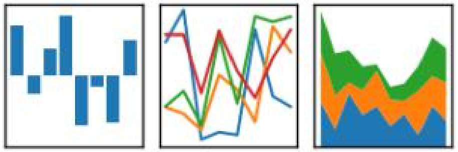


START WITH PANDAS

Pandas Introduction

- Pandas is a software library written for the Python programming language for data manipulation and analysis.
- It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
-

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

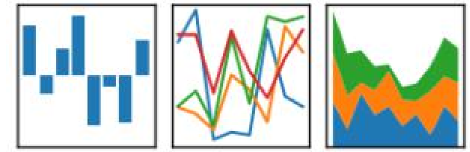
Pandas Introduction

- While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data.
- Often,import convention for pandas:

```
In [1]: import pandas as pd
```

- Import Series and DataFrame into the local namespace:

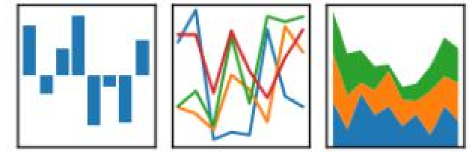
```
In [2]: from pandas import Series, DataFrame
```



Series Introduction

- A Series is a **one-dimensional array-like object** containing a sequence of values (of **similar types** to NumPy types) and an associated array of data labels,**index**.
- Since not specifying an index for the data, a default one consisting of the **integers 0 through N - 1** is created.

```
s = pd.Series(data, index=index)
```



Series Introduction

- From ndarray

```
s = pd.Series(np.random.randn(5), index=['a', 'b',
'c', 'd', 'e'])
```

```
a     2.250327
b     0.684567
c     1.210300
d    -0.226606
e    -1.545200
dtype: float64
```

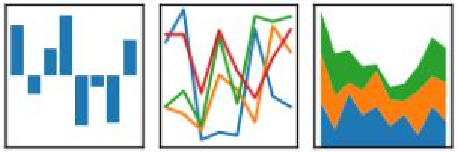
Series Introduction

- From dict

```
d = {'a' : 0., 'b' : 1., 'c' : 2.}

pd.Series(d)

d1=pd.Series(d, index=['b', 'c', 'd', 'a'])
```

```
a      0.0
b      1.0
c      2.0
dtype: float64
```

```
b      1.0
c      2.0
d      NaN
a      0.0
dtype: float64
```



Series Introduction

- From scalar value

```
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

```
a     5.0
b     5.0
c     5.0
d     5.0
e     5.0
dtype: float64
```

Series Introduction

- **Series** vs **ndarray.**

```
In [1]: import numpy as np
        import pandas as pd
        np.array([[1,1,1], [2,2,2]])

Out[1]: array([[1, 1, 1],
               [2, 2, 2]])

In [2]: np.array([[1,1,1], [2,2,2]]).shape

Out[2]: (2, 3)

In [3]: pd.Series([[1,1,1],[2,2,2]])

