Data Processing with Numpy / Pandas

Handle Numeric Values

- In Data Analysis, it often implements high-speed computing for huge data.
- Handle multiple values with List (array), Tuple, Dictionary, etc. in Python.

List(array):
```
lst = [ 0, 1, 2 ]
print( lst[1] ) → 1
```
Organize sequential values ○○, etc.

Tuple:
```
tp = ( 0, 1, 2 )
print( tp[1] ) → 1
```
Similar to Python List (Unchangeable list)

Dictionary:
```
d = { 'k1':0, 'k2':1, … }
print( d['k2'] ) → 1
```
Organize values with key-value pairs, key is a String type.

Useful Libraries for Numerical Computing and Spreadsheet Computation

For numerical computing using Python, Numpy and Pandas are broadly adopted. Similar to Python List(array), an unique array format is provided that is enhanced for high-speed calculations.

- Numpy
  - A Python library developed for high-speed numerical computation.
  - Focus on a multidimensional array of all elements of the same data type.
- Pandas
  - A library that extends Numpy, particularly useful for handling tabular data.
  - The data type may be different in column. Easier to use than Numpy, especially when it contains time series data.
  - The calculation speed is a little slower than Numpy, but there are many functions that are useful for data processing, etc.

Official Manual of Numpy / Pandas

https://docs.scipy.org/doc/numpy/index.html

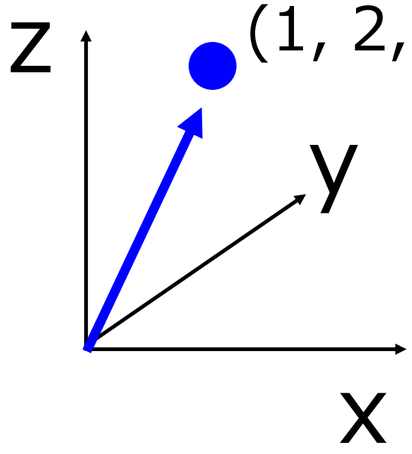
http://pandas.pydata.org/pandas-docs/stable/

Numpy and Pandas have so many features.

Please refer to it for more details that can not be explained in this lecture.

There is also various info on the Web.

Dimension and the number of elements of an Array



1d array with 3 elements
= 3d vector

[1, 2, 3]

A 3x4 matrix (2d array)

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

[[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]]

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} & \begin{matrix} 0 \\ 1 \end{matrix} \\ \begin{pmatrix} -3 & 1 & -1 & 2 \end{pmatrix} & -3 \\ \begin{pmatrix} 9 & 6 & 4 & 2 \end{pmatrix} & 2 \end{pmatrix} \cdots$$

3d array

[[[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]],
[[-3, 1, -1, 2],
…]

Arrays in Numpy and Pandas

- Numpy handles arrays with a data type called ndarray (numpy array)
- For Pandas, 1d data is called "Series", and 2d (tabular) data is called "DataFrame".
  - Regarding Series and DataFrame, they attach line numbers for rows and labels for columns to a ndarray.
  - Details will be described later.

First of All, Import Libraries

Regarding Numpy and Pandas, It almost becomes standard to import them as follows.

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

← import numpy and use "np" as an alias.

← import pandas and use "pd" as an alias.

Hereafter,
numpy.function_name() can be written as np.function_name()
pandas.function_name() can be written as pd.function_name()

How to Create a Numpy Array

You can easily create numpy arrays (ndarrays) from Python lists using the np.array function.

lst = [0, 1, 2]
List

v1d = **np.array(lst)**
numpy array (ndarray)

Difference between List and numpy array:
numpy array is much suitable for numerical computing

lst * 2 → [0, 1, 2, 0, 1, 2] repeat twice

v1d * 2 → [0, 2, 4] each element*2

Operation of Numpy Array(1d arrays)

```
v1 = np.array( [ 1.0, 2.0, 3.0 ] )
v2 = np.array( [ 2.0, 4.0, 6.0 ] )
```

```
print( v1 + v2 ) → [ 3.0, 6.0, 9.0 ]
print( v1 * v2 ) → [ 3.0, 8.0, 18.0 ]
```

Element-wise arithmetic operations

```
print( np.dot(v1, v2 )) → 28.0
```

np.dot() for computing the inner product.
※np.outer(v1, v2) for computing the outer product

The inner product is the operation of multiplying the corresponding elements of v1 and v2 and then adding them together. Mathematically, we refer to it as v1 • v2.

```
v1:  1.0   2.0   3.0
       ×     ×     ×
v2:  2.0   4.0   6.0
     2.0  +  8.0  + 18.0 = 28.0 (← v1 • v2)
```

(Adv) Product between Matrix(2d array) and Vector(1d array)

```
A = np.array( [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]] )  2d array
x = np.array( [1.0, 1.0, 1.0] )   1d array
b = np.array( [2.0, 2.0] )  1d array

print( A * x ) → [ [1 2 3]
                   [4 5 6] ]
```

2d array * 1d array is a multiplication operation in which row elements of 2d array multiple with 1d array.

```
y = np.dot(A, x) + b
print( y )  → [ 8, 17 ]
```

If using np.dot(), it becomes a product of matrix and vector in Math.

Initialize a Numpy array

- Create an array without initializing entries.
  - A = np.empty(5) → a 1d array with 5 elements (random values)
  - A = np.empty((2,3)) → a 3x2 2d array (random values)
- Create an array and initialize entries.
  - A = np.zeros((2,3)) → a 2x3 2d array setting values to zero
  - A = np.ones((2,3)) → a 2x3 2d array setting values to one
  - A = np.identity(3) → a 3x3 2d identity array with its main diagonal set to one, and all other elements 0.

Because it is faster than initializing individual elements by assigning 0's and 1's to them, so these functions are useful when A is a large scale matrix.

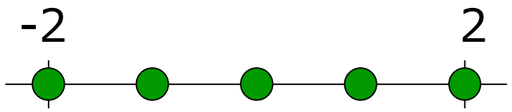
Generation of Sequential Numbers (arithmetic progression)

np.arange([start,] stop[, step])

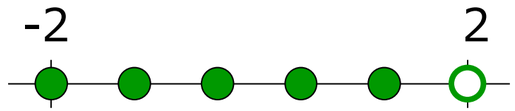
np.arange(5) → [0 1 2 3 4]
np.arange(2, 8) → [2 3 4 5 6 7]
np.arange(2, 8, 2) → [2 4 6]
np.arange(1.8, -1.8, -0.9) → [1.8 0.9 0. -0.9]

np.linspace(start, stop, num=#samples, endpoint=True)

np.linspace(-2, 2, num=5) → [-2. -1. 0. 1. 2.]



np.linspace(-2, 2, num=5, endpoint=False)
→ [-2. -1.2 -0.4 0.4 1.2]



Note:#samples is number of samples to generate.

Change the Shape of a Numpy Array

Method reshape() returns an array containing same data with a new shape.

v1 = np.arange(6) → [0 1 2 3 4 5]

v2 = v1.reshape(2, 3) → [[0 1 2]
[3 4 5]]

Reshape a 2x3 array

※ v2 = v1.reshape(-1, 3)
It is also OK.

One shape dimension can be -1 that the values is inferred automatically.

v3 = v2.T → [[0 3]
[1 4]
[2 5]]

Transpose (switch rows and columns)

Other Operations on Numpy Array

Let X be a numpy array.

- Display the number of rows and columns of X:　X**.shape**
  - If X is a 2x3 2d array, print(X.shape) → (2, 3)
  - If X is a 1d array with 5 elements, print(X.shape) → (5,)

※ Because it is a tuple, it will be (5,) instead of (5)
(To distinguish it from (5) in the formula)

- Make a copy
  - Let X2 = X that X2 is only an alias to X while the entity is same,
    X2 = X**.copy()** that makes a copy of X and assigns it to X2.

