Pandas

- Pandas is a Python package providing fast, flexible, and expressive data structures. It aims to be the fundamental high-level building block for **practical**, **real world data analysis**.
- Pandas is well suited for many different kinds of data:
  - **Tabular data** with **heterogeneously-typed** columns, as in an SQL table or Excel spreadsheet
  - **Ordered** and **unordered** time series data
  - **Arbitrary matrix** with row and column labels
  - Any other form of **observational/statistical data sets**. The data **need not be labeled** at all.
- The two primary data structures of pandas, ***Series***(1D) and ***DataFrame***(2D), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. Pandas is built **on top of NumPy** and is intended to **integrate well within a scientific computing environment** with many other 3rd party libraries.

What Pandas can do?

- Easy handling of ***missing data*** (represented as NaN)
- ***Size mutability***: columns can be inserted and deleted
- Objects can be ***automatically aligned*** to a set of labels
- Powerful, flexible ***group by*** functionality to perform ***split-apply-combine*** operations for aggregating and transforming data
- ***Easy to convert*** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- **Label-based slicing**, **indexing**, and **subsetting** of large data
- Intuitive ***merging*** and ***joining*** data sets
- Flexible ***reshaping*** and ***pivoting*** of data sets
- ***Hierarchical labeling*** of axes(can have multiple labels per tick)
- ***Time series***-specific functionality: date range generation and frequency conversion, date shifting and lagging
- ***Robust IO tools*** for loading data from flat files (CSV and delimited), Excel files, databases, and ultrafast HDF5 format

Most important things

Data alignment is intrinsic

Index/label based operations

Common Sense

Series

- 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, called its ***index***.

```
import pandas as pd

#directly build a series object
obj = pd.Series([4, 7, -5, 3])

#build a series object with explict index
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

#convert a dict to a series object
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)

#convert a dict to a series object with index
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)

#alter the index
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

Questions

#Q1. Can we use the duplicate index?

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

If it works, can we convert it to a dict?

```
dict2=dict(obj2)
```

Very flexible

#Q2. Can the number of index and value be different?

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

#Q3. Can the value or index be different type?

```
obj2 = pd.Series([4, 7, -5, '3'], index=['d', 'b', 'a', 'c'])
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 10])
```

#Q4. How about empty value or empty index?

```
obj2 = pd.Series([4, 7, pd.NA, '3'], index=['d', 'b', 'a', 10])
obj2 = pd.Series([4, 7, pd.NA, '3'], index=['d', 'b', 'a', pd.NA])
```

Index and values

- Two most important attribute of series are **value** and **index**.
- Both the Series object itself and its index have a name attribute

```
In [19]: obj4
Out[19]:
California          NaN
Ohio            35000.0
Oregon          16000.0
Texas           71000.0
dtype: float64
```

```
In [44]: obj4.name='population'

In [45]: obj4.name
Out[45]: 'population'

In [46]: obj4.index.name='state'

In [47]: obj4.index.name
Out[47]: 'state'
```

```
In [20]: obj4.index
Out[20]: Index(['California', 'Ohio', 'Oregon', 'Texas'],
dtype='object')
```

```
In [22]: obj4.values
Out[22]: array([   nan, 35000., 16000., 71000.])
```

Different way to get the value

- Use the index
- Use a mask

```
In [49]: obj4>30000
Out[49]:
state
California    False
Ohio           True
Oregon        False
Texas          True
Name: population, dtype: bool

In [50]: obj4[obj4>30000]
Out[50]:
state
Ohio     35000.0
Texas    71000.0
Name: population, dtype: float64
```

```
In [15]: obj4.values[1]
Out[15]: 35000.0

In [16]: obj4.Ohio
Out[16]: 35000.0

In [17]: obj4['Ohio']
Out[17]: 35000.0

In [18]: obj4['Ohio':'Texas']
Out[18]:
Ohio      35000.0
Oregon    16000.0
Texas     71000.0
dtype: float64

In [19]: obj4['Ohio':'Texas':2]
Out[19]:
Ohio     35000.0
Texas    71000.0
dtype: float64

In [20]: obj4[['Ohio','Texas']]
Out[20]:
Ohio     35000.0
Texas    71000.0
dtype: float64
```

Operations in series

- Using NumPy functions or NumPy-like operations, such as filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve the index-value link.

