한국인		외국인	
		세대	남자	여자	남자	여자
자치구	기간					
관악구	2020.1/4	270760	258982	257680	8239	9049
	2020.2/4	273715	258740	257578	7911	8667
	2020.3/4	275248	257638	256917	7554	8427
송파구	2020.1/4	279301	329058	351825	3199	3589
	2020.2/4	280135	327383	350684	3066	3489
	2020.3/4	281417	326602	350071	2956	3386
중구	2020.1/4	63045	66769	69700	4930	5364
	2020.2/4	63354	66545	69485	4848	5090
	2020.3/4	63594	66193	69128	4667	4854

Applying Functions

```
df.sum(level='자치구')
```

수	세대수	한국인		외국인	
총계	세대	남자	여자	남자	여자
자치구					
중구	189993	199507	208313	14445	15308
관악구	819723	775360	772175	23704	26143
송파구	840853	983043	1052580	9221	10464

```
df.sum(level='수', axis=1)
```

	수	세대수	한국인	외국인
자치구	기간			
중구	2020.1/4	63045	136469	10294
관악구	2020.1/4	270760	516662	17288
송파구	2020.1/4	279301	680883	6788
중구	2020.2/4	63354	136030	9938
관악구	2020.2/4	273715	516318	16578
송파구	2020.2/4	280135	678067	6555
중구	2020.3/4	63594	135321	9521
관악구	2020.3/4	275248	514555	15981
송파구	2020.3/4	281417	676673	6342

	수	세대수	한국인		외국인	
	총계	세대	남자	여자	남자	여자
자치구	기간					
중구	2020.1/4	63045	66769	69700	4930	5364
관악구	2020.1/4	270760	258982	257680	8239	9049
송파구	2020.1/4	279301	329058	351825	3199	3589
중구	2020.2/4	63354	66545	69485	4848	5090
관악구	2020.2/4	273715	258740	257578	7911	8667
송파구	2020.2/4	280135	327383	350684	3066	3489
중구	2020.3/4	63594	66193	69128	4667	4854
관악구	2020.3/4	275248	257638	256917	7554	8427
송파구	2020.3/4	281417	326602	350071	2956	3386

Combining and Merging

Appending DataFrames (1)

- *df*.append(*other*, *ignore_index=False*, ...)
  - Append rows of *other* to the end of caller, returning a new object
  - *left*: DataFrame or Series, or list of these
  - *ignore_index*: If True, the resulting axis will be labeled 0, 1, …, n-1

```
df1
```

	id	name	country
0	1	Alice	Korea
1	2	Bob	US
2	9	Charlie	UK
3	5	Emily	France

```
df2
```

	id	name	age
0	4	Judy	15
1	7	David	19
2	8	Bill	11

```
df1.append(df2)
```

	id	name	country	age
0	1	Alice	Korea	NaN
1	2	Bob	US	NaN
2	9	Charlie	UK	NaN
3	5	Emily	France	NaN
0	4	Judy	NaN	15.0
1	7	David	NaN	19.0
2	8	Bill	NaN	11.0

```
df1.append([df2, df2],
    ignore_index=True)
```

	id	name	country	age
0	1	Alice	Korea	NaN
1	2	Bob	US	NaN
2	9	Charlie	UK	NaN
3	5	Emily	France	NaN
4	4	Judy	NaN	15.0
5	7	David	NaN	19.0
6	8	Bill	NaN	11.0
7	4	Judy	NaN	15.0
8	7	David	NaN	19.0
9	8	Bill	NaN	11.0

Appending DataFrames (2)

- Appending rows with a Python a dictionary

```
newdata = [{'id':8, 'name':'Jack', 'country':'Japan'},
           {'id':6, 'name':'Kate'}]
df.append(newdata, ignore_index=True)
```

	id	name	country
0	1	Alice	Korea
1	2	Bob	US
2	9	Charlie	UK
3	5	Emily	France
4	8	Jack	Japan
5	6	Kate	NaN

