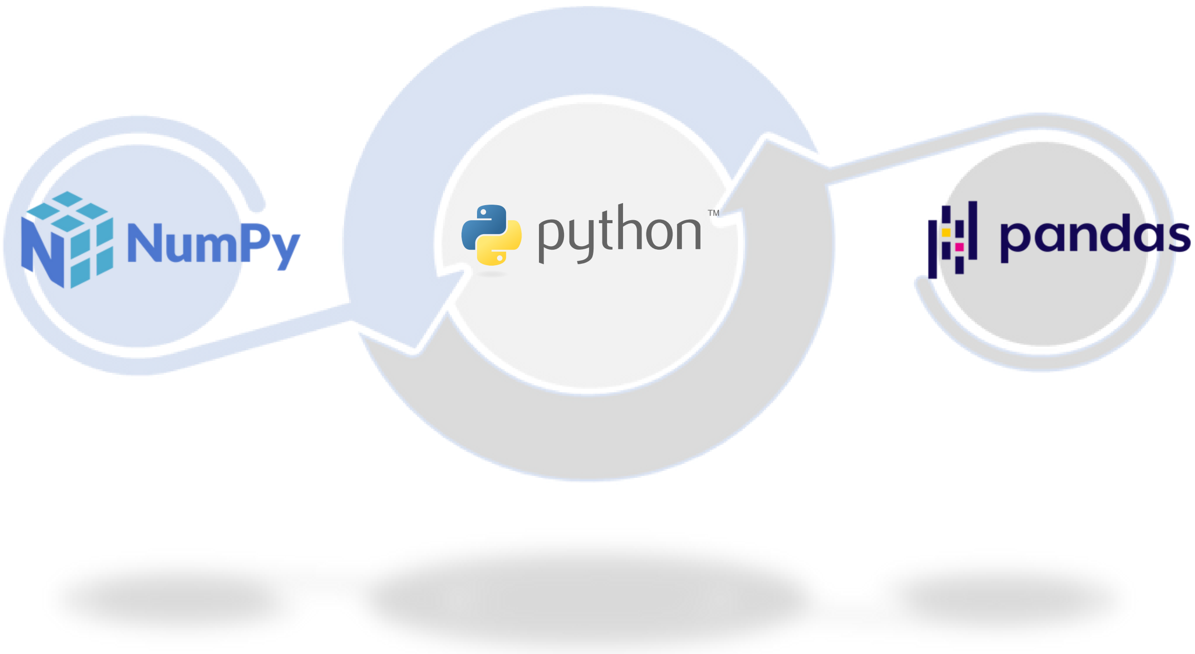
NumPy and Pandas Fundamentals:

A Step-by-Step Guide for Data Analysis

Wei-Hsin Hsu, 2023/12/9

Linkedin

Github



In this document, we will cover fundamental techniques of NumPy and Pandas, two open-source Python libraries that are very helpful for data manipulation, data analysis, and multi-dimensional data processing. They provide convenient and practical data structures and functions, and are widely used in the fields of data analysis.

Once you master the techniques mentioned in this document, you will have the ability to perform basic data processing, analysis, and visualization, taking the first step towards becoming a data analyst. :)

Table of Contents

NumPy

NumPy (short for Numerical Python) is a powerful numerical computing library in Python. It offers many mathematical functions and operations, particularly tailored for multi-dimensional arrays. It provides scientists, engineers, and data scientists with rich tools, making calculations and data analysis in Python more easily and efficient.

We load the NumPy library and commonly use 'np' as an alias.

In [1]:
```
import numpy as np
```

1. NumPy Array

Creating a NumPy Array

Elements of a NumPy array can be assigned using a list or tuple.

In [2]:
```

A= np.array([1, 2, 3])
B= np.array((4, 5, 6))
print(A)
print(B)
```

```
[1 2 3]
[4 5 6]
```

Datatype of the NumPy Array

In [3]:
```
type(A)
```

Out[3]:
```
numpy.ndarray
```

Basic Operations

In [4]:
```
print(A**2)
```

```
[1 4 9]
```

In [5]:
```
print(A+B)
print(A-B)
print(A*B)
print(A/B)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
```

2-dimensional NumPy Array

In [6]:
```
C= np.array([[1, 2, 3], [4, 5, 6]])
D= np.array([[7, 8, 9], [10, 11, 12]])
print(C)
print(D)
```

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

For NumPy arrays with more dimensions, you just need to add data of the same format accordingly.