```
In [32]: obj4*2
Out[32]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
dtype: float64
```

```
In [33]: 'Ohio' in obj4
Out[33]: True

In [34]: 'ohio' in obj4
Out[34]: False
```

```
In [35]: obj4.isnull()
Out[35]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
In [36]: obj4.notnull()
Out[36]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Difference between series and ndarray

- A key difference between Series and ndarray is that operations between Series **automatically align the data based on label**. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.
- The result of an operation between **unaligned Series** will have **the union of the indexes involved**. If a label is not found in one Series or the other, the result will be marked as **missing NaN**.

```
In [37]: obj3
Out[37]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
In [38]: obj4
Out[38]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [39]: obj5=obj3+obj4

In [40]: obj5
Out[40]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Convert to an array

- If you want to do something without index (to disable the auto alignment), you need to convert the series to an array.

```
In [7]: obj4.array
Out[7]:
<PandasArray>
[nan, 35000.0, 16000.0, 71000.0]
Length: 4, dtype: float64
```

- ***Series.array*** is an **ExtensionArray**. Briefly, an ExtensionArray is a thin wrapper around one or more concrete arrays like a numpy.ndarray. pandas knows how to take an ExtensionArray and store it in a Series or a column of a DataFrame.

- While Series is ndarray-like, if you need an actual ndarray, then use Series.to_numpy() and Series.values.

```
In [12]: obj4.to_numpy()
Out[12]: array([   nan, 35000., 16000., 71000.])

In [13]: obj4.values
Out[13]: array([   nan, 35000., 16000., 71000.])
```

DataFrame

- A DataFrame represents a ***rectangular table*** of data and contains an ***ordered*** collection of ***columns***, each of which can be a ***different value type*** (numeric, string, boolean, etc.). The DataFrame has ***both a row and column index***; it can be thought of as a ***dict of Series*** all sharing the same index. Under the hood, the data is stored as one or more 2D blocks rather than a list, dict, or some other collection of 1D arrays.

- While a DataFrame is physically 2D, you can use it to represent ***higher dimensional data*** in a ***tabular format*** using ***hierarchical indexing***.

- There are many ways to construct a DataFrame, though one of the most common is from ***a dict of equal-length lists*** or ***NumPy arrays***. The resulting DataFrame will have its ***index*** assigned automatically as with Series, and the columns are placed in sorted order.

Build a DataFrame object

```
#convert a dict of equal-length lists or numpy array to a DataFrame object
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]}
frame0 = pd.DataFrame(data)
frame1 = pd.DataFrame(np.random.randn(4, 3))

##convert a nested dict of dicts. Pandas will interpret the outer dict keys
##as the columns and the inner keys as the row indices
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
frame2 = pd.DataFrame(pop)
```

```
In [9]: frame0
Out[9]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

```
In [10]: frame0.head()
Out[10]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

```
In [11]: frame0.tail()
Out[11]:
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

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

Index and column

```
#specify a sequence of columns
frame2=pd.DataFrame(data, columns=['year', 'state', 'pop'])

#Pass a column that isn’t contained in data
frame3 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                  index=['one', 'two', 'three', 'four', 'five','six'])

##quesitons
#Q1. Can the number of index and rows of data be different
frame3 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                  index=['one', 'two', 'three', 'four', 'five'])
```

```
In [64]: frame2
Out[64]:
   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
5  2003  Nevada  3.2
```

```
In [65]: frame3
Out[65]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN
```

- Both **index** and **column** can have a name like series

Get the value

- ***frame3[column]*** always works, *but* ***frame3.column*** only works when the column name is a valid Python variable name

```
In [66]: frame3.state
Out[66]:
one          Ohio
two          Ohio
three        Ohio
four       Nevada
five       Nevada
six        Nevada
Name: state, dtype: object
```

```
In [67]: frame3['state']
Out[67]:
one          Ohio
two          Ohio
three        Ohio
four       Nevada
five       Nevada
six        Nevada
Name: state, dtype: object
```