Out[3]: 0    [1, 1, 1]
        1    [2, 2, 2]
        dtype: object

In [5]: pd.Series([[1,1,1],[2,2,2]]).values.shape

Out[5]: (2,)
```

Series Introduction

Observe the inner data type , n-dimension

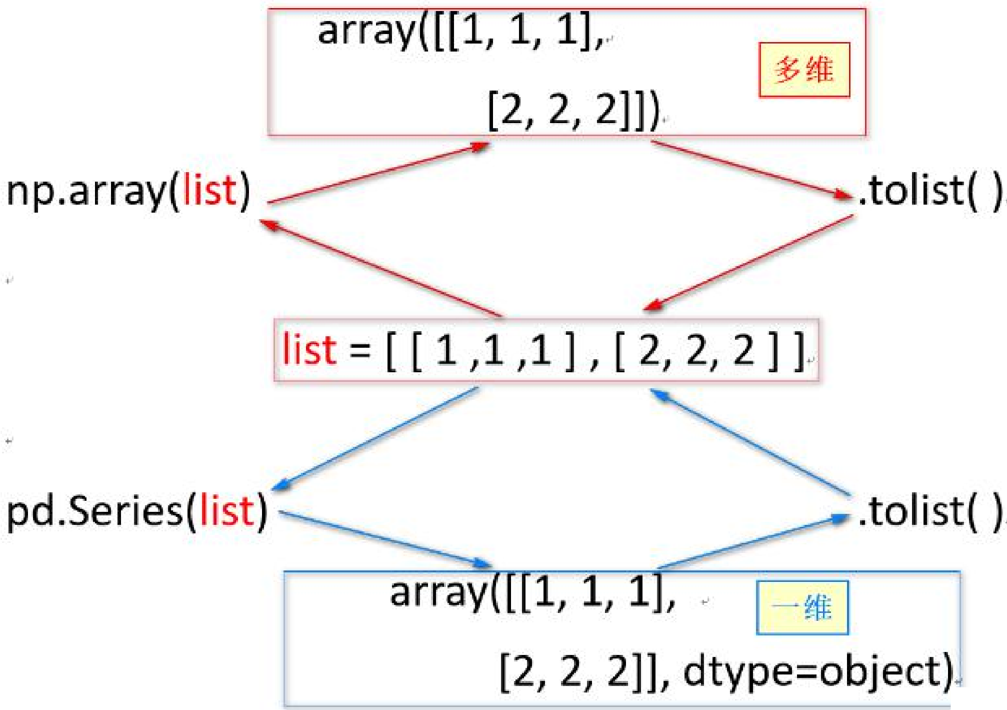
ndarray

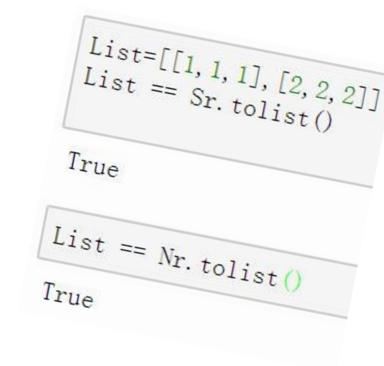
list = [[1 ,1 ,1] , [2, 2, 2]]

Observe the outer data type , one dimension

series

Series Introduction





Series

- Series can use most of NumPy functions.

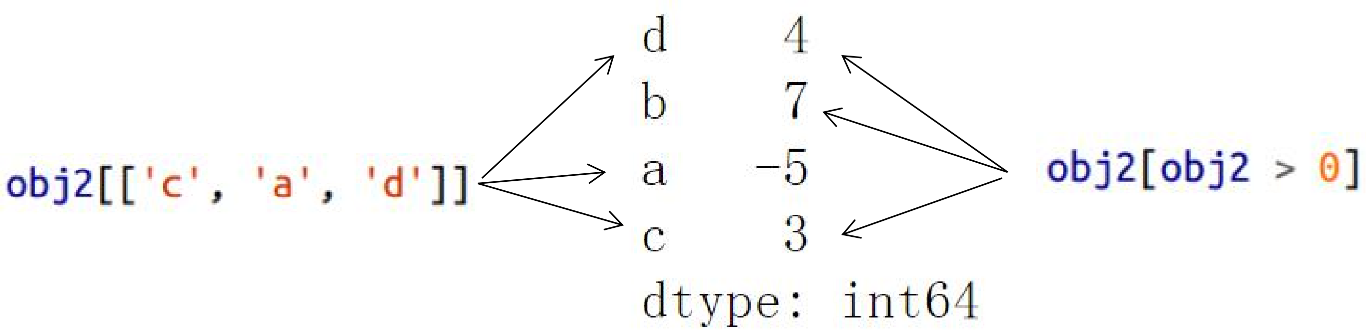
```
In [22]: obj2 * 2

d      8
b     14
a    -10
c      6
dtype: int64
```

```
obj2[obj2>obj2.median()]

d     4
b     7
dtype: int64
```

Series

- Use labels in the index when **selecting single values** or **a set** of values:

```
obj2 = pd.Series([4, 7, -5, 3], index=['d',
'b', 'a', 'c'])
```



Series

- Also, series is **dic-like**.

```
d       4
b       7
a      -5
c       3
dtype: int64
```

```
obj2['a']
```

```
'b' in obj2
```

True

```
'f' in obj2
```

False

Series

- You can **create** a Series **from a python dictionary**.
- When **only** passing a dict, the **index** in the resulting Series will have **the dict's keys** in sorted order.

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

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

Series

- Passing **the dict keys** in the order you want them to appear in the resulting Series:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)
```

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

NaN (not a number), which is considered in pandas to mark missing or NA values.

Series

- Using the terms **“missing”** or **“NA”** interchangeably to refer to missing data. The `isnull` and `notnull` functions in pandas is used to detect missing data:

```
In [32]: pd.isnull(obj4)        In [34]: obj4.isnull()
```

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

Series

- **A useful Series feature**:it automatically aligns by index label in **arithmetic operations**:

```
In [37]: obj3 + obj4
```

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

+

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

```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Series

- Both the **Series object itself** and **its index** have a name attribute, which integrates with other key areas of pandas functionality:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'
```

```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series

- A Series's **index** can be **altered in-place** by assignment:

```
In [42]: obj4.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

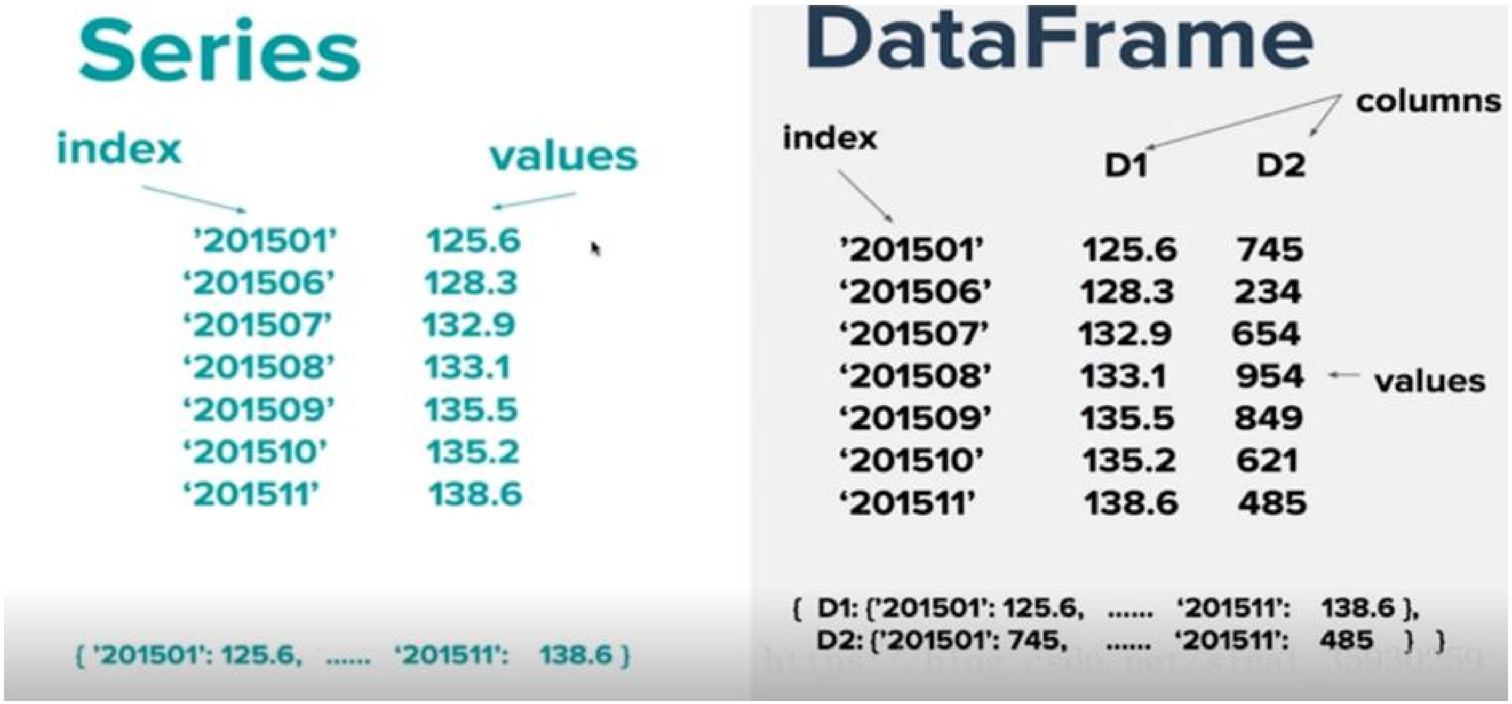
```
Bob           NaN
Steve     35000.0
Jeff      16000.0
Ryan      71000.0
Name: population, dtype: float64
```

DataFrame Introduction

- A DataFrame represents a **rectangular table** of data, which has both a **row** and **column** index.
- It contains an **ordered collection of columns**, which can be different value types.
- The data is stored as one or more **two-dimensional blocks**.

DataFrame Introduction

- **DataFrame** vs **Series**



DataFrame Introduction

- **Construct a DataFrame** from a **dict** of equal-length lists or **NumPy arrays**:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002
5	3.2	Nevada	2003

DataFrame Introduction

- **Construct a DataFrame** from a **nested dict** of dicts，the **outer dict keys** will be the columns and the **inner keys** as the row indices:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

In [66]: frame3 = pd.DataFrame(pop)
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

DataFrame Introduction

- **Construct a DataFrame** fromDicts of Series.

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

DataFrame Introduction

- Possible data inputs to DataFrame constructor.

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

DataFrame

- For large DataFrames, the `head` method selects only the first five rows.
- And the sequence of the DataFrame' s columns can be specified.

```
In [41]: frame2=frame.head()
         pd.DataFrame(frame2, columns=['year', 'state', 'pop'])
```

Out[41]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9

DataFrame

- A column in a DataFrame can be **retrieved as a Series.**

```
In [51]: frame2['state']

0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
Name: state, dtype: object
```

```
In [52]: frame2.year

0    2000
1    2001
2    2002
3    2001
4    2002
Name: year, dtype: int64
```

NOTE: `frame2[column]` works for any column name, but `frame2.column` **only** works when the column name is a valid Python variablename.

DataFrame

- Rows can also be retrieved by position or name with the `loc` attribute：

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada',
'Nevada', 'Nevada'],'year': [2000, 2001, 2002,2001,
2002, 2003],'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame3=pd.DataFrame(data,index=['one','two','three'
,'four','five','six'],columns=['year', 'state',
'pop'])

frame3.loc['three']
```

```
year      2002
state     Ohio
pop        3.6
Name: three, dtype: object
```

DataFrame

- Assigning lists or arrays to a column,espeacially to the empty column.