Merging (Joining)

- *pd.*`merge`(*left*, *right*, [*how*], [*on*], [*left_on*], [*right_on*], [*left_index*], [*right_index*], ...)
  - Merge DataFrame objects with database-style join
  - *left*: DataFrame
  - *right*: Object to merge with
  - *how*: join type -- 'left', 'right', 'outer', or 'inner' (default: 'inner')
  - *on*: column to join on (label or list) -- must be found on both DataFrames
  - *left_on* (or *right_on*): column to join on in the left (or right) DataFrame
  - *left_index* (or *right_index*): if True, use the index from the left (or right) DataFrame

```
result = pd.merge(dfx, dfy, on='key')
```

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Columns for Merging

- Merging two DataFrames by their own index

```
pd.merge(dfx, dfy,
         left_index=True, right_index=True)
```

	A	B
W	A0	B0
X	A1	B1
Y	A2	B2
Z	A3	B3

	C	D
W	C0	D0
X	C1	D1
Y	C2	D2
Z	C3	D3

	A	B	C	D
W	A0	B0	C0	D0
X	A1	B1	C1	D1
Y	A2	B2	C2	D2
Z	A3	B3	C3	D3

- If no information is given, use overlapping column names as the keys

```
pd.merge(dfx, dfy)
```

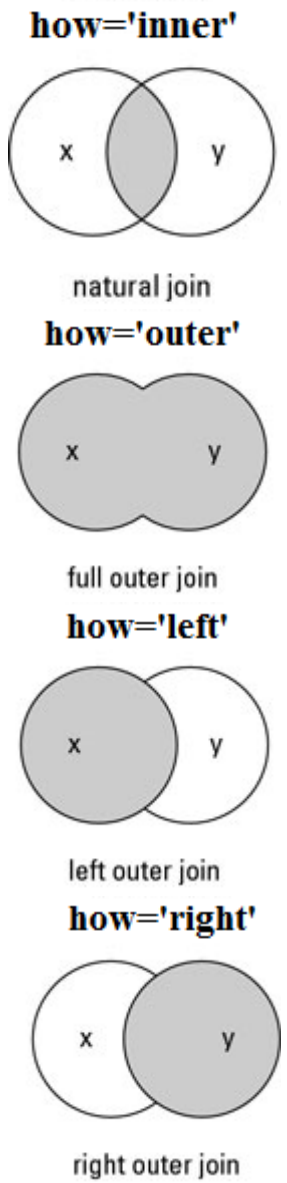
	A	B	X
0	A0	B0	X0
1	A1	B1	X1
2	A2	B2	X2
3	A3	B3	X3

	X	Y	Z
0	X0	Y0	Z0
1	X1	Y1	Z1
2	X2	Y2	Z2
3	X3	Y3	Z3

	A	B	X	Y	Z
0	A0	B0	X0	Y0	Z0
1	A1	B1	X1	Y1	Z1
2	A2	B2	X2	Y2	Z2
3	A3	B3	X3	Y3	Z3

Merging (Joining) Types

- Inner join ('inner') -- *default*
  - Return only the rows in which the left table have matching keys in the right table
- Outer join ('outer')
  - Returns all rows from both tables, join records from the left which have matching keys in the right table.
- Left outer join ('left')
  - Return all rows from the left table, and any rows with matching keys from the right table.
- Right outer join ('right')
  - Return all rows from the right table, and any rows with matching keys from the left table.