- Rows can be retrieved by ***position*** or ***name*** with the ***iloc or loc*** attribute

```
In [46]: frame3.iloc[0:2]
Out[46]:
     year state  pop  debt
one  2000  Ohio  1.5   6.5
two  2001  Ohio  1.7   6.5

In [47]: frame3[0:2]
Out[47]:
     year state  pop  debt
one  2000  Ohio  1.5   6.5
two  2001  Ohio  1.7   6.5

In [48]: frame3.loc[['one','three']]
Out[48]:
       year state  pop  debt
one    2000  Ohio  1.5   6.5
three  2002  Ohio  3.6   6.5

In [49]: frame3['state']['one']
Out[49]: 'Ohio'
```

loc and iloc

```
#loc and iloc enable you to select a subset of the rows and columns from a DataFrame
#using either axis labels (loc) or integers (iloc)
#Both indexing functions work with slices in addition to single labels or lists of labels
print(data.loc['Colorado', ['two', 'three']]); print(data.loc[:'Utah', 'two'])
print(data.iloc[1, [1,2]]); print(data.iloc[[1,3], [1,2]])
```

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.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value, set_value` methods	Select single value by row and column label

More methods of selections

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

#Series indexing (obj[...]) works analogously to NumPy array indexing
#except you can use the Series's index values instead of only integers

obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(obj['b']); print(obj[1]);
print(obj[['b', 'a', 'd']])
print(obj[[1, 3]])
print(obj[obj < 2])
#Slicing with labels behaves differently than normal Python slicing in that
#the end-point is inclusive
print(obj['b':'c'])

#Setting using these methods modifies the corresponding section of the Series
obj['b':'c'] = 5; print(obj)

#Indexing into a DataFrame is for retrieving one or more columns either with
#a single value or sequence
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
print(data['two']); print(data[['three', 'one']])
print(data[data['three'] > 5])

#change the selected elements
data[data < 5] = 0
print(data)
```

Change the value

- Columns can be modified by assignment.
- When you are assigning lists or arrays to a column, the value's length must **match the length** of the DataFrame.
- If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes.
- Assigning a column that doesn't exist will create a new column.
- The ***del*** keyword will delete columns as with a dict.

```
In [72]: frame3['debt'] = np.arange(0,6,1)

In [73]: frame3
Out[73]:
       year   state  pop  debt
one    2000    Ohio  1.5     0
two    2001    Ohio  1.7     1
three  2002    Ohio  3.6     2
four   2001  Nevada  2.4     3
five   2002  Nevada  2.9     4
six    2003  Nevada  3.2     5
```

```
#add a non-existing column
frame2['eastern'] = frame2.state == 'Ohio'
#delete a column
del frame2['eastern']
```

```
In [77]: val = pd.Series([-1.2, -1.5, -1.7],
    ...: index=['two', 'four', 'five'])
    ...: frame3['debt'] = val

In [78]: frame3
Out[78]:
       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
```

Chained indexing

```
### chained indexing, operation odering matters
df = pd.DataFrame({'a': ['one', 'one', 'two',
                          'three', 'two', 'one', 'six'],
                    'c': np.arange(7)})

labels = pd.Index(['ind0','ind1','ind2','ind3','ind4','ind5','ind6'])
df.index = labels

## Q1. check the following statement
dfa = df.copy()
dfa['c'][0]=0.1
dfa.iloc[0]['c']=1
dfa.loc['ind0']['c']=11
dfa.loc['ind0','c']=111
dfa.iloc[1,1]=1111

## Q2. check the operation orders
dfb = df.copy()

mask = dfb['a'].str.startswith('o')
###When get the values, the following two seems to be the same
print('case 1:', dfb['c'][mask])
print('case 2:', dfb[mask]['c'])

###while, they are different when you try to assign values,
###the case 1 works, while the case 2 does not
dfb['c'][mask] = 42
dfb[mask]['c'] = 24

print(type(dfb['c']))
print(type(dfb[mask]))

## Q3. Better to use loc()
dfc = df.copy()
mask = dfc['a'].str.startswith('o')
dfc.loc[mask,'c'] = 42
```

- When setting values in a pandas object, care must be taken to avoid the ***chained indexing***.
- When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.
- More details

Copy??

- The column returned from indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's copy method.

```
In [100]: frame3
Out[100]:
       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
```

```
In [102]: debt=frame3['debt']

In [103]: type(debt)
Out[103]: pandas.core.series.Series

In [104]: debt.one=10
```

```
In [105]: frame3
Out[105]:
       year   state  pop  debt
one    2000    Ohio  1.5  10.0
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
```

