8. Pandas

Chapter Content

- Introduction
- Installation
- DataFrame
- Import and Export data
- Filtering & Selection of a DataFrame
- Missing Data Handling
- Reshaping DataFrame
- Interacting with Database
- Df.groupby
- Add row data
- Df.sort_values
- Df.set_index, df.reset_index
- Change column order
- Series.unique, series.nunique
- Pd.read_csv, pd.read_json
- Time Series
- Categorical Data Type
- Df.rolling
- Df.where, df.mask

Introduction

- Pandas the name derived from Panel Data and Python Data Analysis
- World-class open-source project
- Library written for data manipulation and analysis
- Strength in DataFrame for integrated manipulation
- Adopts significant part of Numpy style array computing
- Well associate with Numpy, SciPy, Scikit-learn, Matplotlib, etc

Installation

In your iPython or Jupyter or Terminal

```
pip install pandas
```

Check version

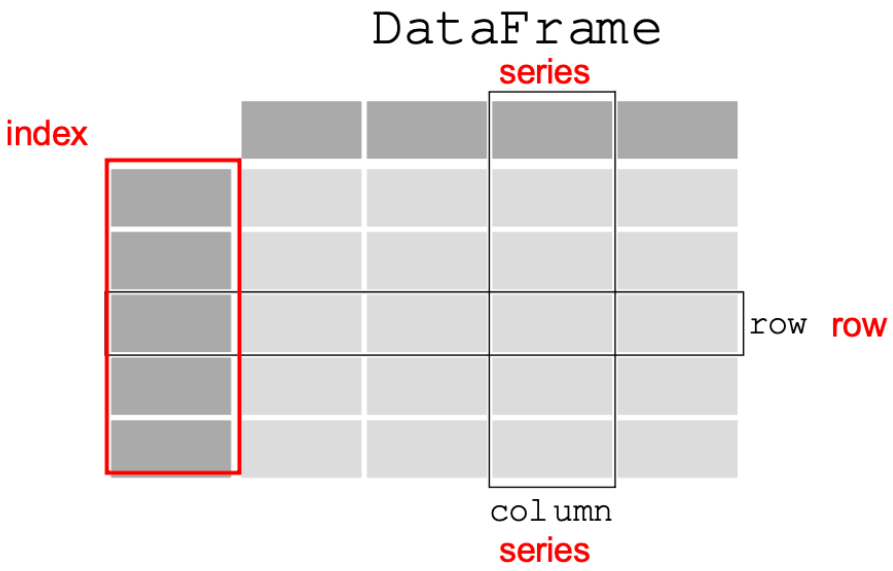
```
import pandas as pd
pd.__version__
```

```
'1.5.2'
```

General terminology translation

General terminology translation

pandas	Excel
`DataFrame`	worksheet
`Series`	column
`Index`	row headings
row	row
`NaN`	empty cell



A **Series** is a 1-Dimension np.array-like object containing a sequence of values and an associated array of data labels (index).

```
obj = pd.Series([3.4, 2.5, 6.8, 4.9, 5.8])
obj
```

```
0    3.4
1    2.5
2    6.8
3    4.9
4    5.8
dtype: float64
```

A **DataFrame** is 2-Dimensional, size-mutable, potentially heterogeneous tabular data.

In [2]:
```
df = pd.DataFrame(
    {
        "Name": ["Alice", "Bobby", "Charlie", "Dorathy" ],
        "Gender": ["female", "male", "male", "female"],
        "Age": [25, 30, 58, 42],
        "Height": [1.68, 1.72, 1.78, 1.63]
    }
)
df
```

Out[2]:

	Name	Gender	Age	Height
0	Alice	female	25	1.68
1	Bobby	male	30	1.72
2	Charlie	male	58	1.78
3	Dorathy	female	42	1.63

Simple DF Manipulation

```
In [3]: df["Name"]
```

```
Out[3]: 0       Alice
        1       Bobby
        2     Charlie
        3     Dorathy
        Name: Name, dtype: object
```

```
In [4]: df["Height"].max()
```

```
Out[4]: 1.78
```

```
In [5]: df["Age"].mean()
```

```
Out[5]: 38.75
```

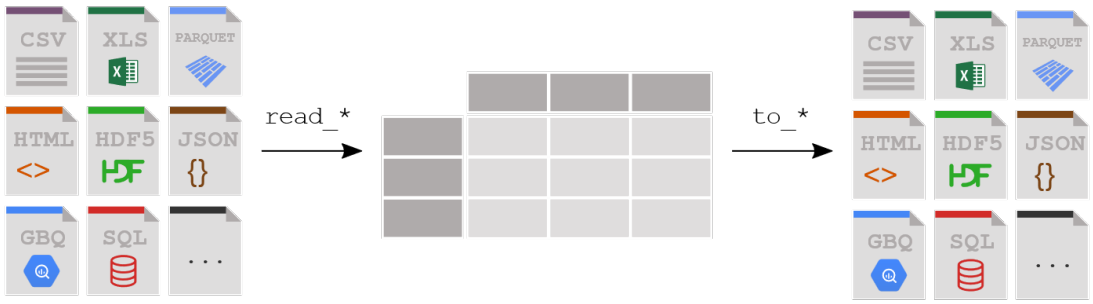
```
In [6]: df.describe()
```

Out[6]:

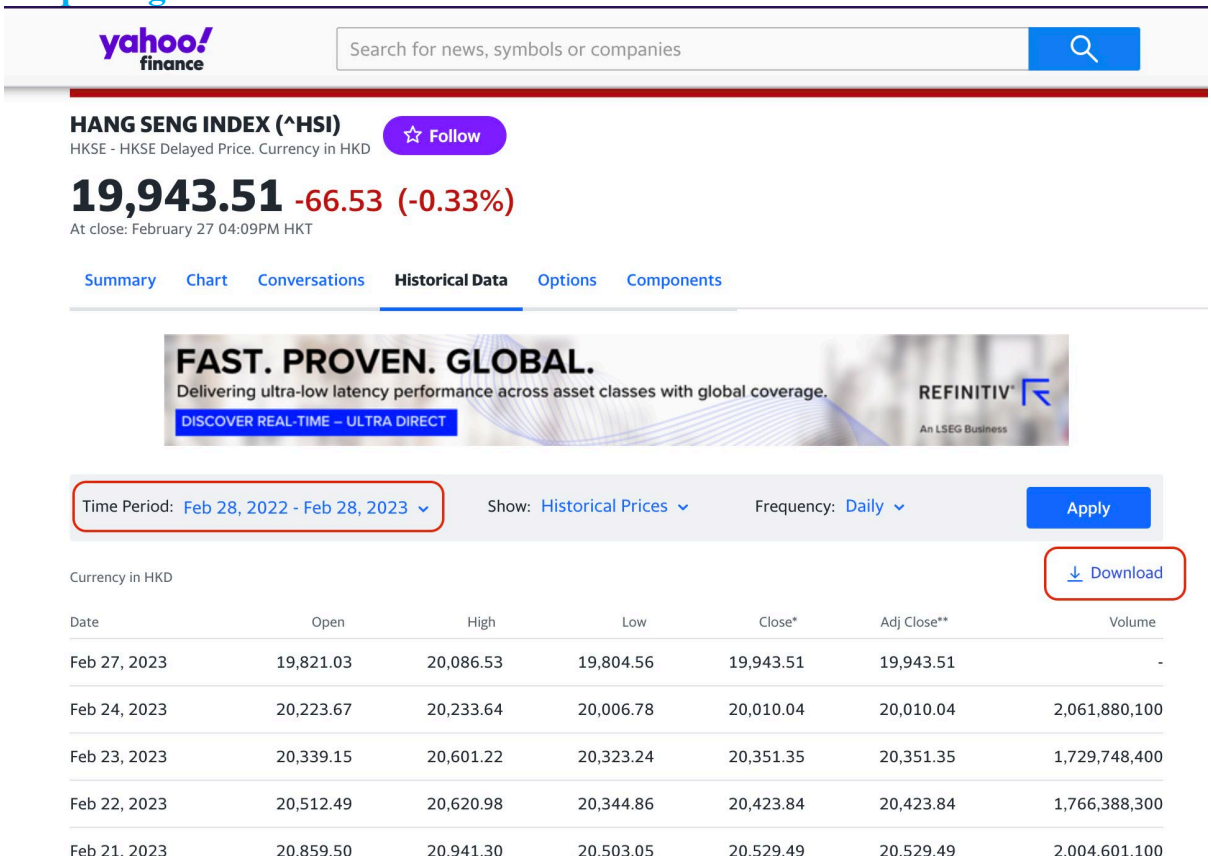
	Age	Height
count	4.000000	4.000000
mean	38.750000	1.702500
std	14.682756	0.063443
min	25.000000	1.630000
25%	28.750000	1.667500
50%	36.000000	1.700000
75%	46.000000	1.735000
max	58.000000	1.780000