In [7]:
```
print(C+D)
# print(C-D)
# print(C*D)
# print(C/D)
```

```
[[ 8 10 12]
 [14 16 18]]
```

Shape, Size, and Dimension

In [8]:
```
A= np.array([[1, 2, 3, 4, 5],[6, 7, 8, 9, 10]])
B= np.array([3, 2, 1])

#Size
print(A.shape)
print(B.shape)
```

```
(2, 5)
(3,)
```

In [9]:
```
#Values count
print(A.size)
print(B.size)
```

```
10
3
```

In [10]:
```
#Dimension
print(A.ndim)
print(B.ndim)
```

```
2
1
```

Transpose

Transposing a 2-D array swaps its rows and columns.

In [11]:
```
A= np.array([[1,2,3],[4,5,6]])
A.transpose() # This is equivalent to A.T
```

Out[11]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

ravel() can convert multi-dimensional array into 1D array.

In [12]:
```
A.ravel()
```

Out[12]:
```
array([1, 2, 3, 4, 5, 6])
```

Creating Array with 0 and 1

np.zeros() creates an array of specified size with 0, while np.ones() creates an array of specified size with 1.

In [13]:
```
print(np.zeros(10))
print(np.ones(10))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [14]:
```
np.zeros((2,4))
```

Out[14]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [15]:
```
np.ones((3,2,1))
```

Out[15]:
```
array([[[1.],
        [1.]],

       [[1.],
        [1.]],

       [[1.],
        [1.]]])
```

2. Sequence

Arithmetic Sequence with Specified Intervals

In [16]:
```
np.arange(0,100,25)
```

Out[16]:
```
array([ 0, 25, 50, 75])
```

Arithmetic Sequence with a Specified Number of Elements.

Noted that argument 'endpoint = True' meaning includes stop, and vice versa.(Default is True)

In [17]:
```
np.linspace(0,100,26, endpoint= True)
```

Out[17]:
```
array([  0.,   4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.,  40.,
        44.,  48.,  52.,  56.,  60.,  64.,  68.,  72.,  76.,  80.,  84.,
        88.,  92.,  96., 100.])
```

3. Operation Functions

Statistical Operation

In [18]:
```
A = np.array([1, 3, 5, 7, 9])
print(np.mean(A))
print(np.median(A))
print(np.std(A))
print(np.var(A))
```

```
5.0
5.0
2.8284271247461903
8.0
```

Mathematical Operation

In [19]:
```
num= 100
print(np.sqrt(num)) #Square root
print(np.exp(num)) #Exponential
print(np.log(num)) #Logarithm with the base e
print(np.log10(num)) #Logarithm with the base 10
#print(np.sin(num))
#print(np.cos(num))
#......
```

```
10.0
2.6881171418161356e+43
4.605170185988092
2.0
```

Calculating Inner product (Dot).

In [20]:
```
A= np.array([1, 2, 3])
B= np.array([4, 5, 6])
C= np.array([1, 4, 7])
print(np.dot(A, B))
print(np.dot(A, C))
```

```
32
30
```

4. Creating Array with Random Values

random.randn() is used to randomly generate data based on the Standard Normal Distribution.

In [21]:
```
np.random.randn(5)
```

Out[21]:
```
array([ 1.6485286 , -1.46597283, -0.37945557, -0.26006813, -0.10395762])
```

In [22]:
```
np.random.randn(2, 3)
```

Out[22]:
```
array([[-0.03435319,  0.71343253, -0.71856028],
       [-0.14204424, -0.2228869 , -3.11251075]])
```

numpy.random.randint(low=x, high=y, size=z) generates z integers from x to y, including x but excluding y.

In [23]:
```
np.random.randint(low= 1, high= 10, size=10)
```

Out[23]:
```
array([7, 7, 1, 8, 5, 6, 2, 7, 5, 5])
```

5. Selecting Data in NumPy Array

In [24]:
```
A= np.array([1, 2, 3, 4, 5, 6])
B= np.array((7, 8, 9, 10, 11, 12))
A[0]
```

Out[24]:
```
1
```

In [25]:
```
A[2:4]
```

Out[25]:
```
array([3, 4])
```

Changing Values

In [26]:
```
A[2]= 11
B[0]= 3
print(A)
print(B)
```

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

Filtering and more Operation

In [27]:
```
C= np.array([[1, 2, 3], [4, 5, 6]])
D= np.array([[7, 8, 9], [10, 11, 12]])