Merging (Joining) Example

`pd.merge(df1, df2)`

df1

	id	name
0	1	Alice
1	2	Bob
2	3	Charlie
3	4	David
4	5	Emily

df2

	id	country
0	2	Korea
1	4	US
2	5	UK
3	6	Italy

inner

	id	name	country
0	2	Bob	Korea
1	4	David	US
2	5	Emily	UK

outer

	id	name	country
0	1	Alice	NaN
1	2	Bob	Korea
2	3	Charlie	NaN
3	4	David	US
4	5	Emily	UK
5	6	NaN	Italy

left

	id	name	country
0	1	Alice	NaN
1	2	Bob	Korea
2	3	Charlie	NaN
3	4	David	US
4	5	Emily	UK

right

	id	name	country
0	2	Bob	Korea
1	4	David	US
2	5	Emily	UK
3	6	NaN	Italy

Many-to-One Join

```
dfx
```

	id	name
0	1	Alice
1	2	Bob
2	3	Charlie
3	4	David
4	5	Emily

```
dfy
```

	id	country
0	2	Korea
1	2	US
2	4	US
3	5	UK
4	5	France
5	6	Italy

```
pd.merge(dfx, dfy)
```

	id	name	country
0	2	Bob	Korea
1	2	Bob	US
2	4	David	US
3	5	Emily	UK
4	5	Emily	France

Concatenating DataFrames (1)

- *pd.concat(objs, axis=0, join='outer', keys=None, ...)*
  - Append rows of *other* to the end of caller, returning a new object
  - *objs*: a sequence of DataFrame or Series
  - *axis*: the axis to concatenate along
  - *join*: how to handle indexes on other axis
  - *keys*: construct hierarchical index using the keys as the outermost level

```
df1
```

	id	name	country
0	1	Alice	Korea
1	2	Bob	US
2	9	Charlie	UK
3	5	Emily	France

```
df2
```

	id	name	age
0	4	Judy	15
1	7	David	19
2	8	Bill	11

```
pd.concat([df1, df2], axis=1)
```

	id	name	country	id	name	age
0	1	Alice	Korea	4.0	Judy	15.0
1	2	Bob	US	7.0	David	19.0
2	9	Charlie	UK	8.0	Bill	11.0
3	5	Emily	France	NaN	NaN	NaN

```
pd.concat([df1, df2])
```

	id	name	country	age
0	1	Alice	Korea	NaN
1	2	Bob	US	NaN
2	9	Charlie	UK	NaN
3	5	Emily	France	NaN
0	4	Judy	NaN	15.0
1	7	David	NaN	19.0
2	8	Bill	NaN	11.0

Concatenating DataFrames (2)

```
pd.concat([df1, df2],
          ignore_index=True)
```

	id	name	country	age
0	1	Alice	Korea	NaN
1	2	Bob	US	NaN
2	9	Charlie	UK	NaN
3	5	Emily	France	NaN
4	4	Judy	NaN	15.0
5	7	David	NaN	19.0
6	8	Bill	NaN	11.0

```
pd.concat([df1, df2], join='inner')
```

	id	name
0	1	Alice
1	2	Bob
2	9	Charlie
3	5	Emily
0	4	Judy
1	7	David
2	8	Bill

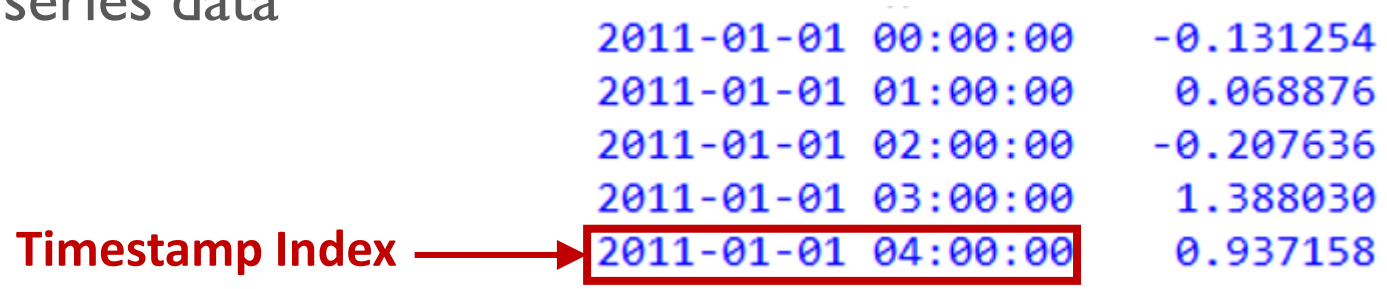
```
pd.concat([df1, df2],
          keys=['df1','df2'])
```

		id	name	country	age
df1	**0**	1	Alice	Korea	NaN
	1	2	Bob	US	NaN
	2	9	Charlie	UK	NaN
	3	5	Emily	France	NaN
df2	**0**	4	Judy	NaN	15.0
	1	7	David	NaN	19.0
	2	8	Bill	NaN	11.0