Manual of Numpy Methods（Functions）

- Methods for Statistical Analysis（Functions）

  https://docs.scipy.org/doc/numpy/reference/routines.statistics.html

- Mathematical Functions

  https://docs.scipy.org/doc/numpy/reference/routines.math.html

Computation of Statistical Measurements

Let dat be a 2d array

np.*function*(dat, axis=0) ←Column-wise statistic
np.*function*(dat, axis=1) ←Row-wise statistic

Example of「*function*」
the minimum of an array: amin
the maximum of an array: amax
the average of an array: mean
the median of an array: median
standard deviation (population) of an array: std
summation of an array: sum

※Compute unbiased standard deviation, using
np.std(dat, ddof=1, axis=0)

Tabular Data and CSV File

- Data is often organized in "Table" format.
- For tabular data, it is often stored in CSV (comma-separated values) format.
  - CSV is a data format in which values are separated by delimiters, such as comma.

CSV file

delimiter

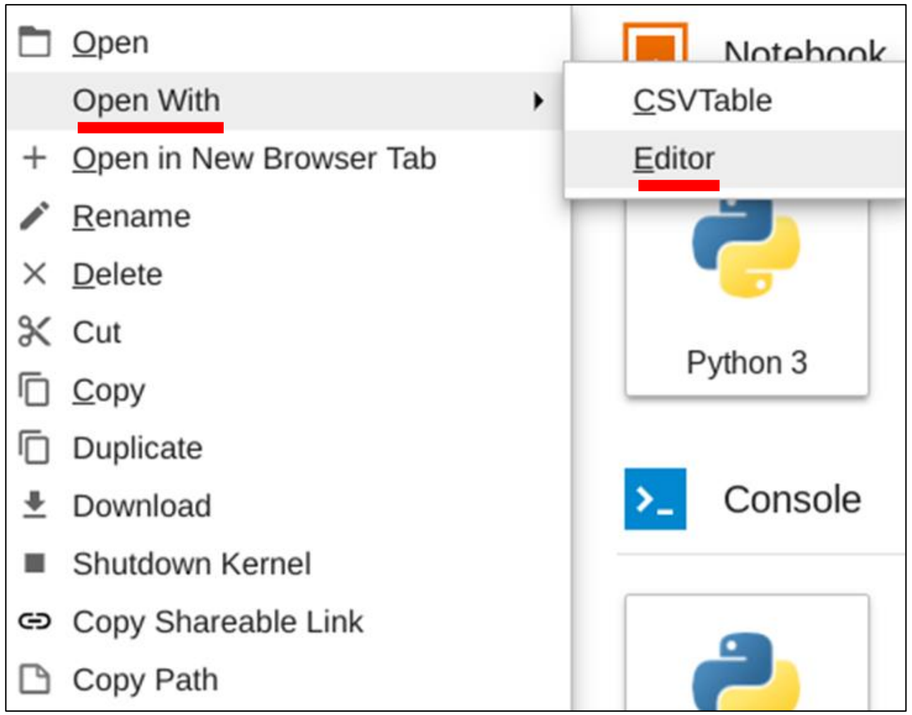
```
Employee data,,,, …
,,,, …
ID,Family,Given,Dept, …
1010,Akabane,Taro,Accounting, …
1021,Hakusan,Hanako,Development, …
1032,Kawagoe,Jiro,Sales, …
```

Excel, etc.

Employee data				
ID	Family	Given	Dept	…
1010	Akabane	Taro	Accounting	…
1021	Hakusan	Hanako	Development	…
1032	Kawagoe	Jiro	Sales	…

Check a CSV File

- Right-click pandas_training-utf8.csv in List view of JupyterLab, Open With > Editor



※ Note that Editor can not open if it contains characters of kanji code other than UTF-8
Use VSCode of other editors instead.

Check the CSV File

- Check the contents of the CSV file with Editor. First, note that it starts with13 lines of descriptions other than tabular data, the next line is a header line containing column labels, and then the following lines are tabular data (aka values).

13 lines other than tabular data

```
1  •Wholesale Customers Data Set,,,,,,,
2  https://archive.ics.uci.edu/ml/datasets/wholesale+customers,,,,,,,
3  "Abreu, N. (2011). Analise do perfil do cliente Recheio e desenvolvimento de um
   sistema promocional. Mestrado em Marketing, ISCTE-IUL, Lisbon",,,,,,,
4  ,,,,,,,
5  CHANNEL(販路): customers's Channel - Horeca (Hotel/Restaurant/Cafe) or Retail
   channel (Nominal),,,,,,,
6  "REGION(地域): customers's Region - Lisbon, Oporto or Other (Nominal) ",,,,,,,
7  FRESH(生鮮): annual spending (m.u.) on fresh products (Continuous); ,,,,,,,
8  MILK(乳製品): annual spending (m.u.) on milk products (Continuous); ,,,,,,,
9  GROCERY(食料品): annual spending (m.u.)on grocery products (Continuous); ,,,,,,,
10 FROZEN(冷凍): annual spending (m.u.)on frozen products (Continuous) ,,,,,,,
11 DET_PAPER(洗剤・紙類): annual spending (m.u.) on detergents and paper products
   (Continuous) ,,,,,,,
12 DELICA(惣菜): annual spending (m.u.)on and delicatessen products (Continuous);
   ,,,,,,,
13 ,,,,,,,
14 CHANNEL,REGION,FRESH,MILK,GROCERY,FROZEN,DET_PAPER,DELICA
15 Retail,Other,12669,9656,7561,214,2674,1338
```

Column lables

The following is data (delimiter is comma)

Reading a CSV File

- In Pandas, tabular data (2d) is called a DataFrame, one column or one row data is called Series.
- Pandas' read_csv() read data in a CSV file into a DataFrame.
- DataFrame attaches row numbers (index) and column labels (columns) to tabular data. Row numbers and column labels will be automatically numbered starting from 0 if not specified.

Read data into a dataframe named df (variable name can be anything else).

Delimiter (can be omitted if it is comma)

The number of lines to skip at the start of the file.

Specify the row number as column names after skipping (row number, start is 0)

```
df = pd.read_csv('CSV file name', delimiter=',', skiprows=N, header=M)
```

delimiter=',' can be sep=','

Reading a CSV File

df = pd.read_csv('CSV file name', delimiter=**','**, skiprows=**N**, header=**M**)

- Skip the first 13 lines and use the next line as column name,

  skiprows=13, header=0

- If the column names contains extra spaces, you can the following option.

  skipinitialspace=True

- When the kanji code is other than utf8, etc., you can specify the following option

  encoding='shift-jis'

If the delimiter is space(s),
set delimiter='¥s+'.
※If the delimiter is tab,
one can set delimiter='¥t'.

Data Types of Columns

- If int(integer), float, object(string) , etc., are clearly inferred, there is no need to specify anything.
- For example, the ID is 0001, 0002, … , and their data type can be inferred automatically as an integer; If the leading "00…" is deleted, 、 the type becomes object(string).
- The data type of the column contains missing values (NaN, np.nan) becomes float.
- If automatic detection does not work, object will be adopted and you have to modify data types yourself.

※ When data contains missing values, the missing values are represented with NaN (Not a Number) or np.nan (data type is float).

