# Pandas

# Pandas

- pandas含有使数据分析工作变得更快更简单的高级数据结构和操作工具。它是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。
- 三大数据结构：Series、DataFrame、Index

# Series

- 带索引的一维数组

```
import pandas as pd

data = pd.Series([0.25,0.5,0.75,1])
print(data)
```

```
0      0.25
1      0.50
2      0.75
3      1.00
dtype: float64
```

# Series

- 索引与数据

```
data = pd.Series([0.25,0.5,0.75,1])
print(data.values)
print(type(data.values))
print(data.index)
print(type(data.index))
```

```
[0.25 0.5  0.75 1.  ]
<class 'numpy.ndarray'>
RangeIndex(start=0, stop=4, step=1)
<class 'pandas.core.indexes.range.RangeIndex'>
```

# Series

- 由字典生成Series对象

```
a_dict = {
    'AAA':23423,
    'BBB':43422,
    'CCC':3334
}

a_ser = pd.Series(a_dict)
print(a_ser)
print(a_ser['BBB'])
```

```
AAA     23423
BBB     43422
CCC      3334
dtype: int64
43422
```

# Series

- 索引参数index中含有字典中不存在的键时，默认设置对应值为NaN

```
a_dict = {
    'AAA':23423,
    'BBB':43422,
    'CCC':3334
}

a_ser = pd.Series(a_dict,index=['AAA','CCC'])
b_ser = pd.Series(a_dict,index=['BBB','DDD'])
print(a_ser)
print(b_ser)
```

```
AAA     23423
CCC      3334
dtype: int64

BBB    43422.0
DDD        NaN
dtype: float64
```

# Series数据选取

- 类比字典

```
data = pd.Series([0.25,0.5,0.75,1.0]
                 ,index=['a','b','c','d'])
print(data['c'])                #通过键来获取字典的值
print('a' in data)              #判断键是否存在
print(data.keys())              #获取键的列表
print(list(data.items()))       #获取键值对的列表
```

```
0.75
True
Index(['a', 'b', 'c', 'd'], dtype='object')
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

# Series数据选取

```
data = pd.Series([0.25,0.5,0.75,1.0]
                 ,index=['a','b','c','d'])
data['e'] = 1.25
print(data)
```

向Series对象添加数据

```
a     0.25
b     0.50
c     0.75
d     1.00
e     1.25
dtype: float64
```

# Series数据选取

- 类比数组

```
data = pd.Series([0.25, 0.5, 0.75, 1.0]
                  ,index=['a','b','c','d'])

print(data['a':'c'])
print(data[0:2])
```

```
a      0.25
b      0.50
c      0.75
dtype: float64

a      0.25
b      0.50
dtype: float64
```

显式索引的切片中，左右两端都包含在结果中

# Series数据选取

```
data = pd.Series([0.25,0.5,0.75,1.0]
                  ,index=['a','b','c','d'])
print(data[(data > 0.2) & (data < 0.6)])
print(data[['a','d']])
```

Index子集索引

布尔数组索引

```
a     0.25
b     0.50
dtype: float64

a     0.25
d     1.00
dtype: float64
```

# Series数据选取

- 显式索引vs隐式索引

```
data = pd.Series([’a’,’b’,’c’,’d’]
                 ,index=[1,3,5,7])
print(data[1]) # 单一索引 - 显式
print(data[1:3]) # 切片索引 - 隐式
```

```
a

3     b
5     c
dtype: object
```

当index的值为数字时容易造成混淆！

用默认位置检索（如data[0]）会报错！

# Series数据选取

- loc索引，指定利用显式索引进行取值和分片操作

```
data = pd.Series(['a','b','c','d']
                 ,index=[1,3,5,7])
print(data.loc[3])
print(data.loc[1:5])
```

```
b

1      a
3      b
5      c
dtype: object
```

# Series数据选取

- iloc索引，表明取值和分片都是采取隐式数值索引

```
data = pd.Series(['a','b','c','d']
                 ,index=[1,3,5,7])
print(data.iloc[2])
print(data.iloc[1:3])
```

```
c

3    b
5    c
dtype: object
```

# DataFrame

- DataFrame是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame既有行索引也有列索引，它可以被看做由Series组成的字典（共用同一个索引）。

# DataFrame