Import and export data

- In most of the case, we import data to build DataFrame from existing dataset
- For learning purpose, we use .csv often



Preparing a csv dataset



Date	Open	High	Low	Close*	Adj Close**	Volume
Feb 27, 2023	19,821.03	20,086.53	19,804.56	19,943.51	19,943.51	-
Feb 24, 2023	20,223.67	20,233.64	20,006.78	20,010.04	20,010.04	2,061,880,100
Feb 23, 2023	20,339.15	20,601.22	20,323.24	20,351.35	20,351.35	1,729,748,400
Feb 22, 2023	20,512.49	20,620.98	20,344.86	20,423.84	20,423.84	1,766,388,300
Feb 21, 2023	20,859.50	20,941.30	20,503.05	20,529.49	20,529.49	2,004,601,100

Import .csv file into Pandas DataFrame

- Save the .csv in the same folder of your notebook .ipynb

```
In [7]: df_hsi = pd.read_csv("^HSI.csv")
        df_hsi
```

Out[7]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2022-01-03	23510.539063	23605.029297	23193.189453	23274.750000	23274.750000	734331100
1	2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200
2	2022-01-05	23323.769531	23323.769531	22851.500000	22907.250000	22907.250000	2768859000
3	2022-01-06	22843.199219	23082.949219	22709.599609	23072.859375	23072.859375	1765786100
4	2022-01-07	23318.919922	23497.500000	23162.849609	23493.380859	23493.380859	2602290600
...	...	...	...	...	...	...	...
241	2022-12-22	19537.449219	19735.000000	19475.679688	19679.220703	19679.220703	1939795100
242	2022-12-23	19382.230469	19686.769531	19380.470703	19593.060547	19593.060547	1363741800
243	2022-12-28	19787.939453	20099.769531	19787.939453	19898.910156	19898.910156	2823780600
244	2022-12-29	19648.400391	19764.519531	19539.839844	19741.140625	19741.140625	2902362900
245	2022-12-30	20030.849609	20073.919922	19781.410156	19781.410156	19781.410156	1747706800

246 rows × 7 columns

Import data from yfinance

Alternatively, we download the dataset via yfinance package.

Please note that yfinance is not Yahoo's affiliate, it is the work of warm-hearted geeks

```
pip install yfinance
```

```
Collecting yfinance
  Downloading yfinance-0.2.12-py2.py3-none-any.whl (59 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 59.2/59.2 kB 54
Requirement already satisfied: numpy>=1.16.5 in /Users/honcyl
rom yfinance) (1.23.1)
Requirement already satisfied: pandas>=1.3.0 in /Users/honcyl
rom yfinance) (1.5.2)
Collecting multitasking>=0.0.7
```

In [9]:
```
import yfinance as yf
df_HSI = yf.download("^HSI", start="2022-01-01", end="2023-01-01")
```

```
[*********************100%***********************]  1 of 1 completed
```

In [10]:
```
df_HSI
```

Out[10]:

Date	Open	High	Low	Close	Adj Close	Volume
2022-01-03	23510.539062	23605.029297	23193.189453	23274.750000	23274.750000	734331100
2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200
2022-01-05	23323.769531	23323.769531	22851.500000	22907.250000	22907.250000	2768859000
2022-01-06	22843.199219	23082.949219	22709.599609	23072.859375	23072.859375	1765786100
2022-01-07	23318.919922	23497.500000	23162.849609	23493.380859	23493.380859	2602290600
...	...	...	...	...	...	...
2022-12-22	19537.449219	19735.000000	19475.679688	19679.220703	19679.220703	1939795100
2022-12-23	19382.230469	19686.769531	19380.470703	19593.060547	19593.060547	1363741800
2022-12-28	19787.939453	20099.769531	19787.939453	19898.910156	19898.910156	2823780600
2022-12-29	19648.400391	19764.519531	19539.839844	19741.140625	19741.140625	2902362900
2022-12-30	20030.849609	20073.919922	19781.410156	19781.410156	19781.410156	1747706800

246 rows × 6 columns

In [11]:
```
print(type(df_hsi))
print(type(df_HSI))
```

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
```

More basic tools for initiate a DataFrame

DF.head( )

DF.tail( )

DF.shape

DF.info( )

DF.to_datetime( )

DF.set_index( )

DF.reset_index( )

DF.describe( )

DF.isna( )

DF.isna( ).values.any( )

Can you guess why do we need an index?

Filtering & Selection of a DataFrame

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position

```
df_hsi['Close']      # select as pd Series
df_hsi[['Close']]    # Select a pd DF
df_hsi[['Close','Volume']]   # Select two columns as pd DF
df_hsi.loc[df_hsi['Close'] > df_hsi['Open']]   # select with condition
df_hsi.loc[df_hsi.index >= '2022-03-01']    # locate by index
df_hsi.loc[df_hsi.index == '2022-07-04']    # filter a specific date
df_hsi.iloc[0:5]     # integer-location based indexing for selection by position
```

Filtering & Selection of a DataFrame

> REMEMBER
>
> - When selecting subsets of data, square brackets `[]` are used.
> - Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon.
> - Select specific rows and/or columns using `loc` when using the row and column names.
> - Select specific rows and/or columns using `iloc` when using the positions in the table.
> - You can assign new values to a selection based on `loc`/`iloc`.

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`

If you only want to access a scalar value, the fastest way is to use the **.at** and **.iat** methods, which are implemented on all of the data structures.

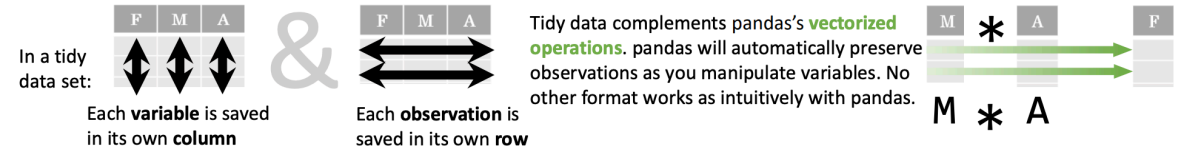
Date	Open	High	Low	Close	Adj Close	Volume
2022-01-03	23510.539063	23605.029297	23193.189453	23274.750000	23274.750000	734331100
2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200
2022-01-05	23323.769531	23323.769531	22851.500000	22907.250000	22907.250000	2768859000
2022-01-06	22843.199219	23082.949219	22709.599609	23072.859375	23072.859375	1765786100
2022-01-07	23318.919922	23497.500000	23162.849609	23493.380859	23493.380859	2602290600