Time Series Data

Time Series Data

- How to analyze time series data?
  - Sample time series data



- Time series data is the data with the timestamp index
  - How to parse time series information from various sources and formats?
  - How to generate sequences of fixed-frequency dates and time spans
  - How to manipulate and convert date times with timezone information?
  - How to group data by time?
  - ...

Python datetime Module

- `datetime.datetime` class: a combination of date and time
  - year, month, day, hour, minute, second, microsecond, tzinfo
- `datetime.now()`: return the current local datetime

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

```
import datetime as dt
now = dt.datetime.now()
print(now)
```

```
2021-01-09 22:56:48.365683
```

```
newyear = dt.datetime(2021, 1, 1)
print(newyear)
```

```
2021-01-01 00:00:00
```

```
print(now - newyear)
```

```
8 days, 22:56:48.365683
```

NumPy datetime64 Type

- NumPy supports datetime functionality with the data type called '`datetime64`'
  - No timezone support

```
import numpy as np
now = np.datetime64('now')
print(now)
```

```
2021-01-09T13:59:25
```

```
np.arange('2021-01', '2021-07', dtype='datetime64[M]')
```

```
array(['2021-01', '2021-02', '2021-03', '2021-04', '2021-05', '2021-06'],
      dtype='datetime64[M]')
```

```
newyear = np.datetime64('2021-1-1')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-bc5dab85414d> in <module>
----> 1 newyear = np.datetime64('2021-1-1')

ValueError: Error parsing datetime string "2021-1-1" at position 5
```

Converting to Datetime

- Pandas supports extensive capabilities and features for working with time series data based on NumPy `datetime64`.
- *pd*.`to_datetime(`*arg*, ...`)`
  - Convert argument to datetime.
  - Return type can be a DatetimeIndex, Series, or Timestamp
  - *arg*: integer, float, string, datetime, list, tuple, 1-D array, Series

```
t = np.array(['1/11/2021', '2021/1/12', '20210113', '2021-1-14', '2021 1 15',
              '2021, 1, 16', 'Jan. 17 2021', '18 Jan 2021'])
pd.to_datetime(t)
```

```
DatetimeIndex(['2021-01-11', '2021-01-12', '2021-01-13', '2021-01-14',
               '2021-01-15', '2021-01-16', '2021-01-17', '2021-01-18'],
              dtype='datetime64[ns]', freq=None)
```

Generating DatetimeIndex

- *pd*.`date_range`(*start=None, end=None, periods=None, freq=None, ...*)
  - Return a fixed frequency DatetimeIndex
  - *start*: left bound for generating dates
  - *end*: right bound for generating dates
  - *periods*: the number of datetime to generate
  - *freq*: the time interval between consecutive datetime values (default: 'D')

Freq string	Description	Freq string	Description
`'D'`	One absolute day	`'M'`	Calendar month end
`'H'`	One hour	`'MS'`	Calendar month begin
`'T' or 'min'`	One minute	`'BM'`	Business month end
`'S'`	One second	`'BMS'`	Business month begin
`'B'`	Business day (weekday)	`'WOM-2THU'`	Second Thursday of the month
`'W'`	One week	`'1h30min'`	One and half hour

date_range() Examples (1)

- Default: everyday

```
pd.date_range('2021-1-11', '2021-1-17')
```

```
DatetimeIndex(['2021-01-11', '2021-01-12', '2021-01-13', '2021-01-14',
               '2021-01-15', '2021-01-16', '2021-01-17'],
              dtype='datetime64[ns]', freq='D')
```