Let's Use the Manual

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

pandas.read_csv

pandas.read_csv(*filepath_or_buffer*, *sep=', '*, *delimiter=None*, *header='infer'*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *prefix=None*, *mangle_dupe_cols=True*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skipinitialspace=False*, *skiprows=None*, *skipfooter=0*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *na_filter=True*, *verbose=False*, *skip_blank_lines=True*, *parse_dates=False*, *infer_datetime_format=False*, *keep_date_col=False*, *date_parser=None*, *dayfirst=False*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal=b'.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *doublequote=True*, *escapechar=None*, *comment=None*, *encoding=None*, *dialect=None*, *tupleize_cols=None*, *error_bad_lines=True*, *warn_bad_lines=True*, *delim_whitespace=False*, *low_memory=True*, *memory_map=False*, *float_precision=None*) [source]

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

filepath_or_buffer : *str, path object, or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

dtype : *Type name or dict of column -> type, optional*

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use *str* or *object* together with suitable *na_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

```
df = pd.read_csv('CSV_file_name',
                 dtype={'Col1':'object', 'Col2':'int', …},
                 delimiter=',', skiprows=N, header=M)
```

or

```
df = pd.read_csv('CSV file name',
                 dtype='object',
                 delimiter=',', skiprows=N, header=M)

df['Col2'] = df['Col2'].astype('int')
```

Specify data types column by column

Read the whole as object (string) type, and make type conversion as needed later

	0
0	Wholesale Customers Data Set,,,,,,
1	https://archive.ics.uci.edu/ml/datasets/wholes...
2	Abreu, N. (2011). Analise do perfil do cliente...
3	,,,,,,
4	CHANNEL(販路): customers's Channel - Horeca (Hot...
5	REGION(地域): customers's Region - Lisbon, Oport...
6	FRESH(生鮮): annual spending (m.u.) on fresh pro...
7	MILK(乳製品): annual spending (m.u.) on milk prod...
8	GROCERY(食料品): annual spending (m.u.)on grocery...
9	FROZEN(冷凍): annual spending (m.u.)on frozen pr...
10	DET_PAPER(洗剤・紙類): annual spending (m.u.) on de...
11	DELICA(惣菜): annual spending (m.u.)on and delic...
12	,,,,,,
13	CHANNEL,REGION,FRESH,MILK,GROCERY,FROZEN,DET_P...
14	Retail,Other,12669,9656,7561,214,2674,1338

The delimiter is ",".
Skip 13 lines. The next line is column labels.
It might use automatic type detection.

Therefore we know the above info without opening the file in the editor.
As you did, you can specify options, delimiter=',', skiprows=13, header=0

(Adv) If the CSV file is Huge

- To deal with a large CSV file that can not be checked by opening it in the editor to find the delimiter and start line, etc.
  - On Mac or Linux, you can display the heads of the file with "head" command, "head CSV_file_name"
  - On Windows, You can check the head part of the file by the following codes.

```
df_head = pd.read_csv(CSV_file_name,  delimiter='¥n',  dtype='object',
                                skiprows=None,  header=None,  nrows=15)
display(df_head)
```

Because the line feed code is used as the delimiter, it becomes a DataFrame with one column. The data type is object(string) (since we have no idea about data). Do not specify the number of skipping rows and the row containing labels. Thru nrows=15, only the first 15 lines will be read.

DataFrame and Series

DataFrame df

Column

df.columns
(Column label)

Row

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

df.index (row number)

Categorical
(Quality) data

numerical data

df.values
(ndarray)

DataFrame and Series

Series: Equivalent to one column or row of DataFrame

Extract one column from df

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

ser = df['MILK']

variable ser can be anything else.

Extract one row from df

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

ser = df.loc[2]

ser

```
0    9656
1    9810
2    8808
Name: MILK, dtype: int64
```

Name(ser.name) is attached.

ser.index

```
CHANNEL      Retail
REGION       Other
FRESH         6353
MILK          8808
GROCERY       7684
FROZEN        2405
DET_PAPER     3516
DELICA        7844
Name: 2, dtype: object
```

Data type (ser.dtype)
※「object」is string type.

DataFrame

- Let df be a DataFrame variable
  - df.shape
    display the number of rows and columns (row numbers only: df.shape[0], column numbers only: df.shape[1])
  - df.dtypes
    display data type of each column
  - df.info()
    display summary of a DataFrame, including data types of columns, non-null values and memory usage.
  - df.describe()
    display descriptive statistics of columns, the number of data, maximum, minimum, mean, etc.
  - df.describe(exclude='number')
    display summary of columns except numeric columns, the number of data, types of values, mode
  - df.head(N)
    display first N rows (if N is omitted, N is 5) ※df.tail(N) for the last N rows.

Other Operations on a DataFrame

- Pretty print for row numbers, column labels, etc.
  - use display instead of print().
- Copy data
  - Let df2 = df that df2 is only an alias to df while the entity is same, df2 = df**.copy()** that makes a copy of df and assigns it to df2.
- Attach column names
  - df.columns = ['col1', 'col2', …]
- Change column names
  - df2 = df.rename(columns = **{** 'old1' **:** 'new1', 'old2' **:** 'new2', … **}**)

Other Operations on a DataFrame

- Switch rows and columns
  - use transpose of matrix

```
df2 = df.T
```

df

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

df2

	0	1	2	3	4	5	6	7	8	9	...
CHANNEL	Retail	Retail	Retail	Horeca	Retail	Retail	Retail	Retail	Horeca	Retail	...
REGION	Other	Other	Other	Other	Other	Other	Other	Other	Other	Other	...
FRESH	12669	7057	6353	13265	22615	9413	12126	7579	5963	6006	...
MILK	9656	9810	8808	1196	5410	8259	3199	4956	3648	11093	...
GROCERY	7561	9568	7684	4221	7198	5126	6975	9426	6192	18881	...

Create Series/DataFrame from List

Example: List(1d array) lst1 = [2, 4, 6]

Convert it to Pandas(alias pd) Series "ser"

ser = **pd.Series**(lst1) ➡ index is automatically generated

```
0    2
1    4
2    6
dtype: int64
```

Example: List lst2 = [[2, 4, 6], [1, 3, 5]]

Convert it to Pandas(alias pd) DataFrame "df"

df = **pd.DataFrame**(lst2) ➡ index and columns are automatically generated.

	0	1	2
0	2	4	6
1	1	3	5

Extract the Entire Row/Column

- To extract a row from DataFrame "df", do as follows

ser = df.loc[**row number(start from 0)**]

Series ser
(variable name can be anything else)

It refers to index

Example: **ser_row2** = df.loc[**2**]
row number 2
(it means the 3rd row since "2" is the third one of indices, 0,1,2,…)

- To retrieve multiple rows(DataFrame) by a slice notation(**start:stop**)

Variable names on the left can be anything else. Note that [] is doubled if a list is given.

df2 = df.loc[**[RowNum1, RowNum2, …]**]

Example: **df_list** = df.loc[**[1, 3]**]

There are 2 rows in total between row 1 and row 3

df2 = df.loc[**start : stop**]

Example: **df_slice** = df.loc[**1:3**]

There are 3 lines in total between row 1 and row 3

Extract the Entire Row/Column

- To extract a column from DataFrame "df", do as follows

ser = df[**'Column_name'**] Example: **ser_milk** = df[**'MILK'**]

Series ser (variable name can be anything else)

- If names of multiple columns is given as a list, you can retrieve multiple column values (as a DataFrame).

Variable names on the left can be anything else. Note that [] is doubled if a list is given.

df2 = df[**['Col1', 'Col2', ⋯]**]

Example: **df_list** = df[**['FRESH', 'GROCERY']**]

Extract the Entire Row/Column

- To retrieve multiple columns (as a DataFrame) by a slice notation (**start:stop**).

use df.loc[…] with a slice notation

df2 = df.loc[**start : stop, 'start_col_name' : 'stop_col_name**']

Example: **df_slice** = df.loc[**:, 'FRESH' : 'GROCERY'**]

Return a slice of values corresponding to the entire rows and columns from 'FRESH' to 'GROCERY'.