print(C[0,1])
print(D[0,0:2])
```

```
2
[7 8]
```

In [28]:
```
print(C>2)
print(C[C>2])
```

```
[[False False  True]
 [ True  True  True]]
[3 4 5 6]
```

In [29]:
```
print(C.min())
print(np.max(C, axis=1)) # This is equilaveltn to C.max(axis=1)
#np.sum()
#np.product()
#......
```

```
1
[3 6]
```

In [30]:
```
print(C.sum())
print(C.sum(axis=0))
print(C.mean(axis=1))
```

```
21
[5 7 9]
[2. 5.]
```

6. Reshaping NumPy Arrays

In [31]:
```
#Reshape
A= np.array([1,2,3,4,5,6,7,8])
A.reshape(2,4)
```

Out[31]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

7. Concatenate NumPy Arrays

In [32]:
```
A= np.array([1, 2, 3, 4, 5, 6, 7, 8])
B= np.array([7, 6, 5, 4, 3, 2, 1, 0])
print(np.concatenate((A,B)))
```

```
[1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 0]
```

8. Changing Datatype

In [33]:
```
A= np.array([1, 2, 3])
print(A.dtype)
B= np.array([[1.5, 2.5, 3.5],[4, 5, 6]])
print(B.dtype)
```

```
int32
float64
```

In [34]:
```
C= A.astype(np.float64)
print(C.dtype)
D= B.astype(np.int64)
print(D)
print(D.dtype)
```

```
float64
[[1 2 3]
 [4 5 6]]
int64
```

9. Sorting Data

In [35]:
```
A= np.array([3, 2, 5, 4, 1])
np.sort(A)
```

Out[35]:
```
array([1, 2, 3, 4, 5])
```

In [36]:
```
A= np.array([[1,3,2],[4,6,5]])
print(np.sort(A))
print(np.sort(A, axis= 0))
```

```
[[1 2 3]
 [4 5 6]]
[[1 3 2]
 [4 6 5]]
```

10. Splitting Data

In [37]:
```
A= np.arange(16).reshape(4,4)
A
```

Out[37]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Horizontal Splitting

In [38]:
```
np.hsplit(A,2) #Splitting to 2 equal parts
```

Out[38]:
```
[array([[ 0,  1],
        [ 4,  5],
        [ 8,  9],
        [12, 13]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11],
        [14, 15]])]
```

In [39]:
```
np.hsplit(A,4) #Splitting to 4 equal parts
```

Out[39]:
```
[array([[ 0],
        [ 4],
        [ 8],
        [12]]),
 array([[ 1],
        [ 5],
        [ 9],
        [13]]),
 array([[ 2],
        [ 6],
        [10],
        [14]]),
 array([[ 3],
        [ 7],
        [11],
        [15]])]
```

Vertial Splitting

In [40]:
```
np.vsplit(A,4) #Splitting to 4 equal parts
```

Out[40]:
```
[array([[0, 1, 2, 3]]),
 array([[4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11]]),
 array([[12, 13, 14, 15]])]
```

Pandas

Pandas (derived from "Panel Data") is a powerful library in Python that provides high-performance, user-friendly data structures and functions for data analysis. Pandas serves as an ideal tool for handling structured data, making data cleaning, processing, and analysis more accessible in Python.

We load the Pandas library and commonly use 'pd' as an alias.

In [41]:
```
import pandas as pd
```

1. Series

A Series is a one-dimensional array of data, much like a column in Excel, and each data has an index.

Creating a Series

In [42]:
```
data_series= pd.Series([1, 2, 3, 4, 5])
data_series
```

Out[42]:
```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Datatype of the Series

In [43]:
```
type(data_series)
```

Out[43]:
```
pandas.core.series.Series
```

Index

In [44]:
```
data_series.index
```

Out[44]:
```
RangeIndex(start=0, stop=5, step=1)
```

We did not assign an index for the series at first, so the index defaults to numbers starting from 0.