```
In [29]: dd=frame0.copy()
```

```
In [31]: dd['year'][0]=4000
```

```
In [32]: dd
Out[32]:
    state  year  pop
0    Ohio  4000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

```
In [33]: frame0
Out[33]:
    state  year  pop
0    Ohio  3000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

Index object

- Pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index:

- Index objects are ***immutable*** and thus can't be modified. Immutability is important so that Index objects can be safely shared among data structures.

Class	Description
`Index`	The most general Index object, representing axis labels in a NumPy array of Python objects.
`Int64Index`	Specialized Index for integer values.
`MultiIndex`	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
`DatetimeIndex`	Stores nanosecond timestamps (represented using NumPy's `datetime64` dtype).
`PeriodIndex`	Specialized Index for Period data (timespans).

Index method and property

- Each Index has a number of methods and properties for set logic and answering other questions about the data it contains

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	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

Examples of index

```
import pandas as pd
import numpy as np

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


# #Q1:Index objects are immutable and thus can't be modified by the user
# index[1]='d'

labels = pd.Index(np.arange(3))
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
print(id(obj2.index))
print(id(labels))
print(obj2.index is labels)

#In addition to being array-like, an Index also behaves like a fixed-size set
print('a' in index)

#Unlike Python sets, a pandas Index can contain duplicate labels
#Selections with duplicate labels will select all occurrences of that label
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
obj3 = pd.Series([1.5, -2.5, 0, 5], index=dup_labels)
print(obj3['foo'])
```

Functionality: Reindexing

- A critical method on pandas objects is reindex, which means to create a new object with the data conformed to a new index.

- Calling reindex on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present.

- 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. ***ffill*** or ***pad***: Fill (or carry) values forward; ***bfill*** or ***backfill***: Fill (or carry) values backward

reindex function arguments

- With DataFrame, reindex can alter either the (row) index, columns, or both. When passed just a sequence, the rows are reindexed in the result.
- The columns can be reindexed using the columns keyword.
- Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0)
- Reindexing can be done more succinctly with **ix** in old version.

Argument	Description
`index`	New sequence to use as index. Can be `Index` instance or any other sequence-like Python data structure. An `Index` will be used exactly as is without any copying
`method`	Interpolation (fill) method, see Table 5-4 for options.
`fill_value`	Substitute value to use when introducing missing data by reindexing
`limit`	When forward- or backfilling, maximum size gap to fill
`level`	Match simple Index on level of MultiIndex, otherwise select subset of
`copy`	Do not copy underlying data if new index is equivalent to old index. `True` by default (i.e. always copy data).

Examples

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj2 = obj.reindex(['a', 'a', 'b', 'c', 'd', 'e'])

obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj4 = obj3.reindex(range(6), method='ffill')


#With DataFrame, reindex can alter either the (row) index, columns, or both.
frame1 = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'], columns=['Ohio', 'Utah', 'Texas'])
frame2 = frame1.reindex(['a', 'b', 'c', 'd'])

#The columns can be reindexed with the columns keyword
states = ['Utah', 'Texas']
frame3 = frame1.reindex(columns=states)

#you can reindex more succinctly by label-indexing with loc
frame4 = frame1.loc[['a', 'c', 'd'], states]

# ##Q1. Can use reindex the non-exsiting column
# states = ['Texas', 'Utah', 'California']
# frame3 = frame1.reindex(columns=states)
# ##Q2. Can we use loc to reindex the non-existing rows?
# states = ['Utah', 'Texas']
# frame4 = frame1.loc[['a', 'b', 'c', 'd'], states]
```

Multilevel index

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada',
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame0 = pd.DataFrame(data)
```

```
In [160]: frame0
Out[160]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

```
#you can use set_index and reset_index to
#realize multilevel idnex
frame5=frame0.set_index(['state','year'])
frame0_back=frame5.reset_index(level=[0,1])
frame0_back1=frame5.reset_index('state')
frame0_back2=frame0_back1.reset_index('year')
```

```
In [159]: frame5
Out[159]:
             pop
state  year
Ohio   2000  1.5
       2001  1.7
       2002  3.6
Nevada 2001  2.4
       2002  2.9
       2003  3.2
```

```
#different to access the multiple level index
print(frame5.loc['Ohio'].loc[2001])
print(frame5.loc['Ohio',2001])
print(frame5.iloc[2])
```