Either the start or stop can be omitted in slice notation, in which case they default to the start and the end. If omitting both start and stop, **":"** means starts from the beginning to the end.

Example: df2 = df.loc[**:2, 'GROCERY':**] values at row index starting from the beginning to row 2 and column index starting from "GROCERY" to the end.

Extract the Entire Row/Column

- Using .loc with a label-based slice notation being "start : stop : step". It means indexing starts from the start to the stop with an interval of step.

Example: df2 = df.loc[**:, 'FRESH':'DET_PAPER':2**]

Row index: starts from the beginning to the end
Column index: starts from FRESH to DET_PAPER with a step size, 2 (skip 1)

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

- If step is negative, access data in reverse order

Example: df2 = df.loc[**3:1:-1**]

Starts from row 3 to row 1 with a step size of 1 in reverse order.

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
3	Horeca	Other	13265	1196	4221	6404	507	1788
2	Retail	Other	6353	8808	7684	2405	3516	7844
1	Retail	Other	7057	9810	9568	1762	3293	1776

Specify Columns with Integer Position

- For integer-location based selection, use iloc instead of loc. In this case, note that **start : stop means** from start to **stop-1** !

Example: df2 = df.**iloc**[1:3, 2:6] Row 1 to 2, Column 2 to **5.**

Column index 0 1 2 3 4 5 6 7

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

Extract the Single Value of a Cell

- .at extracts a single value for a row/column label pair, .iat for integer-location based lookups. They are faster than loc/iloc.

Example 1: val = df.**at**[1, **'FRESH'**]

Example 2: val = df.**iat**[1, **2**]

In both examples, extract the value of the cell corresponding to row 1 and column「FRESH」(column 2) , assign the value to variable val,

Column index 0 1 2 3 4 5 6 7

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

Column-wise Operation

Example: ser = 2 * df['FRESH'] + df['MILK'] − 10000

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

```
0    22899
1    13682
2    10390
3    20751
```

Computation of element-wise operations is performed all at once.

Compute Column/Row-Wise Statistics with DataFrame

axis=0 can be omitted.

```
ser_sum_col = df.loc[:, 'FRESH':'DELICA'].sum(axis=0)
```

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

```
0     34112
1     33266
2     36610
3     27381
4     46100
dtype: int64
```

```
ser_sum_row = df.loc[:, 'FRESH':'DELICA'].sum(axis=1)
```

```
FRESH          5280131
MILK           2550357
GROCERY        3498562
FROZEN         1351650
DET_PAPER      1267857
DELICA          670943
dtype: int64
```

sum() is a column/row-wise summation. Non-numeric values are ignored. Except for this, there are many statistical functions available in DataFrame, such as mean(), var(), std(), max(), min(), etc.

Compute Statistics with Series

Similarly to DataFrame, for Series, There are many statistical functions available, such as Series.mean () and Series.max ().

Example: ser = df['DELICA']; print(ser.mean())

Or even, you can write print(df['DELICA'].mean())

Select a Subset of a DataFrame with Conditions

df[Cond.] retrieves a subset of df that matches the conditions
Example: df['CHANNEL']=='Retail' where values of column CHANNEL is 'Retail'.
For multiple conditions each enclosed with parentheses, use &(and), |(or), and ~(not) to combine them.

Example 1: df_ret = df**[** df['CHANNEL']=='Retail' **]**

Extract rows whose values are equal to "Retail" in column CHANNEL in df, assign them to variable df_ret.

Example 2: df_ret_10k = df**[(**df['CHANNEL']=='Retail'**) & (**df['FRESH']>10000**)]**

Extract rows from df where values in column CHANNEL are equal to "Retail", **and** values in column FRESH are larger than 10000, assign them to variable df_ret_10k.

Example 3: df_noret_10k = df**[~(**df['CHANNEL']=='Retail'**) & (**df['FRESH']>10000**)]**

Extract rows from df where values in column CHANNEL are **NOT** equal to "Retail", **and** values in column FRESH are larger than 10000, assign them to variable df_noret_10k.

Select a Subset of a DataFrame with Conditions

Example 4: When extracting column FRESH where values in column CHANNEL are equal to "Retail",

```
ser_ret_ch = df[ df['CHANNEL']=='Retail' ].loc[:, 'FRESH']
```

It becomes the subset extracted from column FRESH in example 2.

Of course, you can use the result of example 1 to achieve data as below

```
ser_ret = df[ df['CHANNEL']=='Retail' ]
ser_ret_ch = df_ret['FRESH']
```

Examples of Conditions for Row/Column Extraction from DataFrame

df_part = df[**Cond.**]

- df['ColName']==Val1
  - Rows whose values in column "ColName" are consistent with Val1.
- df['ColName']>Val1
  - Rows whose values in column "ColName" are larger than Val1.
- df['ColName'].str.match(r'Regex')
  - Rows whose values in column "ColName" match with a regular expression "Regex".
- df['ColName'].isin(['Val1', 'Val2', …])
  - Rows whose values in column "ColName" are either Val1, Val2, …

Extract a Row Contains Maximum in a Column

Example： ser_max = df.loc[df['FRESH'].idxmax()]

Extract the row whose value is maximum in column FRESH.

```
CHANNEL       Horeca
REGION         Other
FRESH         112151
MILK           29627
GROCERY        18148
FROZEN         16745
DET_PAPER       4948
DELICA          8550
Name: 181, dtype: object
```

※df['ColName'].idxmax() returns the row index of first occurrence of maximum in Column "ColName", while idxmin() for minimum。

Reset Index

Example: df_10k = df[df['FRESH']>10000]

Only rows whose values in column FRESH are larger than 10000

Orignal df

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788
4	Retail	Other	22615	5410	7198	3915	1777	5185

After extraction, df_10k

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
3	Horeca	Other	13265	1196	4221	6404	507	1788
4	Retail	Other	22615	5410	7198	3915	1777	5185

Index become discontinuous due to row extractions that meet the conditions.

Row numbers used in .loc are original indices, but row numbers used in .iloc are new ones after data extraction from DataFrame counted from the beginning → reset index to avoid confusion.

Reset Index

df_10k_rstidx = df_10k**.reset_index()**

	index	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	0	Retail	Other	12669	9656	7561	214	2674	1338
1	3	Horeca	Other	13265	1196	4221	6404	507	1788
2	4	Retail	Other	22615	5410	7198	3915	1777	5185

Reset index by .reset_index(). Original indices are added as column 「index」

df_10k_rstidx2 = df_10k**.reset_index(drop=True)**

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Horeca	Other	13265	1196	4221	6404	507	1788
2	Retail	Other	22615	5410	7198	3915	1777	5185

When the original indices are not needed, use the option, drop=True

Modify a DataFrame Directly

If you want to modify the original data in a dataframe directly instead of assigning data to a new dataframe, set the option inplace to True, i.e., inplace=True.

Example: df_10k.reset_index(drop=True, **inplace=True**)

Instead of assigning data to a variable on the left, modify the index of df_10k directly.

List Unique Values and their Counts in Columns

Display unique values and their counts in a column of df thru df['ColName'].value_counts().

Example: print(df['REGION'].value_counts())

In df, display unique values and their frequencies in REGION

```
Other      316
Lisbon      77
Oporto      47
Name: REGION, dtype: int64
```

default in descending order.