- 7 days since 2021-1-11

```
pd.date_range('2021 1 11', periods=7)
```

```
DatetimeIndex(['2021-01-11', '2021-01-12', '2021-01-13', '2021-01-14',
               '2021-01-15', '2021-01-16', '2021-01-17'],
              dtype='datetime64[ns]', freq='D')
```

date_range() Examples (2)

- Just weekdays

```
pd.date_range('2021 1 1', '2021/1/20', freq='B')
```

```
DatetimeIndex(['2021-01-01', '2021-01-04', '2021-01-05', '2021-01-06',
               '2021-01-07', '2021-01-08', '2021-01-11', '2021-01-12',
               '2021-01-13', '2021-01-14', '2021-01-15', '2021-01-18',
               '2021-01-19', '2021-01-20'],
              dtype='datetime64[ns]', freq='B')
```

- Every Sunday

```
pd.date_range('2021-1-4', '2021-2-26', freq='W-SUN')
```

```
DatetimeIndex(['2021-01-10', '2021-01-17', '2021-01-24', '2021-01-31',
               '2021-02-07', '2021-02-14', '2021-02-21'],
              dtype='datetime64[ns]', freq='W-SUN')
```

date_range() Examples (3)

- First business day every two months

```
pd.date_range('2021-1-1', '2021-12-31', freq='2BMS')
```

```
DatetimeIndex(['2021-01-01', '2021-03-01', '2021-05-03', '2021-07-01',
               '2021-09-01', '2021-11-01'],
              dtype='datetime64[ns]', freq='2BMS')
```

- Every one and half hour

```
pd.date_range('2021-1-11 8:30', periods=7, freq='1h30min')
```

```
DatetimeIndex(['2021-01-11 08:30:00', '2021-01-11 10:00:00',
               '2021-01-11 11:30:00', '2021-01-11 13:00:00',
               '2021-01-11 14:30:00', '2021-01-11 16:00:00',
               '2021-01-11 17:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Finding the Day of the Week

- *pd*.DatetimeIndex.day_name(**args*, ...)
  - Return the day names of the DatetimeIndex

```
idx = pd.date_range(start='2021-01-01', freq='D', periods=3)
idx
```

```
DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03'], dtype='datetime64[n
s]', freq='D')
```

```
idx.day_name()
```

```
Index(['Friday', 'Saturday', 'Sunday'], dtype='object')
```

```
war = pd.date_range(start='1950-6-25', freq='D', periods=3)
war.day_name()
```

```
Index(['Sunday', 'Monday', 'Tuesday'], dtype='object')
```

Creating Time Series Data

- Create a range of `DatetimeIndex` object

```
ts = pd.date_range('1/1/2021', periods=4, freq='2H')
ts
```

```
DatetimeIndex(['2021-01-01 00:00:00', '2021-01-01 02:00:00',
               '2021-01-01 04:00:00', '2021-01-01 06:00:00'],
              dtype='datetime64[ns]', freq='2H')
```

- Use the `DatetimeIndex` object as Pandas Series or DataFrame index

```
df = pd.DataFrame({'A':np.random.randn(len(ts)),
                   'B':np.random.randn(len(ts))},
                  index=ts)
df
```

	A	B
2021-01-01 00:00:00	0.801227	1.536040
2021-01-01 02:00:00	0.004163	2.357660
2021-01-01 04:00:00	0.043119	-2.062572
2021-01-01 06:00:00	0.302858	0.446922

Example: Pandas Time Series Data (1)

- Create a dataframe: input dataset = <*timestamp, access count*>