```
In [34]: df_hsi.at[df_hsi.index[4], 'Open']
Out[34]: 23318.919922

In [35]: df_hsi.iat[4, 4]
Out[35]: 23493.380859
```

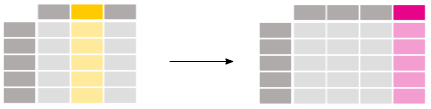
Add New Column

In almost every analysis, we need to create extra columns with calculation, in Pandas the logistic is very simple, tidy and fast. Thanks to the Vectorized data management!

Tidy Data – A foundation for wrangling in pandas



Calculation of the values in new column is element-wise, you don't need to create a loop. Just use Pandas built-in function.



In [34]:
```
df_hsi["Return"] = df_hsi["Adj Close"].pct_change(1)   # percentage change from 1 day before
df_hsi
```

Out[34]:

Date	Open	High	Low	Close	Adj Close	Volume	Return
2022-01-03	23510.539063	23605.029297	23193.189453	23274.750000	23274.750000	734331100	NaN
2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200	0.000648
2022-01-05	23323.769531	23323.769531	22851.500000	22907.250000	22907.250000	2768859000	-0.016427
2022-01-06	22843.199219	23082.949219	22709.599609	23072.859375	23072.859375	1765786100	0.007230
2022-01-07	23318.919922	23497.500000	23162.849609	23493.380859	23493.380859	2602290600	0.018226

In [35]:
```
# add 10,20,50 days moving average
for period in [10,20,50]:
    df_hsi[f"MA{period}"] = df_hsi["Adj Close"].rolling(period).mean()
df_hsi.tail(5)
```

Out[35]:

Date	Open	High	Low	Close	Adj Close	Volume	Return	MA10	MA20	MA50
2022-12-22	19537.449219	19735.000000	19475.679688	19679.220703	19679.220703	1939795100	0.027073	19474.073047	19052.523438	17540.685566
2022-12-23	19382.230469	19686.769531	19380.470703	19593.060547	19593.060547	1363741800	-0.004378	19443.292187	19153.497461	17600.792988
2022-12-28	19787.939453	20099.769531	19787.939453	19898.910156	19898.910156	2823780600	0.015610	19486.820117	19283.545996	17666.513184
2022-12-29	19648.400391	19764.519531	19539.839844	19741.140625	19741.140625	2902362900	-0.007929	19501.314258	19360.369043	17723.044395
2022-12-30	20030.849609	20073.919922	19781.410156	19781.410156	19781.410156	1747706800	0.002040	19512.110351	19419.578027	17788.447012

Drop a Column or a Row

DataFrame.drop(*labels=None*, *, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace =False*, *errors='raise'*)

```
In [38]: # Drop columns
         df_hsi = df_hsi.drop(['MA20'], axis=1)
         df_hsi
```

Out[38]:

Date	Open	High	Low	Close	Adj Close	Volume	Return	MA10	MA50
2022-01-03	23510.539063	23605.029297	23193.189453	23274.750000	23274.750000	734331100	NaN	NaN	NaN
2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200	0.000648	NaN	NaN

```
In [37]: # Drop row
         df_hsi = df_hsi.drop(index=('2022-01-03'))
         df_hsi
```

Out[37]:

Date	Open	High	Low	Close	Adj Close	Volume	Return	MA10	MA50
2022-01-04	23400.619141	23439.300781	23146.890625	23289.839844	23289.839844	1760141200	0.000648	NaN	NaN
2022-01-05	23323.769531	23323.769531	22851.500000	22907.250000	22907.250000	2768859000	-0.016427	NaN	NaN

Missing Data Handling

In a millions-rows database, there will be always some irregular data,

such as Null, NaN, None, Empty or not the type as the defined Series.

According to the need of research, we choose one of the following to handle it.

- DF.isna( ) # indicating if the values are NA. NA values, such as None or numpy.NaN
- DF.dropna( ) # drop row containing NA
- DF.fillna(method="ffill") # fill NA with upper row data
- DF.fillna(0) # fill NA with zero
- DF.fillna(value=some_values) # fill NA with specific value, such as mean or median
- DF['xxx'].interpolate( ) # fill NA with value in between upper and lower

No hard rule to deal with NA, make your reasonable choice.

More observation tools

- DF.nlargest(n, 'column') # Select and order top n entries
- DF.nsmallest(n, 'column') # Select and order top n entries
- DF.sample(n=5) # Randomly select 5 rows
- DF.iloc[[0]] # Select first row on DF
- DF.iloc[[-1]] # Select last row on DF
- DF.loc[(DF.index>??) & (DF.index<??)] # Select with condition

Note: If necessary, store your filtered data in new DF or original DF.

Reshaping DF

Depending the raw database we had, sometimes we need to reshape from column to rows or vice versa.

DF.stack( )

DF.unstack( )

```
In [54]: df_pet = pd.DataFrame([[1, 2], [3, 4]],
                               index=['cat', 'dog'],
                               columns=['weight', 'height'])
         df_pet
```

Out[54]:

	weight	height
cat	1	2
dog	3	4

Stack the columns into single Series

```
In [55]: df_pet = df_pet.stack(level=-1, dropna=True)
         df_pet

Out[55]: cat  weight    1
              height    2
         dog  weight    3
              height    4
         dtype: int64
```

Stack the prescribed level(s) from columns to index

Unstack the rows into columns

```
In [56]: df_pet = df_pet.unstack()
         df_pet
```

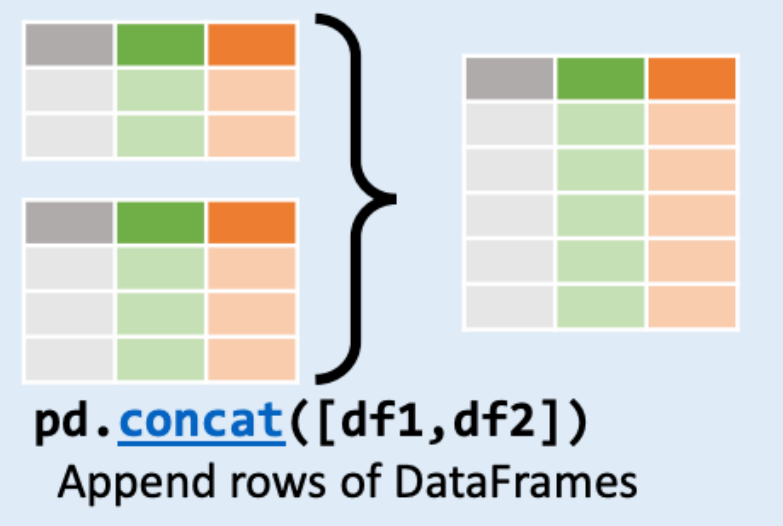
Out[56]:

	weight	height
cat	1	2
dog	3	4

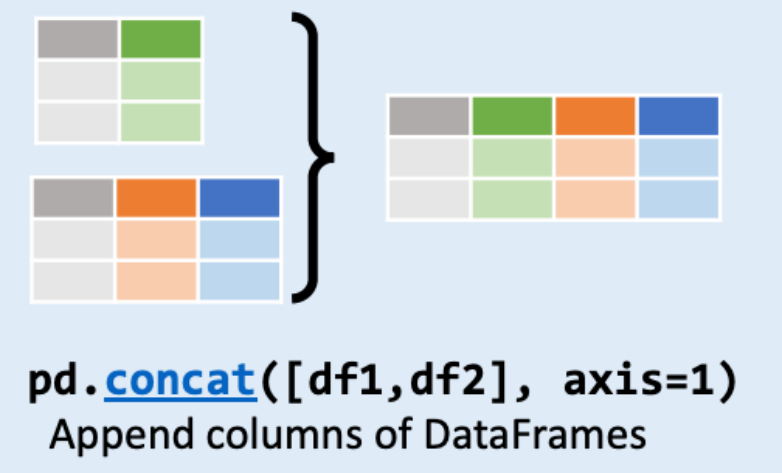
Pivot a level of the (necessarily hierarchical) index labels.

Concatenate DFs

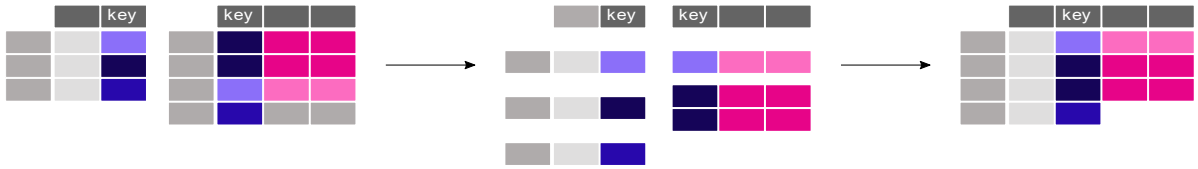
- Append rows



- Append columns or series



Merge DFs with KEY



df_merge = pd.merge(df1, df2, how='left', on='key')