Use nunique() instead of value_counts() If you want to obtain the number of unqiue values.

value_counts(option, option, …)

※ Sort in ascending order: ascending=True
※ Do not sort by values: sort=False
※ Include counts of NaN: dropna=False
※ Normalize the counts to 1: normalize=True

The Number of Entries that Meets the Conditions

(Cond.).sum()

Example: print(((df['CHANNEL']=='Retail') & (df['FRESH']>10000)).sum())

Display the number of entries (i.e., rows here) whose values in column CHANNEL are equal to "Retail" and larger than 10000 in column FRESH.

Grouping and Aggregation by Column Values

- Based on unique categorical values in a column, you can split data in the remaining columns into groups and compute maximum, minimum, mean, etc.

df2 = df.groupby(**'ColName'**)**.statistic_function()**

.statistic_function: max(), min(), mean() , etc.

Example: df_region_max = df.groupby(**'REGION'**)**.max()**

REGION	CHANNEL	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
Lisbon	Retail	56083	28326	39694	18711	19410	6854
Oporto	Retail	32717	25071	67298	60869	38102	5609
Other	Retail	112151	73498	92780	36534	40827	47943

Based on unique values in column REGION, it aggregate values of remaining columns by max().

※ It is possible to aggregate for multiple columns when using a list of 'column names'.

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

For example, regarding column REGION, how to compute mean of columns when values in column REGION are "Other" ?

Group column data based on values in REGION thru groupby(), and then process data with statistical functions, such as mean(), etc, to compute mean, sum, standard deviation, etc. for each category all at once.

Cross Tabulation

- Classify all data based on two categorical columns and compute counts of each category.

df2 = **pd**.crosstab(**Series of Col1, Series of Col2**)

Example: df_reg_ch = **pd**.crosstab(**df['REGION'], df['CHANNEL']**)

CHANNEL	Horeca	Retail
REGION		
Lisbon	59	18
Oporto	28	19
Other	211	105

Based on unique values in REGION, aggregate the other column to find their counts.

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

For example, regarding column CHANNEL and REGION, how to find the number of customers whose values are equal to "Retail" column in CHANNEL and "other" in column REGION?

Use crosstab() to split customers by two columns and compute the number of customers.

Cross Tabulation

- Add row/column subtotals: margins=True
- Normalize values where the summation is 1
  - Column-wise summation becomes 1: normalize='columns',
    Row-wise summation becomes 1: normalize='index'
  - Summation of all cells becomes 1: normalize='all'

```
df_reg_ch2 = pd.crosstab(df['REGION'], df['CHANNEL'],
                         margins=True, normalize='columns')
```

Column-wise summation is 1

CHANNEL / REGION	Horeca	Retail	All
Lisbon	0.197987	0.126761	0.175000
Oporto	0.093960	0.133803	0.106818
Other	0.708054	0.739437	0.718182

Column-wise subtotals

※When normalize='columns' that compute column-wise subtotal, row-wise subtotal is not added.

Pivot Table

- Similar to cross tabulation, classify all data based on two categorical column. For aggregation, compute not only counts, but also statistics such as average, maximum, minimum etc.

```
df2 = df.pivot_table(index='Col1',  columns='Col2',
                     values=['Col3', …],  aggfunc=np.mean)
```

Example:

```
df_pivot = df.pivot_table(index='REGION',  columns='CHANNEL',
                          values=['FRESH', 'FROZEN'])
```

- index, columns can be assigned with a list of multiple column names (In particular, it is better to specify the column you want to keep in the output dataframe with index option)
- Without specifying "values=", all columns will be used.
- Without specifying "aggfunc=", np.mean will be used. Besides this, you can specify a function to calculate statistics such as np.max. Note that it only require the function name without parentheses, ().

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

For example, how to compute mean for FRESH and FROZEN where their values are "Horeca" in CHANNEL and "Lisbon" in REGION ?

Use pivot_table() to split customers into groups by values in CHANNEL and REGION and compute mean, summation, standard deviation, etc., all at once.

	FRESH		FROZEN	
CHANNEL	Horeca	Retail	Horeca	Retail
REGION				
Lisbon	12902.254237	5200.000000	3127.322034	2584.111111
Oporto	11650.535714	7289.789474	5745.035714	1540.578947
Other	13878.052133	9831.504762	3656.900474	1513.200000

Delete Rows/Columns

- Delete a column

  df2 = df.drop(columns='ColName') or df2 = df.drop('ColName', axis=1)

- Delete a row

  df2 = df.drop(index=RowNum) or df2 = df.drop(RowNum)

You can also specify multiple row numbers and column names in a list or slice.

Example: df_del = df.drop(**df.loc[:, 'FRESH':'GROCERY'], axis=1**)

Delete rows of df which starts **from** FRESH **to** GROCERY and assign the result to variable df_del.

Example: df_del = df.drop(**df.index[1:3]**)

Delete rows of df **from** row 1 **to** row 3 and assign the result to variable df_del.

Example: df_del = df.drop(**index=[1, 3], columns=['FRESH', 'GROCERY']**)

Delete row 1 and row 3, and column FRESH **and** column GROCERY, and then assign the result to variable df_del.

Delete Rows Extracted by a Certain Value in a Column

Example: df2 = df.drop(df.index[df['FRESH'] > 10000])

Delete rows whose values in FRESH are larger than 10000, and assign the result to variable df2.

df

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Horeca	Other	13265	1196	4221	6404	507	1788

df2

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
5	Retail	Other	9413	8259	5126	666	1795	1451

Delete Rows Extracted by a Certain Value in a Column

Another solution: extract rows whose values in FRESH column **are not** larger than 10000 and assign them to df2

```
df2 = df[ ~(df['FRESH'] > 10000) ] )
```

df2

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
5	Retail	Other	9413	8259	5126	666	1795	1451

(The same results as using drop)

Adding a Column to DataFrame

- Adding a column

  df['**New Column Name**'] = **List or Series**

Example: df2 = df.iloc[:3].copy()

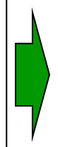
Copy data starting from the first row to row 2 in df by a slice (**:3**), and assign it to variable df2.

df2['**new_col**'] = [2, 4, 6]

or

df2['**new_col**'] = pd.Series([2, 4, 6])

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844



	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA	new_col
0	Retail	Other	12669	9656	7561	214	2674	1338	2
1	Retail	Other	7057	9810	9568	1762	3293	1776	4
2	Retail	Other	6353	8808	7684	2405	3516	7844	6

Adding a Row to DataFrame

- Adding a row by method **append**
  1. Make the index of added row data same as that of columns of DataFrame.
  2. Regarding Index of DataFrame after addition, the name attribute of Series is inherited, but if not necessary, it is recommended to automatically generate using ignore_index=True.

Let df be DataFrame, ser be Series

1. ser = pd.Series([…], **index=df.columns**)

2. df2 = df.**append**(ser, **ignore_index=True**)

Adding a Row to DataFrame

Example: df2 = df.iloc[:3].copy()

Copy data in df from the first row to row 2, and assign it to variable df2.

```
ser = pd.Series(['Retail','Other',0,1,2,3,4,5], index=df2.columns)
df3 = df2.append(ser, ignore_index=True)
```

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Retail	Other	0	1	2	3	4	5

Row with a serial number "3" is automatically added by setting ignore_index=True.

Create a New Column from the Existing Column

Example: Add a new column DELICA5000 which has a value of "L"
if the value in DELICA is larger than or equal to 5000, otherwise "S"

```
df['DELICA5000'] = df['DELICA'].map(lambda x: 'L' if L>=5000 else 'S')
```

※ Relationship of lambda(anonymous function) and def function

```
def func_name(arg1, arg2, …):
    …
    return  expression
```

⟺

```
lambda arg1, arg2, … : expression
```

※ Add a new column by an operation among columns, such as below.