```
import pandas as pd

data = {'date': ['2021-01-01 08:47:05.069722',
                 '2021-01-01 18:47:05.119994',
                 '2021-01-02 08:47:05.178768',
                 '2021-01-02 13:47:05.230071',
                 '2021-01-02 18:47:05.230071',
                 '2021-01-02 23:47:05.280592',
                 '2021-01-03 08:47:05.332662',
                 '2021-01-03 18:47:05.385109',
                 '2021-01-04 08:47:05.436523',
                 '2021-01-04 18:47:05.486877'],
 'counts': [34, 25, 26, 15, 15, 14, 26, 25, 62, 41]}
df = pd.DataFrame(data)
df
```

	date	counts
0	2021-01-01 08:47:05.069722	34
1	2021-01-01 18:47:05.119994	25
2	2021-01-02 08:47:05.178768	26
3	2021-01-02 13:47:05.230071	15
4	2021-01-02 18:47:05.230071	15
5	2021-01-02 23:47:05.280592	14
6	2021-01-03 08:47:05.332662	26
7	2021-01-03 18:47:05.385109	25
8	2021-01-04 08:47:05.436523	62
9	2021-01-04 18:47:05.486877	41

Example: Pandas Time Series Data (2)

- Convert `df['date']` from string to datetime

```
df.date = pd.to_datetime(df.date)
```

- Set `df['date']` as the index

```
df = df.set_index('date')
```

	counts
date	
2021-01-01 08:47:05.069722	34
2021-01-01 18:47:05.119994	25
2021-01-02 08:47:05.178768	26
2021-01-02 13:47:05.230071	15
2021-01-02 18:47:05.230071	15
2021-01-02 23:47:05.280592	14
2021-01-03 08:47:05.332662	26
2021-01-03 18:47:05.385109	25
2021-01-04 08:47:05.436523	62
2021-01-04 18:47:05.486877	41

Accessing Time Series Data (1)

- View data in 2021

```
df['2021']
```

date	counts
2021-01-01 08:47:05.069722	34
2021-01-01 18:47:05.119994	25
2021-01-02 08:47:05.178768	26
2021-01-02 13:47:05.230071	15
2021-01-02 18:47:05.230071	15
2021-01-02 23:47:05.280592	14
2021-01-03 08:47:05.332662	26
2021-01-03 18:47:05.385109	25
2021-01-04 08:47:05.436523	62
2021-01-04 18:47:05.486877	41

- View data in January 2021

```
df['2021-01']
```

date	counts
2021-01-01 08:47:05.069722	34
2021-01-01 18:47:05.119994	25
2021-01-02 08:47:05.178768	26
2021-01-02 13:47:05.230071	15
2021-01-02 18:47:05.230071	15
2021-01-02 23:47:05.280592	14
2021-01-03 08:47:05.332662	26
2021-01-03 18:47:05.385109	25
2021-01-04 08:47:05.436523	62
2021-01-04 18:47:05.486877	41

Accessing Time Series Data (2)

- Observations after 12:00, Jan. 3, 2021

```
df['2021/1/3 12:00':]
```

date	counts
2021-01-03 18:47:05.385109	25
2021-01-04 08:47:05.436523	62
2021-01-04 18:47:05.486877	41

- Observations between Jan. 1 - 2

```
df['1/1/2021':'1/2/2021']
```

date	counts
2021-01-01 08:47:05.069722	34
2021-01-01 18:47:05.119994	25
2021-01-02 08:47:05.178768	26
2021-01-02 13:47:05.230071	15
2021-01-02 18:47:05.230071	15
2021-01-02 23:47:05.280592	14

Accessing Time Series Data (3)

- Truncate observations after Jan. 3, 2021

```
df.truncate(after='1/3/2021')
```

date	counts
2021-01-01 08:47:05.069722	34
2021-01-01 18:47:05.119994	25
2021-01-02 08:47:05.178768	26
2021-01-02 13:47:05.230071	15
2021-01-02 18:47:05.230071	15
2021-01-02 23:47:05.280592	14

- Total counts per day