Hierarchical Indexing

- Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index levels on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form.
- You can use a 2D or high-dimension list as the index.
- With a hierarchically indexed object, so-called partial indexing is possible, enabling you to concisely select subsets of the data

```
In [89]: data = pd.Series(np.linspace(11,19,9), index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],[1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [90]: data
Out[90]:
a  1     11.0
   2     12.0
   3     13.0
b  1     14.0
   3     15.0
c  1     16.0
   2     17.0
d  2     18.0
   3     19.0
dtype: float64
```

```
In [91]: data['a']
Out[91]:
1    11.0
2    12.0
3    13.0
dtype: float64

In [92]: data.loc['a']
Out[92]:
1    11.0
2    12.0
3    13.0
dtype: float64
```

```
In [93]: data.loc['a',:]
Out[93]:
a  1     11.0
   2     12.0
   3     13.0
dtype: float64
```

```
In [98]: data[1]
Out[98]: 12.0

In [99]: data.loc['a',1]
Out[99]: 11.0
```

```
In [101]: data.loc[['a','d'],1]
Out[101]:
a  1     11.0
dtype: float64

In [102]: data.loc[['a','c'],1]
Out[102]:
a  1     11.0
c  1     16.0
dtype: float64
```

Hierarchical Indexing

- Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table.
- You can do the unstack() and stack().
- For DataFrame, either axis can have a hierarchical index

```
In [108]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)), index=[['a', 'a',
'b', 'b'], [1, 2, 1, 2]], columns=[['Ohio', 'Ohio', 'Colorado'], ['Green',
'Red', 'Green']])

In [109]: frame
Out[109]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

```
In [116]: frame.index.names = ['key1', 'key2']
     ...: frame.columns.names = ['state', 'color']

In [117]: frame
Out[117]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

```
In [103]: data.unstack()
Out[103]:
      1     2     3
a  11.0  12.0  13.0
b  14.0   NaN  15.0
c  16.0  17.0   NaN
d   NaN  18.0  19.0

In [104]: data.unstack().stack()
Out[104]:
a  1    11.0
   2    12.0
   3    13.0
b  1    14.0
   3    15.0
c  1    16.0
   2    17.0
d  2    18.0
   3    19.0
dtype: float64
```

- The hierarchical levels can have names .
- A MultiIndex can be created by itself and then reused.

```
In [119]: pd.MultiIndex.from_arrays([['Ohio', 'Ohio',
'Colorado'], ['Green', 'Red',
'Green']],names=['state', 'color'])
Out[119]:
MultiIndex([(    'Ohio', 'Green'),
            (    'Ohio',   'Red'),
            ('Colorado', 'Green')],
           names=['state', 'color'])
```

Deal with Levels

- You can use ***swaplevel()*** to interchange the index.
- You can use ***sort_index()*** to sort the data using only the values in a single level. Many descriptive and summary statistics on have a level option.
- You can ***set_index()*** to create a new DataFrame using one or more of its columns as the index and ***reset_index()*** will do the opposite.

```
In [130]: frame.reset_index('key1')
Out[130]:
state key1  Ohio       Colorado
color      Green Red      Green
key2
1        a     0   1          2
2        a     3   4          5
1        b     6   7          8
2        b     9  10         11
```

```
In [120]: frame.swaplevel('key1', 'key2')
Out[120]:
state        Ohio     Colorado
color       Green Red    Green
key2 key1
1    a          0   1        2
2    a          3   4        5
1    b          6   7        8
2    b          9  10       11

In [121]: frame.sort_index(level=1)
Out[121]:
state        Ohio     Colorado
color       Green Red    Green
key1 key2
a    1          0   1        2
b    1          6   7        8
a    2          3   4        5
b    2          9  10       11

In [122]: frame.sort_index(level=0)
Out[122]:
state        Ohio     Colorado
color       Green Red    Green
key1 key2
a    1          0   1        2
     2          3   4        5
b    1          6   7        8
     2          9  10       11
```

```
In [132]: frame.sum(level=1)
Out[132]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16
```

Dropping and adding Entries from an Axis

```
import pandas as pd
import numpy as np

obj = pd.Series(np.arange(5.), index=['a', 'c', 'c', 'd', 'e'])

#For series, it is pretty simple and straight forward
obj2 = obj.drop('c')
obj3 = obj.drop(['d', 'c'])

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

data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])

data1 = data.drop(['Colorado', 'Ohio'])

#You can drop values from the columns by passing axis=1 or axis='columns'
data2 = data.drop('two', axis=1)
data2 = data.drop('two', axis='columns')

#Many functions, like drop, which modify the size or shape of a Series or
#DataFrame, can manipulate an object in-place without returning a new object
print(obj)
obj.drop('c', inplace=True)
print(obj)

data3 = data2.copy()
### add new column
data3['five']=pd.NA
### add one row by loc
data3.loc['Hong Kong']=[16,17,18,pd.NA]
```

- We can also add multiple rows using the **pandas.concat()**.