Example: df['new'] = 2 * df['FRESH'] + df['MILK'] − 10000

Concatenation between DataFrames

- First, prepare the original data

```
df1 = pd.DataFrame([[0,1,2],[3,4,5]], columns=['c1','c2','c3'])
df2 = pd.DataFrame([[1,3,5],[2,4,6]], columns=['c4','c5','c6'])
df3 = pd.DataFrame([[9,8,7],[6,5,4]], columns=['c1','c2','c3'], index=[100,101])
df4 = pd.DataFrame([[8,6,4],[9,7,5]], columns=['c4','c5','c6'], index=[100,101])
```

df1

	c1	c2	c3
0	0	1	2
1	3	4	5

df2

	c4	c5	c6
0	1	3	5
1	2	4	6

df3

	c1	c2	c3
100	9	8	7
101	6	5	4

df4

	c4	c5	c6
100	8	6	4
101	9	7	5

df12 = pd.concat([df1,df2], axis=1) Because their index are same, concatenate them along the horizontal direction with axis = 1.

df13 = pd.concat([df1,df3]) Because their columns are in common, concatenate them along the vertical direction.

df13_2 = pd.concat([df1,df3], ignore_index=True) index is reset.

df12

	c1	c2	c3	c4	c5	c6
0	0	1	2	1	3	5
1	3	4	5	2	4	6

df13

	c1	c2	c3
0	0	1	2
1	3	4	5
100	9	8	7
101	6	5	4

df13_2

	c1	c2	c3
0	0	1	2
1	3	4	5
2	9	8	7
3	6	5	4

Because index is not consistent, concatenate the second as new rows (where No value is replaced with NaN.)
df14 = pd.concat([df1,df4], axis=1)

Concatenate them in the horizontal direction if index is aligned.
df4.index=df1.index
df14_2 = pd.concat([df1,df4], axis=1)

df14

	c1	c2	c3	c4	c5	c6
0	0.0	1.0	2.0	NaN	NaN	NaN
1	3.0	4.0	5.0	NaN	NaN	NaN
100	NaN	NaN	NaN	8.0	6.0	4.0
101	NaN	NaN	NaN	9.0	7.0	5.0

df14_2

	c1	c2	c3	c4	c5	c6
0	0	1	2	8	6	4
1	3	4	5	9	7	5

Replacement of Element Values

Enable regular expression mode.

Regular expression mode is same as that of re.sub(). Replace all occurrences even if part of the string matches "Old".

ser = df['Col'].replace(Old, New, regex=True)

Column after replacement

Example: df2['REGION'] = df['REGION'].replace('Other', 'OTHER')

Replace all "Other" in REGION with "OTHER"

Example: df2['REGION'] = df['REGION'].replace('Oth', 'OTH', regex=True)

Replace "Oth" of values in REGION with "OTH"
(even part of the string matches)

When substituting the same value into multiple cells that satisfy the condition,

Example: df2.at[df2['FRESH']>9000, 'FRESH'] = 9000

Substituting cell values which are larger than 9000 in FRESH with 9000.

Rearrangement (sort)

df2 = df.sort_values(by=['Col1', 'Col2', …],
ascending=[True/False, True/False, …])

For ascending=[…] , corresponding to Col1, Col2, … from the beginning, sort by the Boolean values where it is True for ascending order and False for descending order.

Example: df2 = df.sort_values(by=['CHANNEL', 'FRESH'],
ascending=[True, False])

The 1 st key : ascending order in CHANNEL (in alphabetical order)
The 2 nd key : descending order in FRESH by their quantities.

Convert Categorical Variable into Dummy Variables

- Categorical variable(Qualitative variable): a variable that can take on one of discrete(limited) values, but not numeric values, such as,「affiliation」,「marital status」etc. In the CSV data we are dealing with, column REGION and column CHANNEL are categorical variables.
- For data analysis, such as regression analysis, it is necessary to quantify categorical variables. When performing numerical transformation for REGION with 0, 1, and 2, it can be misleading that 0 in REGION, as a numerical digit, is close to 1 in REGION than 2 in REGION. To avoid this, generate N-1 **dummy variables** of N categorical variable those are represented with 0 or 1.

REGION	
Lisbon	0
Oporto	1
Other	2

REGION	REGION_Oporto	REGION_Other
Lisbon	0	0
Oporto	1	0
Other	0	1

※For Lisbon, REGION_Oporto and REGION_Other are set to 0, but not create REGION_Lisbon.

Convert Categorical Variable into Dummy Variables

df2 = pd.get_dummies(df, **drop_first=True**)

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

It is an option with which the dummy variables, REGION_Lisbon and CHANNEL_Horeca, are not created.

	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA	CHANNEL_Retail	REGION_Oporto	REGION_Other
0	12669	9656	7561	214	2674	1338	1	0	1
1	7057	9810	9568	1762	3293	1776	1	0	1
2	6353	8808	7684	2405	3516	7844	1	0	1

Compute Correlation Coefficients among All Pairs

- df2 = df.corr()
  - Exclude columns of values other than numeric or bool.
  - For Boolean columns, use 1 for True, 0 for False.
  - Exclude missing values (NaN)
  - "pearson" is used for method option in default if not specified. Besides it, kendall or spearman is available.

	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
FRESH	1.000000	0.100510	-0.011854	0.345881	-0.101953	0.244690
MILK	0.100510	1.000000	0.728335	0.123994	0.661816	0.406368
GROCERY	-0.011854	0.728335	1.000000	-0.040193	0.924641	0.205497
FROZEN	0.345881	0.123994	-0.040193	1.000000	-0.131525	0.390947
DET_PAPER	-0.101953	0.661816	0.924641	-0.131525	1.000000	0.069291
DELICA	0.244690	0.406368	0.205497	0.390947	0.069291	1.000000

For example, the value underlined in blue that can be retrieved as below,

```
df2.at['FROZEN', 'GROCERY']
```

Compute Correlation Coefficients: one versus others, one versus one

For Dataframe versus Series, one can do with DataFrame.corrwith(Series).

- ser2 = df.corrwith(df['FRESH'])

```
FRESH          1.000000
MILK           0.100510
GROCERY       -0.011854
FROZEN         0.345881
DET_PAPER     -0.101953
DELICA         0.244690
dtype: float64
```

For example, the value underlined in blue that can be retrieved as below,

ser2['FROZEN']

For Series versus Series, one can do with Series.corr(Series) .

- corr3 = df['FROZEN'].corr(df['FRESH'])

  → 0.3458814...

Save a DataFrame as a CSV File

Example: df.to_csv('output CSV file',
index=False, encoding='shift-jis')

If index=False, do not write index in a CSV file.

Specify the kanji encoding to use in the output file by encoding option. (default is utf-8)

To outputs certain columns, use the option as below,
columns=['Col1', 'Co2', …]

Datetime Type Conversion

2045_2017-utf8.csv source from 株式投資メモ(https://kabuoji3.com/stock/)

	Date	Open	High	Low	Close	Vol	AdjClose
0	2017-01-04	9340	9340	9270	9270	6	9270
1	2017-01-05	9260	9340	9250	9260	22	9260
2	2017-01-06	9300	9380	9300	9380	69	9380

```
df_sreit=pd.read_csv('2045_2017-utf8.csv',
                     delimiter=',', skiprows=9, header=0)
print(df_sreit['Date'].dtype) → Object (Read as string)

df_sreit['Date']=pd.to_datetime(df_sreit['Date'], format="%Y-%m-%d")
print(df_sreit['Date'].dtype) → datetime64 (Convert to datetime type)
```

Various Time Series Data

- Add a new column, "The 3-Day Moving Average"
  df_sreit['mov_ave3'] =
  df_sreit['AdjClose'].rolling(window=3).mean()
  ※ The first two will have no value (NaN)
- Aggregate values for each column per month
  df_sreit_date = df_sreit.set_index('Date')
  df_sreit_monthly = df_sreit_date.resample('M').mean()