Values

In [45]:
```
data_series.values
```

Out[45]:
```
array([1, 2, 3, 4, 5], dtype=int64)
```

Selecting Data in the Series

Let's use stock price data for 3 companies at a certain point of time as an example.

In [46]:
```
st= {"AAPL":194.27, "META": 326.59, "NVDA":465.96}
st_series= pd.Series(st)
st_series
```

Out[46]:
```
AAPL    194.27
META    326.59
NVDA    465.96
dtype: float64
```

Select the data in the 2nd place, noticed that Python uses zero-based indexing.

In [47]:
```
st_series[1]
```

Out[47]:
```
326.59
```

Select multiple data.

In [48]:
```
st_series[[0,2]]
```

Out[48]:
```
AAPL    194.27
NVDA    465.96
dtype: float64
```

Use the Index to select data.

In [49]:
```
st_series["AAPL"]
```

Out[49]:
```
194.27
```

2. Dataframe

A DataFrame is a two-dimensional data, it has rows and columns in a table format and also has an index for each observation.

Creating a DataFrame

Let's use the daily stock price data for the same companies over a period of 10 days.
※Noted that this data is fictional, and in practice, stock price data may contain missing values because the stock market is closed during holidays.

In [50]:
```
st_m = {"Date":["2023-12-08", "2023-12-09", "2023-12-10", "2023-12-11", "202
    "AAPL": [194.27, 198.32, 199.56, 201.19, 200.35, 201.88, 202.52, 201.47,
    "META": [326.59, 330.12, 327.31, 339.51, 336.87, 338.12, 340.53, 339.12,
    "NVDA": [465.97, 467.13, 471.52, 466.31, 465.19, 466.12, 470.67, 468.53,
}
df = pd.DataFrame(st_m)
df
```

Out[50]:

	Date	AAPL	META	NVDA
0	2023-12-08	194.27	326.59	465.97
1	2023-12-09	198.32	330.12	467.13
2	2023-12-10	199.56	327.31	471.52
3	2023-12-11	201.19	339.51	466.31
4	2023-12-12	200.35	336.87	465.19
5	2023-12-13	201.88	338.12	466.12
6	2023-12-14	202.52	340.53	470.67
7	2023-12-15	201.47	339.12	468.53
8	2023-12-16	203.51	342.90	469.09
9	2023-12-17	204.76	345.19	467.13

Datatype of the DataFrame

In [51]:
```
type(df)
```

Out[51]:
```
pandas.core.frame.DataFrame
```

Other Method of Importing Data

If the data is stored in another format, it can be loaded using pd.read_filetype("file"), such as Excel, CSV, etc.

In [52]:
```
# df= pd.read_excel("file")
# df= pd.read_csv("file")
# ......
```

Saving the Data as Other Format

Once you complete programming, your can save the file as excel, csv, etc. by using these code.

In [53]:
```
# df.to_excel("file")
# df.to_csv("file")
# ......
```

Selecting Data in the DataFrame

In [54]:
```
df.loc[1]
```

Out[54]:
```
Date    2023-12-09
AAPL        198.32
META        330.12
NVDA        467.13
Name: 1, dtype: object
```

In [55]:
```
df.iloc[1]
```

Out[55]:
```
Date    2023-12-09
AAPL        198.32
META        330.12
NVDA        467.13
Name: 1, dtype: object
```

It might appear that loc and iloc are the same, but they are not.
loc is used to retrieve values based on labels, while iloc is used to retrieve values based on integer positions of columns.