```
In [59]: frame3['debt']=16.5
```

	year	state	pop
one	2000	Ohio	1.5
two	2001	Ohio	1.7
three	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9
six	2003	Nevada	3.2

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

DataFrame

In [62]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame3['debt'] = val
```

	year	state	pop
one	2000	Ohio	1.5
two	2001	Ohio	1.7
three	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9
six	2003	Nevada	3.2

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

DataFrame

- We can delete columns using `del` keyword :

```
In [64]: del frame3['pop']
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

	year	state	debt
one	2000	Ohio	NaN
two	2001	Ohio	-1.2
three	2002	Ohio	NaN
four	2001	Nevada	-1.5
five	2002	Nevada	-1.7
six	2003	Nevada	NaN

DataFrame

- Also,We can use `drop` :

```
In  [73]: frame3.drop(columns=['pop'])
          frame3.drop(['one', 'six'])
```

	year	state	pop
one	2000	Ohio	1.5
two	2001	Ohio	1.7
three	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9
six	2003	Nevada	3.2

	year	state	pop
two	2001	Ohio	1.7
three	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9

DataFrame

- A DataFrame‘s index and columns **have their name attributes set**，as the following:

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'
```

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

DataFrame

- The DataFrame can **swap rows and columns** using `frame3.T`:

```
In [75]: frame3.T
```

	year	state	pop
one	2000	Ohio	1.5
three	2001	Ohio	1.7
two	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9
six	2003	Nevada	3.2

→

	one	three	two	four	five	six
year	2000	2001	2002	2001	2002	2003
state	Ohio	Ohio	Ohio	Nevada	Nevada	Nevada
pop	1.5	1.7	3.6	2.4	2.9	3.2

DataFrame

- As with Series, the `values` attribute returns the data as a two-dimensional ndarray:

```
In [74]: frame3.values

array([[nan,  1.5],
       [2.4,  1.7],
       [2.9,  3.6]])
```

- If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns.

Excercise

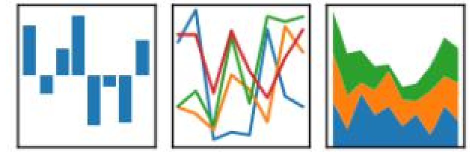
- **For example**：On the table，the data in one column contains two characteristic dimension. How can we split this column into two?

```
df = pd.DataFrame([['Tom','18|男'],['Joho','20|女
'],['Tim','13|女']],columns=['name','age&sex'])
```

	name	age&sex
0	Tom	18|男
1	Joho	20|女
2	Tim	13|女

→

	name	age&sex	age	sex
0	Tom	18|男	18	男
1	Joho	20|女	20	女
2	Tim	13|女	13	女



Excercise

- `df['age&sex'].str.split('|').values` ?

- `List = df['age&sex'].str.split('|').tolist()` ?

- `df['age'],df['sex'] = pd.Series(),pd.Series()` ?

  `df[['age','sex']] = List` ?

Index Objects

- Pandas’ s Index objects are responsible for holding the **axis labels** and other metadata.
  - Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')
```

  - Index objects are **immutable**, thus can’t be modified by the user.

Index Objects

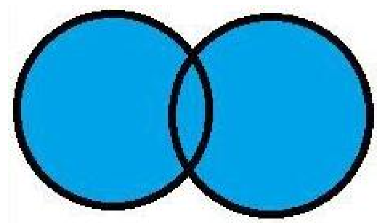
- In addition to being array-like, an **Index** also behaves like a **fixed-size set**:

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

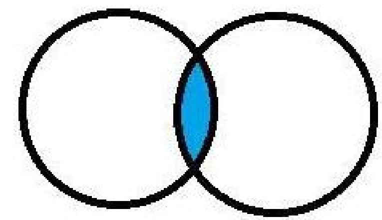
Index Objects

- **set** in python:set \ frozenset

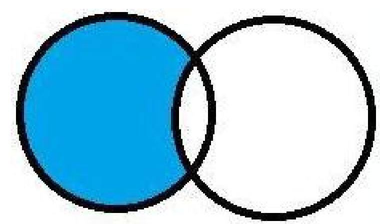
Mathematical Symbol	*Python Symbol*	*Description*
$\in$	**`in`**	Is a member of
$\notin$	**`not in`**	Is not a member of
$=$	`==`	Is equal to
$\neq$	`!=`	Is not equal to
$\subset$	`<`	Is a (strict) subset of
$\subseteq$	`<=`	Is a subset of (includes improper subsets)
$\supset$	`>`	Is a (strict) superset of
$\supseteq$	`>=`	Is a superset of (includes improper supersets)
$\cap$	`&`	Intersection
$\cup$	`|`	Union
$-$ or $\setminus$	`-`	Difference or relative complement
Δ	`^`	Symmetric difference



union



intersect



minus

Index Objects

- Unlike Python sets, a **pandas Index** can contain **duplicate** labels:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Index Objects

Method	Description
`append`	Concatenate with additional Index objects, producing a new Index
`difference`	Compute set difference as an Index
`intersection`	Compute set intersection
`union`	Compute set union
`isin`	Compute boolean array indicating whether each value is contained in the passed collection
`delete`	Compute new Index with element at index `i` deleted
`drop`	Compute new Index by deleting passed values
`insert`	Compute new Index by inserting element at index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique`	Compute the array of unique values in the Index

Index Objects

- `reindex` means to rearrange the data according to the new index.