	Date	Open	High	Low	Close	Vol	AdjClose	mov_ave3
0	2017-01-04	9340	9340	9270	9270	6	9270	NaN
1	2017-01-05	9260	9340	9250	9260	22	9260	NaN
2	2017-01-06	9300	9380	9300	9380	69	9380	9303.333333
3	2017-01-10	9380	9390	9320	9370	465	9370	9336.666667
4	2017-01-11	9400	9520	9400	9490	233	9490	9413.333333

Date	Open	High	Low	Close	Vol	AdjClose	mov_ave3
2017-01-31	9390.769231	9421.538462	9362.307692	9387.692308	113.384615	9387.692308	9398.484848
2017-02-28	9542.631579	9575.263158	9522.105263	9557.368421	112.263158	9557.368421	9549.649123
2017-03-31	9651.500000	9683.000000	9626.500000	9655.500000	1536.300000	9655.500000	9645.000000

Aggregation per month

Various Time Series Data

That is, taking closing price of the first day as 1, it is represented as a ratio of closing price of each day to that of the first day.

- For column "AdjClose", normalize values based on the first one and then add them into a new column "AdjClose_ratio"

```
df_sreit['AdjClose_ratio'] = df_sreit['AdjClose'] / df_sreit.at[0, 'AdjClose']
```

- Compute the number of days away from 2017/1/1 (2017/1/1 as the first day) and add them to a new column "ndays2017"

```
day1 = pd.to_datetime('2017-01-01')
dayw = np.timedelta64(1, 'D')
df_sreit['ndays2017'] = (df_sreit['Date']-day1) / dayw + 1
```

Get the number of days by dividing "1 day"

(Another solution)

```
df_sreit['ndays2017'] = (df_sreit['Date']-day1).dt.days + 1
```

Get the number of days directly

Merge DataFrames

1393_2017-utf8.csv (data: 201 rows)

	1393 TSE ETF UBS ETF US stocks (MSCI USA) (ETF)			
1	Date(日付)			
2	Open: Opening price (始値)			
3	High: High price (高値)			
4	Low: Low price (安値)			
5	Close: Closing price (終値)			
6	Vol: Trading volume (出来高)			
7	CloseAdj: Adjusted closing price (終値調整値)			
8				
9	Date	Open	High	Low
10	2017-01-04	25500	25520	25410
11	2017-01-05	25520	25520	25200
12	2017-01-06	25150	25170	25150
13	2017-01-10	25170	25170	25010
14	2017-01-11	25040	25040	25040
15	2017-01-12	24980	24980	24880

2045_2017-utf8.csv (data: 139 rows)

	2045 TSE ETF NEXT NOTES S&P Singapore REIT (NR)ETN (ETF)			
1	Date(日付)			
2	Open: Opening price (始値)			
3	High: High price (高値)			
4	Low: Low price (安値)			
5	Close: Closing price (終値)			
6	Vol: Trading volume (出来高)			
7	CloseAdj: Adjusted closing price (終値調整値)			
8				
9	Date	Open	High	Low
10	2017-01-04	9340	9340	9270
11	2017-01-05	9260	9340	9250
12	2017-01-06	9300	9380	9300
13	2017-01-10	9380	9390	9320
14	2017-01-11	9400	9520	9400
15	2017-01-12	9450	9550	9450

There are no common in column "Date" of both data, i.e., a date in one side but not in the other one). Taking this into account, the operation of joining these two tables is called "Merge". The names of common columns considered in the merge is called "Key".

Merge two DataFrames based on the value of column "Date" as the key.

```
df_sreit=pd.read_csv('2045_2017-utf8.csv',
                     delimiter=',', skiprows=9, header=0)
df_sreit['Date']=pd.to_datetime(df_sreit['Date'], format="%Y-%m-%d")

df_ubs=pd.read_csv('1393_2017-utf8.csv',
                     delimiter=',', skiprows=9, header=0)
df_ubs['Date']=pd.to_datetime(df_ubs['Date'], format="%Y-%m-%d")
```

Left table　　Name of key column

Right table

```
df_merged = pd.merge(df_sreit, df_ubs, on='Date', how=TypesOfMerge)
```

Types of merge

※ When key names are different in the left/right table, use left_on= and right_on=, respectively.

'inner': inner join(only extract common dates between left/right tables and merge)
'outer': outer join(In merged table, dates present in at least one of the left and right tables are displayed 。If no match found, fill the cell with NaN)
'left': all dates in the left table are displayed. If no match found, fill the cell with NaN.
'right': all dates in the right table are displayed. If no match found, fill the cell with NaN.

Types of Merge

df1

列11	列12	列13
'A'	1	5
'C'	2	6
'D'	3	7
'E'	4	8

key

df2

列21	列22	列23
'B'	10	50
'C'	20	60
'E'	30	70
'F'	40	80

key

```
df3 = pd.merge(df1, df2,
               left_on='列11',
               right_on='列21',
               how=マージ方法)
```

how='inner'

列11	列12	列13	列21	列22	列23
'C'	2	6	'C'	20	60
'E'	4	8	'E'	30	70

how='outer'

列11	列12	列13	列21	列22	列23
'A'	1	5	NaN	NaN	NaN
'C'	2	6	'C'	20	60
'D'	3	7	NaN	NaN	NaN
'E'	4	8	'E'	30	70
NaN	NaN	NaN	'B'	10	50
NaN	NaN	NaN	'F'	40	80

Types of Merge

df1

列11	列12	列13
'A'	1	5
'C'	2	6
'D'	3	7
'E'	4	8

キー

df2

列21	列22	列23
'B'	10	50
'C'	20	60
'E'	30	70
'F'	40	80

キー

```
df3 = pd.merge(df1, df2,
                left_on='列11',
                right_on='列21',
                how=マージ方法)
```

how='left'

列11	列12	列13	列21	列22	列23
'A'	1	5	NaN	NaN	NaN
'C'	2	6	'C'	20	60
'D'	3	7	NaN	NaN	NaN
'E'	4	8	'E'	30	70

how='right'

列11	列12	列13	列21	列22	列23
'C'	2	6	'C'	20	60
'E'	4	8	'E'	30	70
NaN	NaN	NaN	'B'	10	50
NaN	NaN	NaN	'F'	40	80

Example: how='inner'

If the column name is the same in left and right tables, _x, _y will be added

Data in the left table ("_x" is attached)　Data in the right table ("_y" is attached)

	Date	Open_x	High_x	Low_x	Close_x	Vol_x	AdjClose_x	Open_y	High_y	Low_y	Close_y	Vol_y	AdjClose_y
0	2017-01-04	9340	9340	9270	9270	6	9270	25500	25520	25410	25520	21	25520
1	2017-01-05	9260	9340	9250	9260	22	9260	25520	25520	25200	25220	71	25220
2	2017-01-06	9300	9380	9300	9380	69	9380	25150	25170	25150	25170	3	25170

Missing Values

- Right-click pandas_training_missing-utf8.csv listed on the left sidebar of Jupyter, select Open With > Editor

```
 9  GROCERY(食料品): annual spending (m.u.)on grocery products (Cont
10  FROZEN(冷凍): annual spending (m.u.)on frozen products (Continuo
11  DETERGENTS_PAPER(洗剤・紙類): annual spending (m.u.) on detergent
12  DELICATESSEN(惣菜): annual spending (m.u.)on and delicatessen pr
13  ,,,,,,,
14  CHANNEL,REGION,FRESH,MILK,GROCERY,FROZEN,DET_PAPER,DELICA
15  Retail,Other,n/a,9656,7561,n/a,2674,1338
16  Retail,Other,7057,9810,9568,1762,3293,n/a
17  Retail,Other,6353,8808,7684,2405,3516,7844
18  Horeca,Other,13265,1196,4221,6404,507,1788
19  Retail,Other,22615,5410,7198,3915,1777,5185
```