Arithmetic and Data Alignment

- An important pandas feature for some applications is the behavior of arithmetic between objects with **different indexes**.
- When you are 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.
- The internal data alignment introduces missing values in the label locations that don't overlap. Missing values will then propagate in further arithmetic computations.

```
In [35]: s1
Out[35]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

```
In [36]: s2
Out[36]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

```
In [37]: s1+s2
Out[37]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

Alignment of DataFrame

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

```
In [39]: df1
Out[39]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [40]: df2
Out[40]:
          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
```

```
In [41]: df1+df2
Out[41]:
            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

In [42]: df1-df2
Out[42]:
            b   c    d   e
Colorado  NaN NaN  NaN NaN
Ohio     -3.0 NaN -2.0 NaN
Oregon    NaN NaN  NaN NaN
Texas    -3.0 NaN -2.0 NaN
Utah      NaN NaN  NaN NaN
```

Arithmetic methods with fill values

- In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found

```
In [50]: df1
Out[50]:
     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

In [51]: df2
Out[51]:
      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
```

```
In [52]: df1+df2
Out[52]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN

In [53]: df1.add(df2, fill_value=0)
Out[53]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  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
```

- Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value. **df1.reindex(columns=df2.columns, fill_value=0)**

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 (**)

```
In [59]: 1/df1
Out[59]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909

In [60]: df1.rdiv(1)
Out[60]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909
```

```
In [61]: df1**4
Out[61]:
        a       b        c        d
0     0.0     1.0     16.0     81.0
1   256.0   625.0   1296.0   2401.0
2  4096.0  6561.0  10000.0  14641.0

In [62]: df1.pow(4)
Out[62]:
        a       b        c        d
0     0.0     1.0     16.0     81.0
1   256.0   625.0   1296.0   2401.0
2  4096.0  6561.0  10000.0  14641.0

In [63]: df1.rpow(4)
Out[63]:
         a         b          c          d
0      1.0       4.0       16.0       64.0
1    256.0    1024.0     4096.0    16384.0
2  65536.0  262144.0  1048576.0  4194304.0
```

Operations between DataFrame and Series

- As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also using ***broadcasting***.
- The operations are performed based on **column values**.

```
In [67]: frame
Out[67]:
          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

In [68]: series
Out[68]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

```
In [69]: frame-series
Out[69]:
          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
```

With non-existing column

- 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

```
In [71]: frame
Out[71]:
          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

In [72]: series2
Out[72]:
b    0
e    1
f    2
dtype: int64
```

```
In [73]: frame - series2
Out[73]:
          b   d     e   f
Utah    0.0 NaN   1.0 NaN
Ohio    3.0 NaN   4.0 NaN
Texas   6.0 NaN   7.0 NaN
Oregon  9.0 NaN  10.0 NaN
```

Non-existing is not Zero

Broadcasting over index

- We can specify the operations over **index**.

```
In [77]: frame
Out[77]:
          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

In [78]: series3
Out[78]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

```
In [79]: frame.sub(series3, axis='index')
Out[79]:
          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
```

- What will happen for the following command?

frame - series3

Function Application and Mapping

```
#NumPy ufuncs (element-wise array methods) also work with pandas objects
###apply function
#f = lambda x: x.max() - x.min()
def f(x):
    return x.max()-x.min()
def f2(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
print(frame.apply(f))
print(frame.max()-frame.min())

print(frame.apply(f2))

format = lambda x: '%.2f' % x
print(frame.applymap(format))

###Series has a function map
print(frame['b'].map(format))
```

Descriptive Statistics

- pandas objects are equipped with a set of common mathematical and statistical methods.

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

Difference between rows and columns

- Naively speaking, there are no fundamental differences between rows and columns. One can assign either dimension or 2D data to be row in the sense of pure math or data.

- However, in real life, the meaning of different dimensions are quite different. For example, we have a table of students' scores for different courses, we usually put different courses as the column and assign different students as different rows. Different courses are like attributes of one students and the number is finite, while students are more like objectives and its number can be as large as possible.

- In pandas, it is the default setting. Therefore, in many cases, you can interpret the operations or functions of Pandas based on this kind of understanding.

Deal with different files

- pandas features a number of functions for reading tabular data as a DataFrame object.

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format