```
In [92]: obj
Out[92]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Essential Functionality – Reindexing

- For ordered data like **time series**, it may be desirable to do some **interpolation** or **filling** of values when **reindexing**.
- The **method** option allows us to do this,for example `ffill`:

```
In [97]: obj3.reindex(range(6), method='ffill')
```

```
0       blue
2     purple
4     yellow
dtype: object
```

→

```
0       blue
1       blue
2     purple
3     purple
4     yellow
5     yellow
dtype: object
```

Essential Functionality – Reindexing

- With DataFrame, **reindex** can alter either the (row) index, columns, or both.

```
In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

Essential Functionality – Reindexing

- The columns can be **reindexed** with the columns keyword:

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Essential Functionality – Reindexing

- Also，you can label-indexing with `loc`.

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
```

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

Dropping Entries from an Axis

- `drop` method will return a new object with the indicated value or values **deleted from an axis**.

```
In [107]: new_obj = obj.drop('c')
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Dropping Entries from an Axis

- With DataFrame, index values can be deleted from either axis.

```
In [112]: data.drop(['Colorado', 'Ohio'])

In [114]: data.drop(['two', 'four'], axis='columns')
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

Indexing, Selection, and Filtering

- Series indexing works analogously to NumPy array indexing, except youcan use the **Series's index values** instead of only integers.

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
In [122]: obj[['b', 'a', 'd']]
In [124]: obj[obj < 2]
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
a    0.0
b    1.0
dtype: float64
```

Indexing, Selection, and Filtering

- Slicing:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [133]: data[data['three'] > 5]
Out[133]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Indexing, Selection, and Filtering

- Another use case：indexing with a boolean DataFrame.

```
In [134]: data < 5
Out[134]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Selection with loc and iloc

- `loc` and `iloc` enable you to select a subset of the rows and columns from a DataFrame.

```
In [137]: data.loc['Colorado', ['two', 'three']]
```

```
two      5
three    6
Name: Colorado, dtype: int32
```

Selection with loc and iloc

```
In [139]: data.iloc[2]
```

```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

```
In [140]: data.iloc[[1, 2], [3, 0, 1]]
```

	four	one	two
Colorado	7	4	5
Utah	11	8	9

```
In [142]: data.iloc[:, :3][data.three > 5]
```

	one	two	three
Colorado	4	5	6
Utah	8	9	10
New York	12	13	14

Arithmetic and Data Alignment

- When adding together objects, if any **index pairs** are **not** the **same**, the respective index in the result will be the **union** of the index pairs.

```
a     7.3
c    -2.5
d     3.4
e     1.5
dtype: float64
```

+

```
a    -2.1
c     3.6
e    -1.5
f     4.0
g     3.1
dtype: float64
```

```
a     5.2
c     1.1
d     NaN
e     0.0
f     NaN
g     NaN
dtype: float64
```

introduce missing values in the label locations that don‘t overlap

Arithmetic and Data Alignment

- In the case of DataFrame, alignment is performed on both the **rows** and the **columns**:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

+

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

=

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Arithmetic and Data Alignment

- If you add DataFrame objects with **no column** or **row labels** in common, the result will contain **all nulls**:

	A
0	1
1	2

−

	B
0	3
1	4

→

	A	B
0	NaN	NaN
1	NaN	NaN

Arithmetic methods with fill values

- In arithmetic operations, when an axis label is found in one object but not the other,you might want to **fill with a special value, like 0**.

df1

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

df2

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Arithmetic methods with fill values

Method1:

```
In [167]: df2.loc[1, 'b'] = np.nan
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Method2:

```
In [171]: df1.add(df2, fill_value=0)
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Arithmetic methods with fill values

- When **reindexing** a Series or DataFrame, you can also specify a different fill value.

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

Arithmetic methods with fill values

Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Operations between DataFrame and Series

- Arithmetic between DataFrame andSeries is also defined.

Suppose: `arr` →

```
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.]])
```

`arr-arr[0]` →

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

Operations between DataFrame and Series

- Like the above, operations between a DataFrame and a Series are similar.

frame

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

−

series

```
b      0.0
d      1.0
e      2.0
Name: Utah, dtype: float64
```

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Match the index and broadcasting down the rows

Operations between DataFrame and Series

- If an index value is **not found** in either the DataFrame's columns or the Series's index,the objects will be reindexed to form the union.

frame

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

+

series2

```
b      0
e      1
f      2
dtype: int64
```

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

Operations between DataFrame and Series

- If you want to match on the rows,not over the columns ,the following methods will be used.

```
In [189]: frame.sub(series3, axis='index')
```

frame

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

series3

```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

Function Application and Mapping

- NumPy ufuncs (element-wise array methods) also work with pandas objects,like the following:

```
In [192]: np.abs(frame)
```

- Another frequent operation is applying a function on one-dimensional arrays to each column or row.

```
In [193]: f = lambda x: x.max() - x.min()
In [194]: frame.apply(f)
```

	b	d	e
Utah	-1.021910	-0.152804	-0.494643
Ohio	-1.797998	1.155429	1.045093
Texas	-0.565406	0.848529	-0.057742
Oregon	-0.389400	-0.229348	-0.394567

→

```
b       1.408598
d       1.384777
e       1.539736
dtype: float64
```

Function Application and Mapping

- The function passed to apply can also **return a Series** with multiple values.