Missing Values

```
df_missing = pd.read_csv('pandas_training_missing-utf8.csv',
                         delimiter=',', skiprows=13, header=0)
```

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	NaN	9656	7561	NaN	2674	1338.0
1	Retail	Other	7057.0	9810	9568	1762.0	3293	NaN
2	Retail	Other	6353.0	8808	7684	2405.0	3516	7844.0

Check for missing values

```
print( df_missing.isnull().sum() )
```

```
CHANNEL      0
REGION       0
FRESH        1
MILK         0
GROCERY      0
FROZEN       1
DET_PAPER    0
DELICA       1
dtype: int64
```

Display the number of missing values of each column.

```
display( df_missing[ df_missing.isnull().any(axis=1) ] )
```

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	NaN	9656	7561	NaN	2674	1338.0
1	Retail	Other	7057.0	9810	9568	1762.0	3293	NaN

Display rows containing missing values.

Delete Missing Values

Example: df_del = df_missing.**dropna**(axis=0)

Delete rows containing missing of df_missing and assign the result to df_del.

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
2	Retail	Other	6353.0	8808	7684	2405.0	3516	7844.0
3	Horeca	Other	13265.0	1196	4221	6404.0	507	1788.0
4	Retail	Other	22615.0	5410	7198	3915.0	1777	5185.0

※ When resetting the index, use .reset_index(drop=True)

Example: df_del = df_missing.**dropna**(axis=0).reset_index(drop=True)

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	6353.0	8808	7684	2405.0	3516	7844.0
1	Horeca	Other	13265.0	1196	4221	6404.0	507	1788.0
2	Retail	Other	22615.0	5410	7198	3915.0	1777	5185.0

Fill Missing Values

Example: df_filled = df_missing.**fillna**(df_missing.**mean()**)

Fill the missing values of each column with **the mean of the column** in df_missing and assign the result to df_filled.

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	11998.774487	9656	7561	3078.441913	2674	1338.000000
1	Retail	Other	7057.000000	9810	9568	1762.000000	3293	1524.298405
2	Retail	Other	6353.000000	8808	7684	2405.000000	3516	7844.000000

Extract/Delete duplicate rows

DataFrame
df_dup

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844
3	Retail	Other	7057	9810	9568	1762	3293	1776

Example: display(df_dup[df_dup.duplicated()])

Display the duplicated rows

Example: df_nodup = df_dup.drop_duplicates()

Drop the duplicates except for the first occurrence by default.

DataFrame
df_nodup

	CHANNEL	REGION	FRESH	MILK	GROCERY	FROZEN	DET_PAPER	DELICA
0	Retail	Other	12669	9656	7561	214	2674	1338
1	Retail	Other	7057	9810	9568	1762	3293	1776
2	Retail	Other	6353	8808	7684	2405	3516	7844

Other Tips

- In Pandas, if displaying a dataframe with display() when data size is huge, rows and columns are replaced with "…". Without omission, one can set the maximum number of columns and rows displayed to a large value (for example, 999).
  - The maximum number of columns displayed:
    pd.options.display.max_columns = 999
  - The maximum number of rows displayed :
    pd.options.display.max_rows = 999

(Adv) Other Tips

- In a notebook, when measuring the execution time of a cell,
  - Add "%%time" to the top of the cell as a line
- In a notebook, when measuring the execution time of one-line python program,
  - Add "% time (blank)" at the beginning of the line
  - When you want to run the program multiple times and display the maximum time and average time of executions, add "% timeit (blank)" at the beginning of the line.

Other Tips

- For example, if you want to read the specification of arguments of pd.read_csv(),

?pd.read_csv

```
Signature:
pd.read_csv(
    filepath_or_buffer,
    sep=',',
    delimiter=None,
    header='infer',
    names=None,
    index_col=None,
    usecols=None,
        ⋮
    memory_map=False,
    float_precision=None,
)
Docstring:
Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <http://pandas.pydata.org/pandas-docs/stable/io.html>`_.

Parameters
----------
filepath_or_buffer : str, path object, or file-like object
    Any valid string path is acceptable. The string could be a URL. Valid
    URL schemes include http, ftp, s3, and file. For file URLs, a host is
    expected. A local file could be: file://localhost/path/to/table.csv.
        ⋮
```

Other Tips

- For example, if you want to read the specification of arguments of pd.read_csv(),

```
help(pd.read_csv)
```

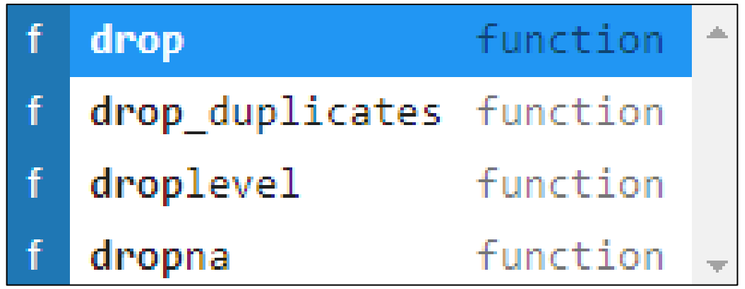
```
Help on function read_csv in module pandas.io.parsers:

read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, in
dex_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype
=None, engine=None, converters=None, true_values=None, false_values=None, skipinitia
lspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_
na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, in
fer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, it
erator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lin
eterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comme
nt=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn
_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_pr
ecision=None)
    Read a comma-separated values (csv) file into DataFrame.

    Also supports optionally iterating or breaking of the file
    into chunks.

    Additional help can be found in the online docs for
    `IO Tools <http://pandas.pydata.org/pandas-docs/stable/io.html>`_.

    Parameters
    ----------
    filepath_or_buffer : str, path object, or file-like object
        Any valid string path is acceptable. The string could be a URL. Valid
```

Other Tips

- If forgetting method name (function) or attribute name

```
df = df.dro
```

Here, press **TAB key**



A list of method (function) names and attribute names of dataframes starting with "dro" is displayed. Then, you can select it from the list.

Other Tips

- Copy and paste function/attribute names, etc., with mouse

```
df_nodup = df_dup.drop_duplicates()
```

If double-clicking on it, "df_nodup" will be highlighted and then you can easily copy it by right-click menu.

Why use Numpy/Pandas Functions ?

Ans: they are faster than calculating thru for loop

Example:

```
%%time
a_for = np.empty( (5000, 5000) )
for  i   in  range(5000):
    for  j  in  range(5000):
        a_for[i, j] = 1.0
```

Wall time: 4.55 s

```
%%time
b_np = np.ones( (5000, 5000) )
```

Wall time: 99 ms

Execution time changes at each execution.

Therefore, you may first check if there were a Numpy/Pandas function implements the analysis you want to do, and use it if available.

Operations in Jupyter are compared with Excel ?

Of course, Excel also provide quick visualization of relatively small data being useful for simple analysis.

- No need to display data itself on the sheet.
  - The same code can be used to do analysis even if the CSV file contains tens of thousands lines.
- Data can be handled in row-wise and/or column-wise way
  - No need to click on individual cells (probably making mistakes)
  - "automatically change the cell reference in the formula" being aware of individual cells (which is prone to mistakes) is not needed.
- Formulas are all "visible" in the program
  - No need to click on individual cells to check the formula.
- It may be worried that one can not see the data stored in variables
  You can use print(df.shape), print(df.info()), display(df.head()), display(df.describe()), etc.