```
In [64]: df1 = pd.DataFrame({'Name': ['Alex', 'Bob', 'Chris'], 'ID': [1, 2, 3]})
         df1
```

Out[64]:

	Name	ID
0	Alex	1
1	Bob	2
2	Chris	3

```
In [65]: df2 = pd.DataFrame({'ID':[1,3],'Score':[65,60]})
         df2
```

Out[65]:

	ID	Score
0	1	65
1	3	60

```
In [66]: df = pd.merge(df1,df2, how='left', on='ID')
         df
```

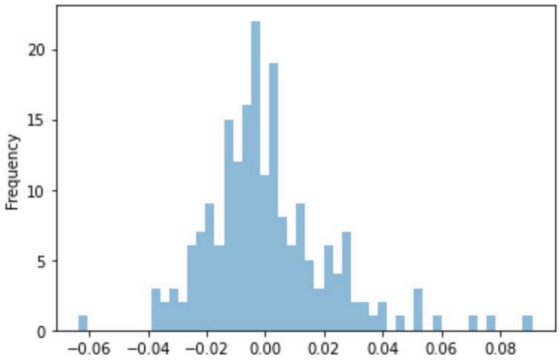
Out[66]:

	Name	ID	Score
0	Alex	1	65.0
1	Bob	2	NaN
2	Chris	3	60.0

Ploting DataFrame – Histogram

Pandas default chart library is Matplotlib. We can plot a quick chart in simple code.

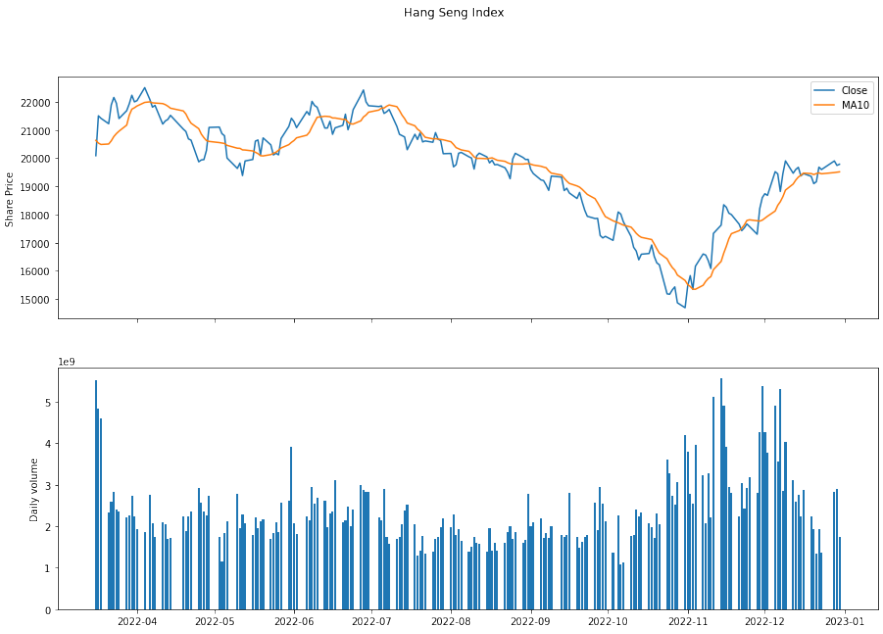
```
# Easy plot with Pandas plot.hist
ax = df_hsi['Return'].plot.hist(bins=50, alpha=0.5)
```



We may plot a better graph with Matplotlib

```
# pip install matplotlib
```

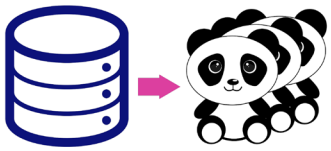
```
import matplotlib.pyplot as plt
fig, (ax1,ax2) = plt.subplots(2,1, sharex=True, figsize=(15, 10))

ax1.plot( df_hsi.index, df_hsi['Close'], df_hsi['MA10'])
ax1.set_ylabel('Share Price')
ax1.legend(['Close', 'MA10'])

ax2.bar( df_hsi.index, df_hsi['Volume'] )
ax2.set_ylabel('Daily volume')

fig.suptitle('Hang Seng Index')
```



Interacting with Database

In real scenario, data are stored in SQL-based relational Databases, such as SQL Server, PostgreSQL, and MySQL. Loading data from SQL based server is straightforward. We use simple local sqlite3 database as example.

Firstly we create a sample table.



```
In [71]: import sqlite3
```

```
In [79]: query = """
         CREATE TABLE sales(
         firstname VARCHAR(20),
         lastname VARCHAR(20),
         age INTEGER,
         sales REAL
         );
         """
```

Connect DB and execute the query

```
In [80]: con = sqlite3.connect('salesdata.sqlite')
         con.execute(query)   # you can create a table once
```

```
Out[80]: <sqlite3.Cursor at 0x11e70df10>
```

```
In [81]: con.commit()
```

```
In [82]: data = [('David', 'Chan', 35, 68000.50),
                 ('Edgar', 'Doe', 45, 79000.60),
                 ('Fion', 'Elle', 30, 88650.30)]
         stmt = "INSERT INTO sales VALUES(?, ?, ?, ?)"
         con.executemany(stmt, data)
```

```
Out[82]: <sqlite3.Cursor at 0x11e70db20>
```

```
In [83]: con.commit()
```

Pull the data and import to Pandas DF

In [84]:
```
cursor = con.execute('select * from sales')
rows = cursor.fetchall()
rows
```

Out[84]:
```
[('David', 'Chan', 35, 68000.5),
 ('Edgar', 'Doe', 45, 79000.6),
 ('Fion', 'Elle', 30, 88650.3)]
```

In [85]:
```
cursor.description
```

Out[85]:
```
(('firstname', None, None, None, None, None, None),
 ('lastname', None, None, None, None, None, None),
 ('age', None, None, None, None, None, None),
 ('sales', None, None, None, None, None, None))
```

In [88]:
```
df_sales = pd.DataFrame(rows,
        columns=[x[0] for x in cursor.description])
df_sales
```

Out[88]:

	firstname	lastname	age	sales
0	David	Chan	35	68000.5
1	Edgar	Doe	45	79000.6
2	Fion	Elle	30	88650.3

Read & Write HDF5

Hierarchical Data Format (**HDF**) is a set of file formats (**HDF4**, **HDF5**) designed to store and organize large amounts of data. The HDF5 format is designed to address some of the limitations of the HDF4 library, and to address current and anticipated requirements of modern systems and applications.

The banking data may be part of these, one file could size over 10GB.

- To write

  df.to_hdf(”data.h5", "df")

- Read from H5

  pd.read_hdf(“data.h5", "df")

DF.groupby

Apart from manipulate data from different DF, it is common to have analysing data in category in the same DF.

GroupBy objects are returned by groupby calls: pandas.DataFrame.groupby(), pandas.Series.groupby()

g = df.groupby(column) a grouped representation of the table

- Can iterate over the groups
- Can aggregate values within each group to get summary stats using agg() function, ['mean', 'max', 'count', 'sum']

Team	Score
A	8.1
A	8.3
A	9.2
B	6.5
B	7.1
B	8.6
B	7.3

max

A	9.2
B	8.6

Assume that we have a DF on the right and want to compare Company A and B's top Mile-Per-Gallon.

	Company	Model	Year	Transmission	EngineSize	MPG
0	A	A1	2019	Manual	1.4	55.4
1	A	A2	2020	Automatic	2.0	67.3
2	A	A3	2021	Automatic	1.4	58.9
3	B	B1	2018	Manual	1.5	52.3
4	B	B2	2019	Automatic	2.0	64.2
5	B	B3	2020	Automatic	1.5	68.9
6	B	B4	2021	Manual	1.5	83.1