```
area_dict = {'California':423967,'Texas':695662,'New York':141297}
population_dict = {'California':1.3, 'Texas':0.98, 'New York':1.13}
area = pd.Series(area_dict)
population = pd.Series(population_dict)
print(area)
print(population)
```

```
California    423967
Texas         695662
New York      141297
dtype: int64

California    1.30
Texas         0.98
New York      1.13
dtype: float64
```

# DataFrame

```
area_dict = {'California':423967,'Texas':695662,'New York':141297}
population_dict = {'California':1.3, 'Texas':0.98, 'New York':1.13}
area = pd.Series(area_dict)
population = pd.Series(population_dict)
states_df = pd.DataFrame({'area':area,'population':population})
print(states_df)
```

```
              area  population
California  423967        1.30
Texas       695662        0.98
New York    141297        1.13
```

通过Series对象创建DataFrame对象

# DataFrame

- 获取DataFrame对象的行列索引值

```
states_df = pd.DataFrame({'area':area,'population':population})
print(states_df.index)
print(states_df.columns)
```

```
Index(['California', 'Texas', 'New York'], dtype='object')
Index(['area', 'population'], dtype='object')
```

# DataFrame

- 通过字典创建DataFrame对象

```
data = [{'a':13,'b':4},{'a':'CHN','b':'USA'}]
df = pd.DataFrame(data,index=['c','d'])
print(df)
```

```
     a    b
c   13    4
d  CHN  USA
```

# DataFrame

- 通过数组创建DataFrame对象

```
df = pd.DataFrame(np.random.rand(3,2),
                  columns=['foo','bar'],
                  index=['a','b','c'])
print(df)
```

```
        foo       bar
a  0.127044  0.336537
b  0.804913  0.273848
c  0.417295  0.967519
```

# DataFrame数据选取

- 类比字典

```
area = pd.Series({’California’:423967,’Texas’:695662,
                  ’New York’:141297,’Floriade’:170312,
                  ’Illinois’:149995})

pop = pd.Series({’California’:38332521,’Texas’:26448193,
                 ’New York’:19651127,’Floriade’:19552860,
                 ’Illinois’:12882135})

data = pd.DataFrame({’area’:
print(data)
```

```
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Floriade    170312  19552860
Illinois    149995  12882135
```

# DataFrame数据选取

```
data = pd.DataFrame({'area':area, 'pop':pop})
print(data['area'])
```

```
California    423967
Texas         695662
New York      141297
Floriade      170312
Illinois      149995
Name: area, dtype: int64
```

类比于用字典的key来访问其value

# DataFrame数据选取

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['a'] = [1,2,3,4,5]
data['b'] = 0
print(data)
```

用字典的语法形式进行列扩充

| | area | pop | a | b |
|---|---|---|---|---|
| California | 423967 | 38332521 | 1 | 0 |
| Texas | 695662 | 26448193 | 2 | 0 |
| New York | 141297 | 19651127 | 3 | 0 |
| Floriade | 170312 | 19552860 | 4 | 0 |
| Illinois | 149995 | 12882135 | 5 | 0 |

# DataFrame数据选取

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data)
```

```
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Floriade    170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

通常应用中某一列是其他列的计算结果，如人口密度

# DataFrame数据选取

- 类比二维数组

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data.values)
```

```
[[4.23967000e+05 3.83325210e+07 9.04139261e+01]
 [6.95662000e+05 2.64481930e+07 3.80187404e+01]
 [1.41297000e+05 1.96511270e+07 1.39076746e+02]
 [1.70312000e+05 1.95528600e+07 1.14806121e+02]
 [1.49995000e+05 1.28821350e+07 8.58837628e+01]]
```

# DataFrame数据选取

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data.values[2])
print(data.values[1,1])
print(data.values[1:,:2])
```

跟操作二维数组一样

```
[1.41297000e+05 1.96511270e+07 1.39076746e+02]

26448193.0

[[  695662. 26448193.]
 [  141297. 19651127.]
 [  170312. 19552860.]
 [  149995. 12882135.]]
```

# DataFrame数据选取

- iloc索引器，行列都使用隐式索引，我们可以像操作ndarray数组一样，对DataFrame数据类型进行索引分片操作：

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data.iloc[:2,1:2])
```

```
                 pop
California  38332521
Texas       26448193
```

# DataFrame数据选取

- loc索引器，采用显式的标签值索引进行分片，规则是左右都取

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data.loc[:'Floriade', 'area':'pop'])
```

```
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Floriade    170312  19552860
```