```
df.resample('D').sum()
```

date	counts
2021-01-01	59
2021-01-02	70
2021-01-03	51
2021-01-04	103

Resampling

- The process of converting a time series from one frequency to another
  - Downsampling (similar to groupby operation): aggregating higher frequency data to lower frequency
  - Upsampling: converting lower frequency to higher frequency

- *df*.`resample(`*rule, axis=0, closed=None, label=None, ...*`)`
  - *rule*: the offset string or object representing target conversion
  - *axis*: axis to use for up- or down-sampling
  - *closed*: which side of bin interval is closed (default: 'right' for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W', and 'left' for others)
  - *label*: which bin edge label to label bucket with (same default value as *closed*)

Downsampling

closed='left'

	counts
2021-01-01 00:00:00	0
2021-01-01 00:01:00	1
2021-01-01 00:02:00	2
2021-01-01 00:03:00	3
2021-01-01 00:04:00	4
2021-01-01 00:05:00	5
2021-01-01 00:06:00	6
2021-01-01 00:07:00	7
2021-01-01 00:08:00	8
2021-01-01 00:09:00	9
2021-01-01 00:10:00	10
2021-01-01 00:11:00	11

```
df.resample('5min').sum()
```

	counts
2021-01-01 00:00:00	10
2021-01-01 00:05:00	35
2021-01-01 00:10:00	21

```
df.resample('5min', closed='right').sum()
```

	counts
2020-12-31 23:55:00	0
2021-01-01 00:00:00	15
2021-01-01 00:05:00	40
2021-01-01 00:10:00	11

closed='right'

```
df.resample('5min', closed='right', label='right').sum()
```

	counts
2021-01-01 00:00:00	0
2021-01-01 00:05:00	15
2021-01-01 00:10:00	40
2021-01-01 00:15:00	11

OHLC Resampling

- Open-High-Low-Close resampling
  - Used in finance

```
df.resample('5min').ohlc()
```

	counts			
	open	high	low	close
2021-01-01 00:00:00	0	4	0	4
2021-01-01 00:05:00	5	9	5	9
2021-01-01 00:10:00	10	11	10	11

	counts
2021-01-01 00:00:00	0
2021-01-01 00:01:00	1
2021-01-01 00:02:00	2
2021-01-01 00:03:00	3
2021-01-01 00:04:00	4
2021-01-01 00:05:00	5
2021-01-01 00:06:00	6
2021-01-01 00:07:00	7
2021-01-01 00:08:00	8
2021-01-01 00:09:00	9
2021-01-01 00:10:00	10
2021-01-01 00:11:00	11

Upsampling

```
df.resample('20s').ffill()
```

	counts
2021-01-01 00:00:00	0
2021-01-01 00:00:20	0
2021-01-01 00:00:40	0
2021-01-01 00:01:00	1
2021-01-01 00:01:20	1
2021-01-01 00:01:40	1
2021-01-01 00:02:00	2
2021-01-01 00:02:20	2
2021-01-01 00:02:40	2
2021-01-01 00:03:00	3

```
df.resample('20s').asfreq()
```

	counts
2021-01-01 00:00:00	0.0
2021-01-01 00:00:20	NaN
2021-01-01 00:00:40	NaN
2021-01-01 00:01:00	1.0
2021-01-01 00:01:20	NaN
2021-01-01 00:01:40	NaN
2021-01-01 00:02:00	2.0
2021-01-01 00:02:20	NaN
2021-01-01 00:02:40	NaN
2021-01-01 00:03:00	3.0