Dropping a Column

In [56]:
```
df= pd.DataFrame(st_m, index= st_m["Date"] ) #Set index = Date
df.drop(["Date"],axis=1,  inplace= True) #Because We've set index to Date, s
df
```

Out[56]:

	AAPL	META	NVDA
2023-12-08	194.27	326.59	465.97
2023-12-09	198.32	330.12	467.13
2023-12-10	199.56	327.31	471.52
2023-12-11	201.19	339.51	466.31
2023-12-12	200.35	336.87	465.19
2023-12-13	201.88	338.12	466.12
2023-12-14	202.52	340.53	470.67
2023-12-15	201.47	339.12	468.53
2023-12-16	203.51	342.90	469.09
2023-12-17	204.76	345.19	467.13

Noticed that the 'inplace' argument above.
When working with data, if you want to make modifications to the original object, you need to set 'inplace =True'. Otherwise, the changes will not be applied to the original object.

You can see that loc can be used to select data based on index labels.

In [57]:
```
df.loc["2023-12-11"]
```

Out[57]:
```
AAPL    201.19
META    339.51
NVDA    466.31
Name: 2023-12-11, dtype: float64
```

Selecting a Column

In [58]:
```
df["AAPL"]
```

Out[58]:
```
2023-12-08    194.27
2023-12-09    198.32
2023-12-10    199.56
2023-12-11    201.19
2023-12-12    200.35
2023-12-13    201.88
2023-12-14    202.52
2023-12-15    201.47
2023-12-16    203.51
2023-12-17    204.76
Name: AAPL, dtype: float64
```

Mathematical Operations on an Column

In [59]:
```
df["AAPL"].mean()
#df["AAPL"].sum()
#df["AAPL"].max()
#......
```

Out[59]:
```
200.78300000000002
```

Check the Data Distribution

In [60]:
```
df["AAPL"].value_counts()
```

Out[60]:
```
194.27    1
198.32    1
199.56    1
201.19    1
200.35    1
201.88    1
202.52    1
201.47    1
203.51    1
204.76    1
Name: AAPL, dtype: int64
```

Filtering Data

In [61]:
```
df["AAPL"]> 200
```

Out[61]:
```
2023-12-08    False
2023-12-09    False
2023-12-10    False
2023-12-11     True
2023-12-12     True
2023-12-13     True
2023-12-14     True
2023-12-15     True
2023-12-16     True
2023-12-17     True
Name: AAPL, dtype: bool
```

Filtering Data with multiple conditions.

In [62]:
```
df[(df["AAPL"] > 200) & (df["NVDA"] > 470)]
```

Out[62]:

	AAPL	META	NVDA
2023-12-14	202.52	340.53	470.67

Sorting Data

In [63]:
```
df.sort_values(by ="AAPL")
```

Out[63]:

	AAPL	META	NVDA
2023-12-08	194.27	326.59	465.97
2023-12-09	198.32	330.12	467.13
2023-12-10	199.56	327.31	471.52
2023-12-12	200.35	336.87	465.19
2023-12-11	201.19	339.51	466.31
2023-12-15	201.47	339.12	468.53
2023-12-13	201.88	338.12	466.12
2023-12-14	202.52	340.53	470.67
2023-12-16	203.51	342.90	469.09
2023-12-17	204.76	345.19	467.13

Sorting data in descending order.

In [64]:
```
df.sort_values(by= "AAPL", ascending= False)
```

Out[64]:

	AAPL	META	NVDA
2023-12-17	204.76	345.19	467.13
2023-12-16	203.51	342.90	469.09
2023-12-14	202.52	340.53	470.67
2023-12-13	201.88	338.12	466.12
2023-12-15	201.47	339.12	468.53
2023-12-11	201.19	339.51	466.31
2023-12-12	200.35	336.87	465.19
2023-12-10	199.56	327.31	471.52
2023-12-09	198.32	330.12	467.13
2023-12-08	194.27	326.59	465.97

Checking the Data in the First n Row.

For the last n row, you can use .tail(n) instead.