```
# groupby columns on Col1 and estimate the
# maximum value of column Col2 for each group
# df.groupby([Col1])[Col2].max()
df.groupby(["Company"])["MPG"].max()
```

```
Company
A    67.3
B    83.1
Name: MPG, dtype: float64
```

df.groupby will NOT re-arrange the original sequence of rows. Whereas Pivoting will.

You can pass the 'mean', 'max', 'count', 'sum', etc to the agg( ) function.
Aggregate using one or more operations over the specified axis.

```
# alternatively, you can pass 'max' to the agg() function
df.groupby(["Company"])["MPG"].agg('max')
```

```
Company
A    67.3
B    83.1
Name: MPG, dtype: float64
```

```
df.groupby(["Company"])["MPG"].agg(['mean','count','std'])
```

Company	mean	count	std
A	60.533333	3	6.115826
B	67.125000	4	12.736921

Add new row of data

Use pd.concat([df, new_df], ignore_index=True) to add new row of data.

Many website use df.append(new_df), but that will be depreciated soon.

```
df = pd.concat([df, pd.DataFrame({
                                    "Company":["C"],
                                    "Model":["C1"],
                                    "Year":[2023],
                                    "Transmission":["Automatic"],
                                    "EngineSize":[1.8] ,
                                    "MPG":[70],
                                    })
                ], ignore_index=True)
```

	Company	Model	Year	Transmission	EngineSize	MPG
0	A	A1	2019	Manual	1.4	55.4
1	A	A2	2020	Automatic	2.0	67.3
2	A	A3	2021	Automatic	1.4	58.9
3	B	B1	2018	Manual	1.5	52.3
4	B	B2	2019	Automatic	2.0	64.2
5	B	B3	2020	Automatic	1.5	68.9
6	B	B4	2021	Manual	1.5	83.1
7	C	C1	2023	Automatic	1.8	70.0

```
df = df.append(pd.DataFrame({
    "Company":["C"],
    "Model":["C1"],
    "Year":[2023],
    "Transmission":["Automatic"],
    "EngineSize":[1.8] ,
    "MPG":[70],
}))
```

```
/var/folders/cq/_ztw83fd0vsg5vkyf8hn0zd40000gn/T/ipykernel_52155/842031746.py:1: FutureWarning: The frame.append meth
od is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df = df.append(pd.DataFrame({
```

Access DF

The fastest way to access a row is by its index. **df.loc[ ]** is convenient; **df.iloc[ ]** is fast.

	loc	iloc
	Label-based (Location)	***Integer position-based (iLocation)***
A Value	A single label `e.g.) df.loc['x1']`	A single integer `e.g.) df.iloc[0]`
A List or Array	A list of labels `e.g.) df.loc[['x1', 'x2']]`	A list of integers `e.g.) df.iloc[[0, 1]]`
A Slice Object	A slice object with labels 'x1' and 'x2' are included `e.g.) df.loc['x1':'x2']`	A slice object with inetgers i is included, j is not included `e.g.) df.iloc[i:j]`

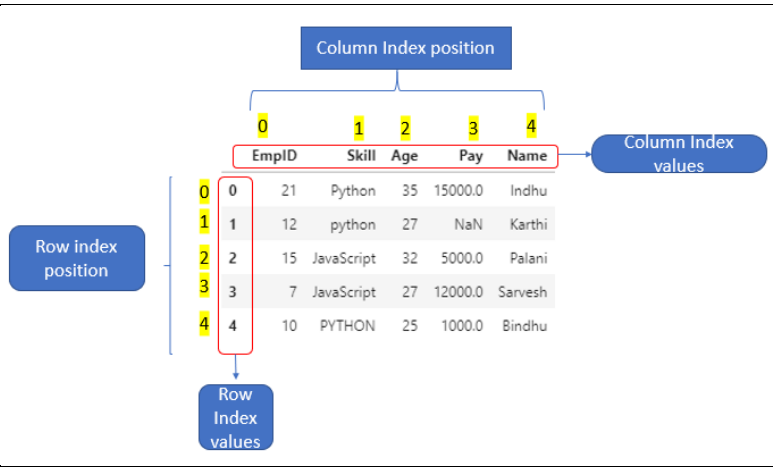
Conditions or Boolean Array	Conditional that returns a boolean Series
	`e.g.) df.loc[df['x1'] > 10]`
	`df.loc[[True, False, False]]`
		A boolean array
		`e.g.) df.iloc[[True, False, False]]`

A Callable Function	A callable function `e.g.) df.loc[lambda df: df['x1'] > 10]`	A callable function `e.g.) df.iloc[lambda x: x.index %2 == 0]`

DataFrame index

In Pandas DF, the index defines the accessing speed of data. Mark the unique and identical column as index or leave it with default increment integer. Typical index can be datetime, or id. There are 4 methods to handle DF's index.

DataFrame.set_index
Set the DataFrame index using existing columns.
DataFrame.reset_index
Opposite of set_index.
DataFrame.reindex
Change to new indices or expand indices.
DataFrame.reindex_like
Change to same indices as other DataFrame.



DF.set_index

	Company	Model	Year	Transmission	EngineSize	MPG
0	A	A1	2019	Manual	1.4	55.4
1	A	A2	2020	Automatic	2.0	67.3
2	A	A3	2021	Automatic	1.4	58.9
3	B	B1	2018	Manual	1.5	52.3
4	B	B2	2019	Automatic	2.0	64.2
5	B	B3	2020	Automatic	1.5	68.9
6	B	B4	2021	Manual	1.5	83.1
7	C	C1	2023	Automatic	1.8	70.0

Set ”Model” as new index

```
1  df.set_index('Model')
```

Model	Company	Year	Transmission	EngineSize	MPG
A1	A	2019	Manual	1.4	55.4
A2	A	2020	Automatic	2.0	67.3
A3	A	2021	Automatic	1.4	58.9
B1	B	2018	Manual	1.5	52.3
B2	B	2019	Automatic	2.0	64.2
B3	B	2020	Automatic	1.5	68.9
B4	B	2021	Manual	1.5	83.1
C1	C	2023	Automatic	1.8	70.0

Df.sort_values – sorting rows order

- Sort by the values along either axis.

```
df.sort_values(by="Year", ascending=True)
```

	Company	Model	Year	Transmission	EngineSize	MPG
3	B	B1	2018	Manual	1.5	52.3
0	A	A1	2019	Manual	1.4	55.4
4	B	B2	2019	Automatic	2.0	64.2
1	A	A2	2020	Automatic	2.0	67.3
5	B	B3	2020	Automatic	1.5	68.9
2	A	A3	2021	Automatic	1.4	58.9
6	B	B4	2021	Manual	1.5	83.1
7	C	C1	2023	Automatic	1.8	70.0

Sorting rows by multiple series

- Note that the integer index was reordered as well. We may reset the index value.