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
```

	b	d	e
min	-1.797998	-0.229348	-0.494643
max	-0.389400	1.155429	1.045093

Function Application and Mapping

- **Element-wise Python functions**:Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with `applymap`:

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
```

	b	d	e
Utah	-1.02	-0.15	-0.49
Ohio	-1.80	1.16	1.05
Texas	-0.57	0.85	-0.06
Oregon	-0.39	-0.23	-0.39

What about `frame['e'].map(format)`?

Sorting and Ranking

- Another important **built-in operation**:**sort** by row or column index, use the `sort_index`,`sort_values` method.
- With a DataFrame, you can sort by index on either axis.

Sorting and Ranking

```
d     0
a     1
b     2
c     3
dtype: int64
```

`obj.sort_index()`

```
a     1
b     2
c     3
d     0
dtype: int64
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

`frame.sort_index()`

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

`frame.sort_index(axis=1)`

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

Sorting and Ranking

```
d      0
a      1
b      2
c      3
dtype: int64
```

`obj.sort_values()` →

```
d      0
a      1
b      2
c      3
dtype: int64
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

`frame.sort_values(by=['a', 'b'])` →

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

Sorting and Ranking

- **Ranking** assigns ranks from one through the number of valid data points in an array.
- By default rank breaks ties by assigning each group **the mean rank**.

```
DataFrame.rank(axis=0, method='average',
numeric_only=None, na_option='keep',
ascending=True, pct=False)
```

Sorting and Ranking

Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group
'min'	Use the minimum rank for the whole group
'max'	Use the maximum rank for the whole group
'first'	Assign ranks in the order the values appear in the data
'dense'	Like method='min', but ranks always increase by 1 in between groups rather than the number of equal elements in a group

Sorting and Ranking

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [216]: obj.rank()
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

```
In [218]: obj.rank(ascending=False, method='max')
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

Sorting and Ranking

- **DataFrame** can compute **ranks** over the rows or the columns:

```
In [221]: frame.rank(axis='columns')
```

	a	b	c
0	0	4.3	-2.0
1	1	7.0	5.0
2	0	-3.0	8.0
3	1	2.0	-2.5

	a	b	c
0	2.0	3.0	1.0
1	1.0	3.0	2.0
2	2.0	1.0	3.0
3	2.0	3.0	1.0

Return ranks

Axis Indexes with Duplicate Labels

- While many pandas functions (like reindex) require the unique labels, it’s not mandatory.
- Consider a small Series with duplicate indices:

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
```

```
In [225]: obj['a']

a    0
a    1
dtype: int64
```

```
In [226]: obj['c']

4
```

The output type from indexing can vary based on whether a label is repeated or not

Sorting and Ranking

- The same logic extends to **indexing** rows in a **DataFrame**:

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
In [229]: df.loc['b']
```

	0	1	2
a	-0.175280	0.821154	0.209438
a	-0.446488	0.400457	-0.115591
b	-1.629132	-0.948003	0.400754
b	0.662287	-0.859950	-0.738493

	0	1	2
b	-1.629132	-0.948003	0.400754
b	0.662287	-0.859950	-0.738493

Summarizing and Computing Descriptive Statistics

- **Reductions or summary statistics methods** extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame.
- Calling DataFrame's sum method,`df.sum()`, `df.mean(axis='columns',skipna=False)` returns a Series containing column sums.

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
one     9.25
two    -5.80
dtype: float64
```

```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

Summarizing and Computing Descriptive Statistics

Options for reduction methods

Method	Description
`axis`	Axis to reduce over; 0 for DataFrame's rows and 1 for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Summarizing and Computing Descriptive Statistics

- Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [235]: df.idxmax()
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
one      b
two      d
dtype: object
```

Summarizing and Computing Descriptive Statistics

- Other methods like accumulations.

```
In [236]: df.cumsum()
```

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

Summarizing and Computing Descriptive Statistics

- Another type of method, like `describe`, **produce multiple summary statistics** in one shot.

```
In [237]: df.describe()
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

Summarizing and Computing Descriptive Statistics

- On non-numeric data, describe produces **alternative summary statistics**.

```
In [239]: obj.describe()
```

```
0     a
1     a
2     b
3     c
4     a
5     a
6     b
7     c
8     a
9     a
10    b
11    c
12    a
13    a
14    b
15    c
dtype: object
```

```
count     16
unique     3
top        a
freq       8
dtype: object
```

Correlation and Covariance

- Let's consider some DataFrames of **stock prices and volumes** obtained from Yahoo!
- Finance using the add-on pandas-datareader package.