```
df.resample('20s').bfill(limit=1)
```

	counts
2021-01-01 00:00:00	0.0
2021-01-01 00:00:20	NaN
2021-01-01 00:00:40	1.0
2021-01-01 00:01:00	1.0
2021-01-01 00:01:20	NaN
2021-01-01 00:01:40	2.0
2021-01-01 00:02:00	2.0
2021-01-01 00:02:20	NaN
2021-01-01 00:02:40	3.0
2021-01-01 00:03:00	3.0

	counts
2021-01-01 00:00:00	0
2021-01-01 00:01:00	1
2021-01-01 00:02:00	2
2021-01-01 00:03:00	3

Aggregation Functions

function	Description
`.asfreq()`	Return the values at the new freq, essentially a reindex
`.fillna()`	Fill missing values introduced by upsampling ('ffill', 'bfill', 'nearest')
`.ffill() / .bfill()`	Forward (or backward) fill the values
`.first() / .last()`	Compute first (or last) of group values
`.min() / .max()`	Compute min (or max) of group values
`.mean() / .median()`	Compute mean (or median) of groups, excluding missing values
`.sum()`	Compute sum of group values
`.std() / .var()`	Compute standard deviation (or variance) of groups, excluding missing values
`.ohlc()`	Compute open, high, low and close values of a group, excluding missing values
`.count()`	Compute count of group, excluding missing values
`.nunique()`	Return number of unique elements in the group
`.quantile()`	Return value at the given quantile
`.interpolate()`	Interpolate values according to different methods

Moving Window Functions

- *df*.rolling(*window, min_periods=None, ...*)
  - Provide rolling window calculations
  - *window*: size of the moving window
  - *min_periods*: minimum number of observations in window required to have a value

- *df*.ewm(*com=None, span=None, alpha=None, min_periods=0, ...*)
  - Provide exponential weighted (EW) functions
  - *com*: specify decay in terms of center of mass, $\alpha = \frac{1}{1+com}$ $for\ com \geq 0$
  - *span*: specify decay in terms of span, $\alpha = \frac{2}{span+1}$ $for\ span \geq 1$
  - *alpha*: specify smoothing factor α directory, $0 < \alpha \leq 1$

Example:

```
lge = pd.read_csv('066570.csv')
lge.head()
```

	Date	Open	High	Low	Close	Volumn	AdjOpen	AdjHigh	AdjLow	AdjClose	AdjVolumn
0	2020-09-18	90700.0	90900.0	NaN	NaN	NaN	NaN	89900.0	NaN	NaN	NaN
1	2020-09-17	91000.0	91200.0	NaN	NaN	NaN	NaN	90600.0	NaN	NaN	NaN
2	2020-09-16	91200.0	93300.0	NaN	NaN	NaN	NaN	90600.0	NaN	NaN	NaN
3	2020-09-15	92000.0	93100.0	NaN	NaN	NaN	NaN	91800.0	NaN	NaN	NaN
4	2020-09-14	93500.0	94900.0	NaN	NaN	NaN	NaN	92100.0	NaN	NaN	NaN

Example:

```
lge.tail()
```

	Date	Open	High	Low	Close	Volumn	AdjOpen	AdjHigh	AdjLow	AdjClose	AdjVolumn
245	2019-09-25	66400.0	66900.0	NaN	NaN	NaN	NaN	65503.6466	NaN	NaN	NaN
246	2019-09-24	64800.0	67200.0	NaN	NaN	NaN	NaN	66097.3352	NaN	NaN	NaN
247	2019-09-23	65800.0	65800.0	NaN	NaN	NaN	NaN	64415.2174	NaN	NaN	NaN
248	2019-09-20	65200.0	65900.0	NaN	NaN	NaN	NaN	65206.8022	NaN	NaN	NaN
249	2019-09-19	66600.0	66700.0	NaN	NaN	NaN	NaN	64514.1655	NaN	NaN	NaN

Example:

```
lge.Date = pd.to_datetime(lge.Date)
lge = lge.set_index('Date')
lge = lge[['Open', 'High']]
lge.describe()
```

	Open	High
count	250.000000	250.00000
mean	67286.600000	68261.40000
std	10312.154604	10560.35722
min	42350.000000	42850.00000
25%	61325.000000	61750.00000
50%	67950.000000	68950.00000
75%	71400.000000	71975.00000
max	93500.000000	95000.00000

Example:

```
%config InlineBackend.figure_format = 'svg'
#import matplotlib.pyplot as plt
lge.High.plot()
lge.High.rolling('5d').mean().plot()
lge.High.rolling('30d').mean().plot()
#plt.legend(['High','5D','30D'])
```