```
df1 = df.sort_values(by=["Company","Year"], ascending=[True,False])
df1
```

	Company	Model	Year	Transmission	EngineSize	MPG
2	A	A3	2021	Automatic	1.4	58.9
1	A	A2	2020	Automatic	2.0	67.3
0	A	A1	2019	Manual	1.4	55.4
6	B	B4	2021	Manual	1.5	83.1
5	B	B3	2020	Automatic	1.5	68.9
4	B	B2	2019	Automatic	2.0	64.2
3	B	B1	2018	Manual	1.5	52.3
7	C	C1	2023	Automatic	1.8	70.0

Alter column/series order

Altering column order is relatively simple.

```
df[["MPG","EngineSize","Transmission","Company","Model","Year"]]
```

	MPG	EngineSize	Transmission	Company	Model	Year
0	55.4	1.4	Manual	A	A1	2019
1	67.3	2.0	Automatic	A	A2	2020
2	58.9	1.4	Automatic	A	A3	2021
3	52.3	1.5	Manual	B	B1	2018
4	64.2	2.0	Automatic	B	B2	2019
5	68.9	1.5	Automatic	B	B3	2020
6	83.1	1.5	Manual	B	B4	2021
7	70.0	1.8	Automatic	C	C1	2023
8	70.0	1.8	Automatic	C	C1	2023

Using .iloc[ ] is the fastest way

```
df.iloc[:,[5,4,3,0,1,2]]
```

	MPG	EngineSize	Transmission	Company	Model	Year
0	55.4	1.4	Manual	A	A1	2019
1	67.3	2.0	Automatic	A	A2	2020
2	58.9	1.4	Automatic	A	A3	2021
3	52.3	1.5	Manual	B	B1	2018
4	64.2	2.0	Automatic	B	B2	2019
5	68.9	1.5	Automatic	B	B3	2020
6	83.1	1.5	Manual	B	B4	2021
7	70.0	1.8	Automatic	C	C1	2023
8	70.0	1.8	Automatic	C	C1	2023

Series.unique

Series.**unique**()
Return unique values of Series object.
ndarray or ExtensionArray.
The unique values returned as a NumPy array.
Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.
Index.unique() serves the same

```
1 df["EngineSize"].unique()
```

```
array([1.4, 2. , 1.5, 1.8])
```

```
1 df["Company"].unique()
```

```
array(['A', 'B', 'C'], dtype=object)
```

```
1 df["Year"].unique()
```

```
array([2019, 2020, 2021, 2018, 2023])
```

Series.nunique

Series.**nunique**(*dropna=True*)
Return number of unique elements in the object.
Excludes NA values by default.
dropna=True means NaN does not count by default.
Index.nunique() serves the same

```
1 df["EngineSize"].nunique()
```

```
4
```

```
1 df["Year"].nunique()
```

```
5
```

```
1 df["Transmission"].nunique()
```

```
2
```

Pd.read_csv and pd.read_json

To read file from json is simple as csv.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write.

API stands for Application Programming Interface. Meaning, the protocol for application communicating each other. Like the data we extract from HKO API, there are many open data available and support json format.

Let's import HK property sales data this time.

Domestic Sales (from 2002)

RATING AND VALUATION DEPARTMENT | Housing | XLS | API Available

LAST UPDATED ON: **2023-03-06**
UPDATE FREQUENCY: **MONTHLY, EXCEPT FOR STOCK, VACANCY AND TAKE-UP TO BE UPDATED ANNUALLY**

Domestic Sales (from 2002)

- First find the json link from data.gov.hk, then pd.read_json

```
link = "https://api.data.gov.hk/v2/filter?q=%7B%
data = pd.read_json(link)
```

- Change the column name with df.rename(columns={old:new})
- Year and Month column shall be combined for identical index.

```
# change the column name
data.rename(columns = {'年':'Year','月':'Month','數目 No.':'Case',
    '總值  (百萬元) Consideration   ($ million)':'Amount_mil'}, inplace = True)
```

```
data
```

	Year	Month	Case	Amount_mil
0	2002	5	7325	16940
1	2002	6	7195	16129
2	2002	7	4961	10272
3	2002	8	4881	9867
4	2002	9	6278	12113
...	...	...	...	...
193	2018	6	6713	68023
194	2018	7	6091	65237
195	2018	8	4822	46765
196	2018	9	3500	37083
197	2018	10	4243	38571

198 rows × 4 columns

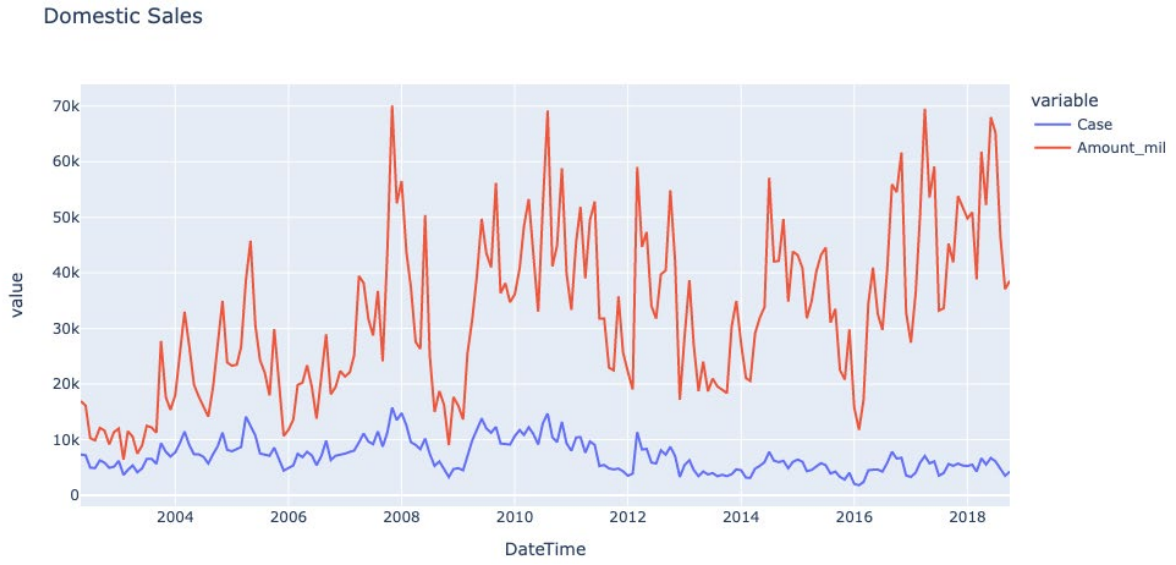
```python
# join Year and Month as new column
data["DateTime"] = data["Year"].astype(str) + "-" + data["Month"].astype(str)
# drop Year and Month
data = data.drop(['Year','Month'], axis=1)
# set DateTime as index
data = data.set_index('DateTime')
data
```

DateTime	Case	Amount_mil
2002-5	7325	16940
2002-6	7195	16129
2002-7	4961	10272
2002-8	4881	9867
2002-9	6278	12113

- Join Year and Month column as DateTime
- Drop Year and Month column
- Set index as DateTime

Let plot a quick chart with Plotly. We shall discuss Plotly usage in detail soon.

```python
import plotly.express as px
fig = px.line(data, x=data.index, y=["Case","Amount_mil"], title='Domestic Sales')
fig.show()
```



Df.corr

df.corr(), series.corr()

Compute pairwise correlation of columns, excluding NA/null values.