In [65]:
```
df.head()
#df.tail()
```

Out[65]:

	AAPL	META	NVDA
2023-12-08	194.27	326.59	465.97
2023-12-09	198.32	330.12	467.13
2023-12-10	199.56	327.31	471.52
2023-12-11	201.19	339.51	466.31
2023-12-12	200.35	336.87	465.19

In [66]:
```
df.head(3)
#df.tail(3)
```

Out[66]:

	AAPL	META	NVDA
2023-12-08	194.27	326.59	465.97
2023-12-09	198.32	330.12	467.13
2023-12-10	199.56	327.31	471.52

3. Working with Missing Data

Checking Missing Values

In [67]:
```
df.isnull()
```

Out[67]:

	AAPL	META	NVDA
2023-12-08	False	False	False
2023-12-09	False	False	False
2023-12-10	False	False	False
2023-12-11	False	False	False
2023-12-12	False	False	False
2023-12-13	False	False	False
2023-12-14	False	False	False
2023-12-15	False	False	False
2023-12-16	False	False	False
2023-12-17	False	False	False

When dealing with a large amount of data, the above method may seem a bit clumsy. We can use sum() to quickly check the number of missing values.

In [68]:
```
df.isnull().sum()
```

Out[68]:
```
AAPL    0
META    0
NVDA    0
dtype: int64
```

Let's use a new dataset, a DataFrame containing product names, prices, and quantities but contains some issues.

In [69]:
```
from numpy import nan
data = {"Product":["A", "B", "C", "D", "E", "F", "F", "G", "H"],
        "Price": [25, 50, 25, 25, nan, 40, 40, 25, 15],
        "Quantity":[4, 2, nan, 9, 4, 6, 6, 4, nan] ,
        "GP%" : [0.2, 0.15, 0.2, 0.15, 0.4, 0.15 ,0.2, 0.5, 0.2]
}
data = pd.DataFrame(data, index=data["Product"])
data.drop("Product", axis=1, inplace= True)
data
```

Out[69]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	NaN	0.20
D	25.0	9.0	0.15
E	NaN	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	NaN	0.20

We can see that there are some NaN values.
dropna() will delete rows that contain missing values.

In [70]:
```
data.dropna() #Noticed that we did not specified the argument inplace = True
```

Out[70]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
D	25.0	9.0	0.15
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50

If we set the argument how = "all", it will only delete rows where all values are missing.

In [71]:
```
data.dropna(how="all")
```

Out[71]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	NaN	0.20
D	25.0	9.0	0.15
E	NaN	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	NaN	0.20

Replacing Missing Values

In [72]:
```
data.fillna(0)
```

Out[72]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	0.0	0.20
D	25.0	9.0	0.15
E	0.0	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	0.0	0.20

In practice, we often use specific values to replace missing values, such as the mean. Of course, it depends on the type of data.
In the Quantity column, we use the mean to replace missing values. As for the Price column, we use 15.

In [73]:
```
data["Quantity"].fillna(data["Quantity"].mean(), inplace= True)
data["Price"].fillna(15, inplace= True)
data
```

Out[73]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	5.0	0.20
D	25.0	9.0	0.15
E	15.0	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	5.0	0.20

4. Working wtih Duplicate Data

Checking Duplicate Values

In [74]:
```
duplicate_values = data.duplicated()
print(duplicate_values)
```

```
A    False
B    False
C    False
D    False
E    False
F    False
F    False
G    False
H    False
dtype: bool
```

The default method of the duplicated() is to return True if the entire row is duplicated, and it will not return True if only part of it is duplicated.

We can use the index.duplicated() to check if there are duplicate values in the product columns(Index).

In [75]:
```
duplicate_index = data.index.duplicated()
print(f"Index has duplicates: {duplicate_index}")
```

```
Index has duplicates: [False False False False False False  True False Fals
e]
```

You can add some code to meet your specific needs, such as any(), sum(), etc.

In [76]:
```
for i in data.columns:
    duplicate_values = data[i].duplicated().sum()
    print(f"Column '{i}' has duplicates: {duplicate_values}")
```

```
Column 'Price' has duplicates: 5
Column 'Quantity' has duplicates: 4
Column 'GP%' has duplicates: 5
```

Dropping Duplicate Values

In [77]:
```
data.drop_duplicates(inplace=True)
data
#data.drop_duplicates(keep="last") ## Keep only the last duplicate values.
#df.drop_duplicates(keep="False") ## Delete all duplicate values.
```

Out[77]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	5.0	0.20
D	25.0	9.0	0.15
E	15.0	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	5.0	0.20

5. Renaming a Column

In [78]:
```
data.rename(columns= {"Quantity": "Number"})
data
```

Out[78]:

	Price	Quantity	GP%
A	25.0	4.0	0.20
B	50.0	2.0	0.15
C	25.0	5.0	0.20
D	25.0	9.0	0.15
E	15.0	4.0	0.40
F	40.0	6.0	0.15
F	40.0	6.0	0.20
G	25.0	4.0	0.50
H	15.0	5.0	0.20

6. Grouping Data

Groupby

In [79]:
```
price= data.groupby("Price")
type(price)
```

Out[79]:
```
pandas.core.groupby.generic.DataFrameGroupBy
```

In [80]:
```
price.size() #This is equivalent to data["price"].value_counts
```

Out[80]:
```
Price
15.0    2
25.0    4
40.0    2
50.0    1
dtype: int64
```

We can obtain the quantities for various-priced products easily with groupby.

In [81]:
```
price.sum()
```

Out[81]:

	Quantity	GP%
Price		
15.0	9.0	0.60
25.0	22.0	1.05
40.0	12.0	0.35
50.0	2.0	0.15

Aggregate

Pandas provides the aggregate() method with agg() as alias, which allows for quick summarization and calculation of column data.

In [82]:
```
data.groupby("Price")["Quantity"].agg("sum") #This is equivalent to the quan
```

Out[82]:
```
Price
15.0     9.0
25.0    22.0
40.0    12.0
50.0     2.0
Name: Quantity, dtype: float64
```

Using aggregate method, we can easily trigger out information.
Such as the median price and quantity among products with different gross profit margins, this could provide some insights for decision-making.

In [83]:
```
data.groupby("GP%").agg(["median"])
```

Out[83]:

	Price	Quantity
	median	median
GP%		
0.15	40.0	6.0
0.20	25.0	5.0
0.40	15.0	4.0
0.50	25.0	4.0

7. Data Overview

We come back to our df, stock price data.

Info

Using info() to obtain a brief summary of the DataFrame. It is very convenient for exploratory data analysis.