```
conda install pandas-datareader
```

```python
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.items()})
```

Correlation and Covariance

Price.head()

Date	AAPL	GOOG	IBM	MSFT
2010-01-04	27.990226	313.062468	113.304536	25.884104
2010-01-05	28.038618	311.683844	111.935822	25.892466
2010-01-06	27.592626	303.826685	111.208683	25.733566
2010-01-07	27.541619	296.753749	110.823732	25.465944
2010-01-08	27.724725	300.709808	111.935822	25.641571

Volumn.head()

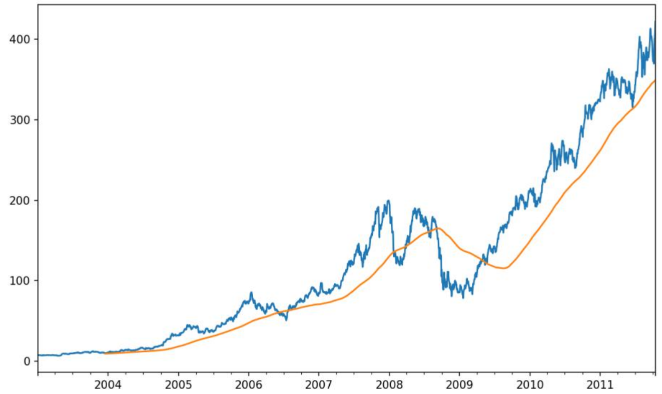
Date	AAPL	GOOG	IBM	MSFT
2010-01-04	123432400	3927000	6155300	38409100
2010-01-05	150476200	6031900	6841400	49749600
2010-01-06	138040000	7987100	5605300	58182400
2010-01-07	119282800	12876600	5840600	50559700
2010-01-08	111902700	9483900	4197200	51197400

Correlation and Covariance

- Now compute percent changes of the prices.

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
```

Date	AAPL	GOOG	IBM	MSFT
2016-10-17	-0.000680	0.001837	0.002072	-0.003483
2016-10-18	-0.000681	0.019616	-0.026168	0.007690
2016-10-19	-0.002979	0.007846	0.003583	-0.002255
2016-10-20	-0.000512	-0.005652	0.001719	-0.004867
2016-10-21	-0.003930	0.003011	-0.012474	0.042096



Correlation and Covariance

- The `corr` method computes the correlation of the overlapping, non-NA,aligned-by-index values in two Series.
- `cov` computes the covariance.

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

Correlation and Covariance

- DataFrame's `corr` and `cov` methods, return a full correlation or covariance matrix as a DataFrame, respectively.

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000
```

Correlation and Covariance

Pearson r correlation:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X-\mu_X)(Y-\mu_Y))}{\sigma_X \sigma_Y} = \frac{E(XY)-E(X)E(Y)}{\sqrt{E(X^2)-E^2(X)}\sqrt{E(Y^2)-E^2(Y)}}$$

Suppose：

y = α+βx+u

COV (u1*u2) =0 ；independent variable ；Var(u|x)=σ^2

Correlation and Covariance

Cohen's standard

		B	B
		Yes	No
A	Yes	20	5
A	No	10	15

- Reader A said "Yes" to 25 applicants and "No" to 25 applicants. Thus reader A said "Yes" 50% of the time.
- Reader B said "Yes" to 30 applicants and "No" to 20 applicants. Thus reader B said "Yes" 60% of the time.

$$\boxed{\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)} = \frac{0.70 - 0.50}{1 - 0.50} = 0.40}$$

Correlation and Covariance

In [37]:
```
returns.corr('spearman')
```
Out[37]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.457218	0.379259	0.431567
GOOG	0.457218	1.000000	0.455885	0.535769
IBM	0.379259	0.455885	1.000000	0.509883
MSFT	0.431567	0.535769	0.509883	1.000000

In [38]:
```
returns.corr('kendall')
```
Out[38]:

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.324028	0.265168	0.305033
GOOG	0.324028	1.000000	0.324124	0.386234
IBM	0.265168	0.324124	1.000000	0.364763
MSFT	0.305033	0.386234	0.364763	1.000000

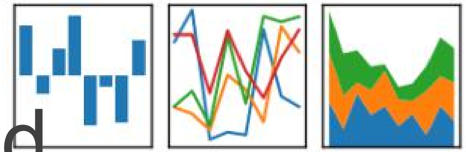
Correlation and Covariance

- Using DataFrame’s `corrwith` method, you can compute pairwise correlations between a DataFrame’s columns or rows with another Series or DataFrame.
  - Passing a Series returns a Series with the correlation value computed for each column.
  - Passing a DataFrame computes the correlations of matching column names.

Correlation and Covariance

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```



Unique Values, Value Counts, and Membership

- **Extract information** about the values contained in a one-dimensional Series.
  - The first function is `unique`, which gives you an array of the unique values in a Series.
  - `value_counts` computes a Series containing value frequencies.
  - `isin` performs a vectorized set membership check and can be useful in filtering a dataset.

Unique Values, Value Counts, and Membership

- In some cases, you may want to compute a **histogram** on multiple related columns in a DataFrame.