```
1 data.corr()
```

	Case	Amount_mil
Case	1.000000	0.484761
Amount_mil	0.484761	1.000000

```
1 data["Case"].corr(data["Amount_mil"])
```

```
0.48476064701593535
```

Note: The coefficient always has a value between −1 and 1 .

- -1 means perfect negative linear correlation.
- +1 means perfect positive linear correlation.
- 0 means no linear dependency between variables.

df.T

DateTime	Case	Amount_mil
2002-5	7325	16940
2002-6	7195	16129
2002-7	4961	10272
2002-8	4881	9867
2002-9	6278	12113
...	...	...
2018-6	6713	68023
2018-7	6091	65237
2018-8	4822	46765
2018-9	3500	37083
2018-10	4243	38571

198 rows × 2 columns

df.T or df.transpose
The transpose of the DataFrame.

```
1 data.T
```

DateTime	2002-5	2002-6	2002-7	2002-8	2002-9	2002-10	2002-11	2002-12	2003-1	2003-2	...
Case	7325	7195	4961	4881	6278	5863	4941	5129	6187	3649	...
Amount_mil	16940	16129	10272	9867	12113	11658	9122	11352	11997	6405	...

2 rows × 198 columns

Df.to_numpy

df.to_numpy, series.to_numpy

Convert the DataFrame to a NumPy array. By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame.

```
1 data["Case"].to_numpy()
```

```
array([ 7325,  7195,  4961,  4881,  6278,  5863,  4941,  5129,  6187,
        3649,  4550,  5373,  4130,  4833,  6525,  6559,  5632,  9360,
        7811,  6967,  7726,  9449, 11449,  8994,  7380,  7362,  6911,
        5716,  7385,  8811, 11281,  8166,  7909,  8260,  8673, 14124,
       12463, 10750,  7497,  7298,  7100,  8554,  6308,  4426,  4899,
        5369,  7456,  6849,  7812,  7150,  5398,  7032,  9811,  6335,
```

```
1 data.to_numpy()
```

```
array([[ 7325, 16940],
       [ 7195, 16129],
       [ 4961, 10272],
       [ 4881,  9867],
       [ 6278, 12113],
       [ 5863, 11658],
       [ 4941,  9122],
       [ 5129, 11352],
       [ 6187, 11997],
       [ 3649,  6405],
       [ 4550, 11520],
```

Time Series

Time Series is a series of values of a quantity obtained at successive times, often with equal intervals between them. In mathematics, a **time series** is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.

The domestic sales property data over months, stocks data, weather data over years, real-time traffic flow data, are the good examples.

To handle time series analysis, the first point to check is the index. It must be set to datetime index.

We take US Treasury data in csv format from yahoo finance as example.
Df.index shows the index is ranged integer.


```
1  tnx = pd.read_csv("^TNX.csv")
2  tnx.tail(5)
```

	Date	Open	High	Low	Close	Adj Close	Volume
7057	2022-12-26	NaN	NaN	NaN	NaN	NaN	NaN
7058	2022-12-27	3.787	3.862	3.787	3.860	3.860	0.0
7059	2022-12-28	3.818	3.890	3.815	3.887	3.887	0.0
7060	2022-12-29	3.868	3.886	3.818	3.835	3.835	0.0
7061	2022-12-30	3.869	3.905	3.831	3.879	3.879	0.0

```
1  tnx.index
```

```
RangeIndex(start=0, stop=7062, step=1)
```

df.info() shows all series dtypes and information

```
1  tnx.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7062 entries, 0 to 7061
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       7062 non-null   object
 1   Open       5780 non-null   float64
 2   High       5780 non-null   float64
 3   Low        5780 non-null   float64
 4   Close      5780 non-null   float64
 5   Adj Close  5780 non-null   float64
 6   Volume     5780 non-null   float64
dtypes: float64(6), object(1)
memory usage: 386.3+ KB
```

pd.to_datetime

Convert series to datetime. This function converts a scalar, array-like, Series or DataFrame/dict-like to a pandas datetime object.

```
tnx['Date'] = pd.to_datetime(tnx['Date'],format="%Y-%m-%d")
tnx.Date
```

```
0       2000-01-03
1       2000-01-04
2       2000-01-05
3       2000-01-06
4       2000-01-07
           ...
7057    2022-12-26
7058    2022-12-27
7059    2022-12-28
7060    2022-12-29
7061    2022-12-30
Name: Date, Length: 7062, dtype: datetime64[ns]
```

Setting the datetime index

```
tnx = tnx.set_index('Date')
tnx.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7062 entries, 2000-01-03 to 2022-12-30
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Open       5780 non-null   float64
 1   High       5780 non-null   float64
 2   Low        5780 non-null   float64
 3   Close      5780 non-null   float64
 4   Adj Close  5780 non-null   float64
 5   Volume     5780 non-null   float64
dtypes: float64(6)
memory usage: 386.2 KB
```

```
tnx.tail(3)
```

Date	Open	High	Low	Close	Adj Close	Volume
2022-12-28	3.818	3.890	3.815	3.887	3.887	0.0
2022-12-29	3.868	3.886	3.818	3.835	3.835	0.0
2022-12-30	3.869	3.905	3.831	3.879	3.879	0.0

Grouping columns for datetime

In some of the dataset, the year, month, day columns are separate in different columns. You can group them with

pd.to_datetime(df_dt[['year','month','day']])

```
df_dt = pd.DataFrame({'year': [2023, 2023],
                      'month': [1, 1],
                      'day': [1, 2],
                      'sales': [3750, 3900]})
df_dt
```

	year	month	day	sales
0	2023	1	1	3750
1	2023	1	2	3900

```
df_dt['date'] = pd.to_datetime(df_dt[['year','month','day']])
df_dt = df_dt.drop(['year','month','day'], axis=1)
df_dt = df_dt.set_index('date')
df_dt
```

	sales
date	
2023-01-01	3750
2023-01-02	3900

Unix epoch time

Unix is enterprise server OS, the time format is a series of number. It can convert to Pandas datetime as well.

```
pd.to_datetime(1681038343, unit='s')
```

```
Timestamp('2023-04-09 11:05:43')
```

```
pd.to_datetime(1681038343433502912, unit='ns')
```

```
Timestamp('2023-04-09 11:05:43.433502912')
```

Unix Epoch Clock

Unix time across midnight into 17 September 2004 (no leap second)

TAI (17 September 2004)	UTC (16 to 17 September 2004)	Unix time
2004-09-17T00:00:30.75	2004-09-16T23:59:58.75	1 095 379 198.75
2004-09-17T00:00:31.00	2004-09-16T23:59:59.00	1 095 379 199.00
2004-09-17T00:00:31.25	2004-09-16T23:59:59.25	1 095 379 199.25

Categorical Data Type

Pandas supports categorical data type, similar to R language's factor.

Categorical are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (categories; levels in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

```
1 animal = pd.Series(["Bird","Cat", "Dog","Elephant"], dtype="category")
```

```
1 animal.info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 4 entries, 0 to 3
Series name: None
Non-Null Count  Dtype
--------------  -----
4 non-null      category
dtypes: category(1)
memory usage: 336.0 bytes
```

Sometimes we need to group scalar data in categorical data.

For example, we have a score sheet and need to rank them into several classes.

```
1 import numpy as np
2 df_exam = pd.DataFrame({'Score': np.random.randint(1, 101, size=40)})
```

```
1 df_exam.head(6)
```

	Score
0	6
1	8
2	41
3	60
4	93
5	35

np.where

Use np.where to iterate conditional statement. Nested np.where can apply.

```
df_exam['Result'] = pd.Series(np.where(df_exam['Score']>70, 'Distiction',
                              np.where(df_exam['Score']>50, 'Pass', 'Fail')
                    )).astype('category')
```

```
df_exam.sample(5)
```

	Score	Result
31	51	Pass
24	50	Fail
6	18	Fail
36	84	Distiction
39	59	Pass

Handling category data is much faster than string variable, since Pandas treat it as single item.