In [84]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, 2023-12-08 to 2023-12-17
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AAPL    10 non-null     float64
 1   META    10 non-null     float64
 2   NVDA    10 non-null     float64
dtypes: float64(3)
memory usage: 620.0+ bytes
```

Changing the Datatype

In [85]:
```
df["AAPL"]= df["AAPL"].astype(int)
df.head()
```

Out[85]:

	AAPL	META	NVDA
2023-12-08	194	326.59	465.97
2023-12-09	198	330.12	467.13
2023-12-10	199	327.31	471.52
2023-12-11	201	339.51	466.31
2023-12-12	200	336.87	465.19

Describe

Using describe() to view some basic discriptive statistics details.

In [86]:
```
df.describe()
```

Out[86]:

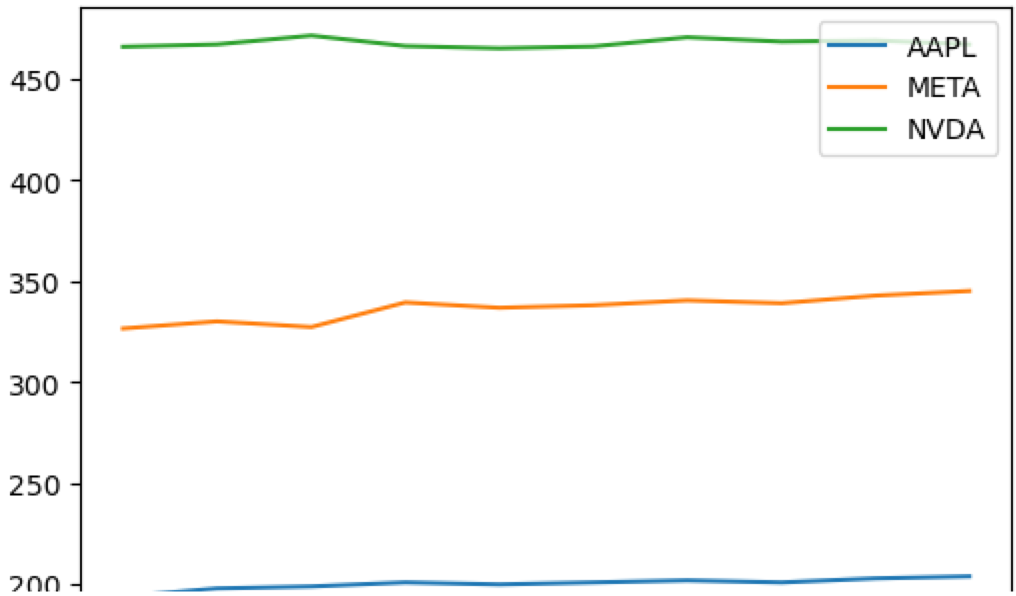
	AAPL	META	NVDA
count	10.000000	10.000000	10.000000
mean	200.300000	336.626000	467.766000
std	2.830391	6.451638	2.117704
min	194.000000	326.590000	465.190000
25%	199.250000	331.807500	466.167500
50%	201.000000	338.620000	467.130000
75%	201.750000	340.275000	468.950000
max	204.000000	345.190000	471.520000

8. Basic Visualization With Pandas

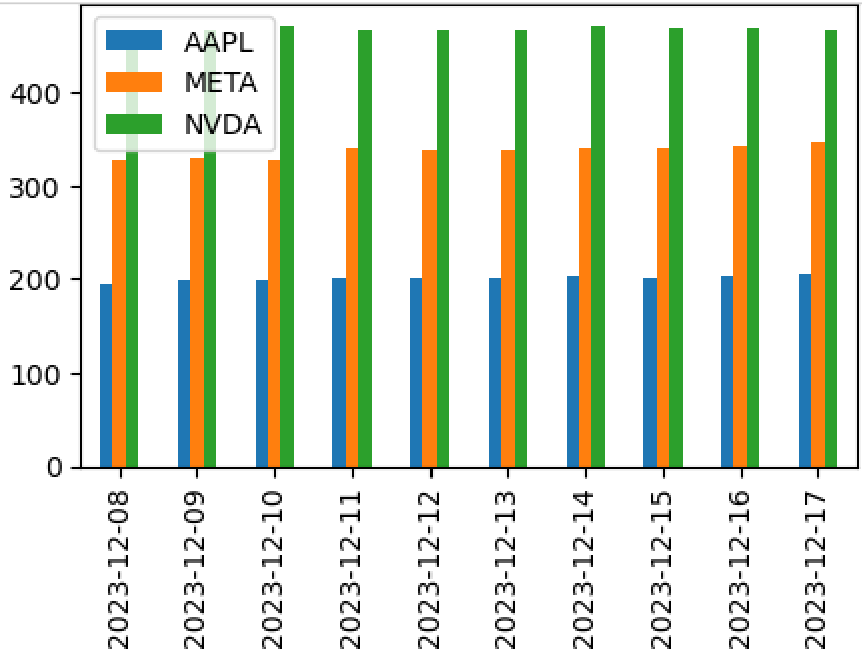
Pandas also provides some basic plotting functions to quickly generate simple plots.
But for more advanced and complex functionalities, you can used other libraries such as Matplotlib, Seaborn, etc.

In [87]:
```
#df.plot(x="Category", y="Value", figsize=(a, b))
df.plot(figsize = (6,4))
# This is equivalent to df.plot.line()
# If not specified arguments, Python will automatically identify and apply t
```

Out[87]: <AxesSubplot:>



In [88]:
```
df.plot.bar(figsize=(5,3))
#df.plot.bar()
#df.plot.box()
#df.plot.hist()
#......
```



Conclusion

This document introduces many essential but useful techniques in NumPy and Pandas. You now have an basic understanding of data structures such as NumPy Arrays, Series, DataFrame, and have mastered various data manipulation skills.

Equipped with this knowledge, you will be able to quickly get started and demonstrate more efficient workstyle in your data analysis projects. I hope this document helps you solidify your foundation and achieve excellent results by leverging these 2 libraries.

- Follow my LinkedIn for more information.: https://www.linkedin.com/in/weihsin-hsu/ (https://www.linkedin.com/in/weihsin-hsu/)
- Follow my GitHub for more programming projects.: https://github.com/endlessnoc (https://github.com/endlessnoc)