```
In [265]: result = data.apply(pd.value_counts).fillna(0)
```

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

→

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

Will you give the picture?

Excercise:Use Pandas visiting xls files

Python2.xls is like the following:

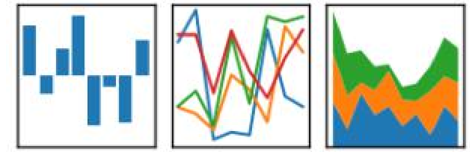
StuNO	Name	Grade	Major
SA18225021	茶健豪	18级大数据与人工智能02班	大数据与人工智能
SA18225022	查顺考	18级大数据与人工智能01班	大数据与人工智能
SA18225023	常承启	18级嵌入式系统设计01班	嵌入式系统设计
SA18225036	陈旻	18级网络与信息安全02班	信息安全工程
SA18225038	陈琦	18级大数据与人工智能02班	大数据与人工智能
SA18225049	陈桢秀	18级嵌入式系统设计01班	嵌入式系统设计
SA18225051	程伟	18级大数据与人工智能01班	大数据与人工智能
SA18225057	邓祥明	18级软件系统设计01班	软件系统设计
SA18225065	段明非	18级软件系统设计01班	软件系统设计
SA18225070	范广宝	18级网络与信息安全01班	信息安全工程
SA18225074	方家辉	18级软件系统设计02班	软件系统设计
SA18225084	甘朔	18级网络与信息安全02班	信息安全工程
SA18225088	高冉	18级软件系统设计01班	软件系统设计
SA18225091	高源	18级大数据与人工智能02班	大数据与人工智能
SA18225111	郝泳杰	18级软件系统设计01班	软件系统设计
SA18225112	何红飞	18级网络与信息安全02班	信息安全工程
SA18225117	何先华	18级软件系统设计02班	软件系统设计
SA18225125	胡瑞云	18级网络与信息安全02班	信息安全工程
SA18225132	黄康晋	18级网络与信息安全01班	信息安全工程
SA18225134	黄磊	18级嵌入式系统设计02班	嵌入式系统设计
SA18225137	黄婷	18级大数据与人工智能01班	大数据与人工智能
SA18225141	季闽城	18级嵌入式系统设计01班	嵌入式系统设计
SA18225157	柯浩	18级大数据与人工智能01班	大数据与人工智能
SA18225161	孔维喆	18级大数据与人工智能02班	大数据与人工智能
SA18225162	匡天宇	18级软件系统设计01班	软件系统设计
SA18225183	李景福	18级嵌入式系统设计01班	嵌入式系统设计
SA18225185	李军	18级软件系统设计01班	软件系统设计

Excercise:Use Pandas visiting xls files

```
import pandas as pd
f=open('D:/Python/Python2.xls','rb')
data=pd.read_excel(f)
```

- When using `data.shape` (103,5) will return

	StuNO	Name	Grade	Major
0	SA18225021	茶健豪	18级大数据与人工智能02班	大数据与人工智能
1	SA18225022	查顺考	18级大数据与人工智能01班	大数据与人工智能
2	SA18225023	常承启	18级嵌入式系统设计01班	嵌入式系统设计
3	SA18225036	陈旻	18级网络与信息安全02班	信息安全工程
4	SA18225038	陈琦	18级大数据与人工智能02班	大数据与人工智能



Excercise:Use Pandas visiting xls files

```
NO_set = set(data['StuNO'])
Name_set = set(data['Name'])
NO_list = []
Name_list = []
for each in NO_set:
    NO_list.append(each)
for each in Name_set:
    Name_list.append(each)
```

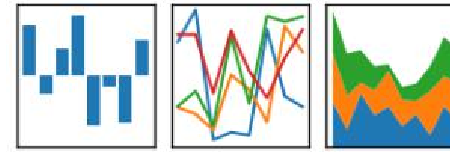
- `NO_list` and `Name_list` will contain the students' NO. and Students' Name on the table.

Excercise:Use Pandas visiting xls files

- Also, we can insert one column into the table.

```
data['Score'] = pd.Series()
Score_list=range(0,103)
data['Score'] = Score_list
```

	StuNO	Name	Grade	Major	Score
0	SA18225021	茶健豪	18级大数据与人工智能02班	大数据与人工智能	0
1	SA18225022	查顺考	18级大数据与人工智能01班	大数据与人工智能	1
2	SA18225023	常承启	18级嵌入式系统设计01班	嵌入式系统设计	2
3	SA18225036	陈旻	18级网络与信息安全02班	信息安全工程	3
4	SA18225038	陈琦	18级大数据与人工智能02班	大数据与人工智能	4
5	SA18225049	陈桢秀	18级嵌入式系统设计01班	嵌入式系统设计	5
6	SA18225051	程伟	18级大数据与人工智能01班	大数据与人工智能	6
7	SA18225057	邓祥明	18级软件系统设计01班	软件系统设计	7
8	SA18225065	段明非	18级软件系统设计01班	软件系统设计	8
9	SA18225070	范广宝	18级网络与信息安全01班	信息安全工程	9



Think About...

- How can we write xls files from a word or txt file?
- How can we use pandas to visit a SQL database?
- How can we modify the dataset back to one database?
- ....

Wish You Practice!