```
df_exam.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 40 entries, 0 to 39
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype   
---  ------  --------------  -----   
 0   Score   40 non-null     int64   
 1   Result  40 non-null     category
dtypes: category(1), int64(1)
memory usage: 620.0 bytes
```

Pandas rolling window

Pandas **dataframe.rolling()** function provides the feature of rolling window calculations. The concept of rolling window calculation is most primarily used in signal processing and time-series data.

The syntax:

DataFrame.**rolling**(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*, *step=None*, *method='single'*)
Prepare time series data of Heng Sang Index via yfinance.

From 2020-01-01 to 2023-01-01

```
1 import numpy as np
2 import pandas as pd
3 import yfinance as yf
```

```
1 hsi = yf.download("^HSI", start="2020-01-01", end="2023-01-01")
2 hsi.head(3)
```

```
[**********100%**********]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-02	28249.369141	28543.519531	28245.970703	28543.519531	28543.519531	1262732800
2020-01-03	28828.359375	28883.300781	28428.169922	28451.500000	28451.500000	1797904800
2020-01-06	28326.500000	28367.869141	28054.289062	28226.189453	28226.189453	1793426600

Moving Average (MA)- rolling mean

Prepare new column 'MA10' which is the moving average 10 days mean of 'Adj Close'

```
1 hsi['MA10'] = hsi['Adj Close'].rolling(10).mean()
2 hsi.tail(5)
```

Date	Open	High	Low	Close	Adj Close	Volume	MA30	MA10
2022-12-22	19537.449219	19735.000000	19475.679688	19679.220703	19679.220703	1939795100	19474.073047	19474.073047
2022-12-23	19382.230469	19686.769531	19380.470703	19593.060547	19593.060547	1363741800	19443.292187	19443.292187
2022-12-28	19787.939453	20099.769531	19787.939453	19898.910156	19898.910156	2823780600	19486.820117	19486.820117
2022-12-29	19648.400391	19764.519531	19539.839844	19741.140625	19741.140625	2902362900	19501.314258	19501.314258
2022-12-30	20030.849609	20073.919922	19781.410156	19781.410156	19781.410156	1747706800	19512.110352	19512.110352

Exponentially Weighted Moving Average (EMA)

Prepare 10 days EMA data of 'Adj Close' in new column 'EMA10'

Syntax:

Series.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True, ignore_na=False, axis=0, times=None, method='single')

decaying adjustment factor in beginning periods

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + \ldots + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + \ldots + (1-\alpha)^t}$$

```
hsi['EMA10'] = hsi['Adj Close'].ewm(span=10).mean()
hsi.tail(5)
```

Date	Open	High	Low	Close	Adj Close	Volume	MA30	MA10	EMA10
2022-12-22	19537.449219	19735.000000	19475.679688	19679.220703	19679.220703	1939795100	19474.073047	19474.073047	19329.083741
2022-12-23	19382.230469	19686.769531	19380.470703	19593.060547	19593.060547	1363741800	19443.292187	19443.292187	19377.079524
2022-12-28	19787.939453	20099.769531	19787.939453	19898.910156	19898.910156	2823780600	19486.820117	19486.820117	19471.957821
2022-12-29	19648.400391	19764.519531	19539.839844	19741.140625	19741.140625	2902362900	19501.314258	19501.314258	19520.900149
2022-12-30	20030.849609	20073.919922	19781.410156	19781.410156	19781.410156	1747706800	19512.110352	19512.110352	19568.265605

Getting specific period MA/EMA

Use slicing series and get the last row of data

```
hsi['Adj Close'][-10:].rolling(10).mean()[-1]
```

```
19512.1103515625
```

```
hsi['Adj Close'][-10:].ewm(span=10).mean()[-1]
```

```
19618.33963667069
```

Rolling Standard Deviation (STD)

Measuring volatility and risk with Standard Deviation. Let use 250 days STD

```
hsi['250Std'] = hsi['Adj Close'].rolling(250).std()
hsi.tail(5)
```

Date	Adj Close	MA30	MA10	EMA10	Kurt50	Skew50	250Max	250Min	250Std
2022-12-22	19679.220703	19474.073047	19474.073047	19329.083741	-1.232598	-0.145976	24965.550781	14687.019531	2365.320875
2022-12-23	19593.060547	19443.292187	19443.292187	19377.079524	-1.244126	-0.208008	24965.550781	14687.019531	2360.480438
2022-12-28	19898.910156	19486.820117	19486.820117	19471.957821	-1.239929	-0.258532	24965.550781	14687.019531	2354.616331
2022-12-29	19741.140625	19501.314258	19501.314258	19520.900149	-1.247615	-0.310313	24965.550781	14687.019531	2348.397469
2022-12-30	19781.410156	19512.110352	19512.110352	19568.265605	-1.217817	-0.375461	24965.550781	14687.019531	2341.891306

Autocorrelation of lag 1 and lag 2 period

Autocorrelation means the correlation of a series and its previous n term. Usually use lag 1 or lag 2.

Find to HSI autocorrelation using lag 1 and 2 day respectively.

```
df_all['HSI'].autocorr(lag=1)
```

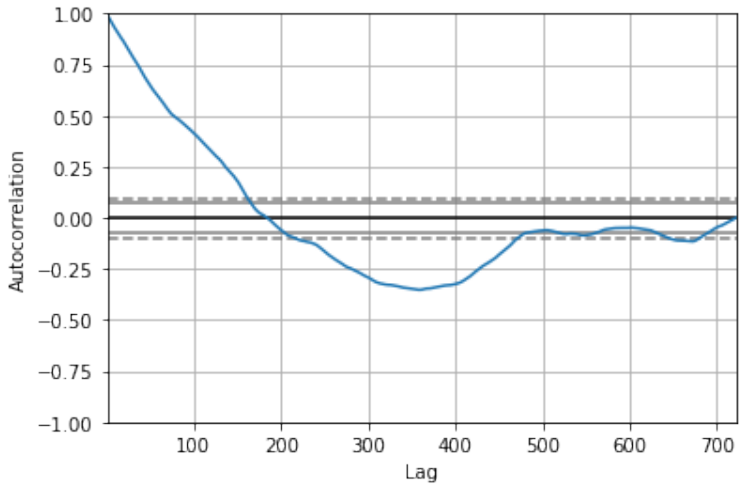
```
0.9941400668946364
```

```
df_all['HSI'].autocorr(lag=2)
```

```
0.9885254205676482
```

```
pd.plotting.autocorrelation_plot(df_all['HSI'])
```

```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



For machine learning method, analyst might use AR, ARMA and ARIMA

Series.where() and Series.mask()

```
p = pd.Series([1,2,3,4,5])
p
```

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

```
# Replace values where the condition is False
p.where(p<=3, 'where p<=3 is false')
```

```
0                        1
1                        2
2                        3
3    where p<=3 is false
4    where p<=3 is false
dtype: object
```

```
# Replace values where the condition is True
p.mask(p<=3, 'where p<=3 is true')
```

```
0    where p<=3 is true
1    where p<=3 is true
2    where p<=3 is true
3                     4
4                     5
dtype: object
```

Chapter Summary

We had learnt all the basic Pandas frame work. It's time to research some real data and project.

The more you tried, the more you would success. Don't afraid of the bugs and mistakes, senior programmer also experienced them daily.

Since Pandas, NumPy, SciPy and Python are evolving rapidly, some online information may not be update. The official document is always trustworthy.

References

Official Website:

- https://pandas.pydata.org/docs/

Reference textbook:

- Python for Data Analysis, 2nd edition, Wes McKinney, O'Reilly
- Python Data Science Handbook, Jake VanderPlas, O'Reilly
- Community Backup: https://pandas.pydata.org/docs/getting_started/tutorials.html