# DataFrame数据选取

```
# 人口密度大于100的州
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data[data['density'] > 100])
```

```
              area       pop     density
New York    141297  19651127  139.076746
Floriade    170312  19552860  114.806121
```

# DataFrame数据选取

```
# 人口密度大于100的州，且只看其面积和人口密度
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
print(data.loc[data['density'] > 100, ['area','density']])
```

```
             area     density
New York   141297  139.076746
Floriade   170312  114.806121
```

# DataFrame数据选取

- 如需修改DataFrame中的某个值，使用任何一种索引器方法定位到具体的一个数据项即可

```
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']
data.loc['Floriade', 'area'] = 9999999
data.iloc[4,1] = 8888888
print(data)
```

```
               area       pop     density
California   423967  38332521   90.413926
Texas        695662  26448193   38.018740
New York     141297  19651127  139.076746
Floriade    9999999  19552860  114.806121
Illinois     149995   8888888   85.883763
```

# 数值运算

# 数值运算

- NumPy中使用的一元运算符可以拿到Pandas数据对象中使用

```
ser = pd.Series([2,4,6,8],
                index=['a','b','c','d'])
print(ser)
print(np.exp(ser))
```

```
a    2
b    4
c    6
d    8
dtype: int64
```

```
a       7.389056
b      54.598150
c     403.428793
d    2980.957987
dtype: float64
```

# 数值运算

```
rng = np.random.RandomState(18)
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['a','b','c','d'])
print(df)
print(np.sin(df * np.pi / 4))
```

```
   a  b  c  d
0  3  8  5  1
1  2  2  8  8
2  2  1  5  5
          a             b             c             d
0  0.707107 -2.449294e-16 -7.071068e-01  7.071068e-01
1  1.000000  1.000000e+00 -2.449294e-16 -2.449294e-16
2  1.000000  7.071068e-01 -7.071068e-01 -7.071068e-01
```

# 数值运算

- 二元运算会在计算过程中对齐两个对象的索引

```
ser1 = pd.Series({'a':10,'b':20,'d':40})
ser2 = pd.Series({'b':2,'c':3,'d':4})
print(ser1 / ser2)
```

```
a     NaN
b    10.0
c     NaN
d    10.0
dtype: float64
```

# 数值运算

```
ser1 = pd.Series({'a':10,'b':20,'d':40})
ser2 = pd.Series({'b':2,'c':3,'d':4})
print(ser1.add(ser2, fill_value=0))
```

指定对齐时填充的值

```
a    10.0
b    22.0
c     3.0
d    44.0
dtype: float64
```

```python
rng = np.random.RandomState(10)
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=['A', 'B'])
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=['B', 'C', 'A'])
print(A)
print(B)
print(A + B)
```

```
    A  B
0   9  4
1  15  0

   B  C  A
0  1  9  0
1  1  8  9
2  0  8  6

      A    B   C
0   9.0  5.0 NaN
1  24.0  1.0 NaN
2   NaN  NaN NaN
```

DataFrame对象同样会进行索引对齐

```
rng = np.random.RandomState(10)
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=[’A’, ’B’])
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=[’B’, ’C’, ’A’])

print(A)
print(B)
print(A.add(B, fill_value=A.stack(
```

```
    A  B
0   9  4
1  15  0

   B  C  A
0  1  9  0
1  1  8  9
2  0  8  6

      A    B     C
0   9.0  5.0  16.0
1  24.0  1.0  15.0
2  13.0  7.0  15.0
```

# 数值运算

- Series 与 DataFrame之间运

```
rng = np.random.RandomState(10)
A = pd.DataFrame(rng.randint(0,
                 columns=['A', '

print(A)
print(A.iloc[0])
print(A - A.iloc[0])
```

```
   A  B  C  D
0  9  4  0  1
1  9  0  1  8
2  9  0  8  6

A    9
B    4
C    0
D    1
Name: 0, dtype: int32

   A  B  C  D
0  0  0  0  0
1  0 -4  1  7
2  0 -4  8  5
```

# 数值运算

```
rng = np.random.RandomState(1
A = pd.DataFrame(rng.randint(
                 columns=['A'

print(A)
print(A['B'])
print(A.sub(A['B'], axis=0))
```

```
   A  B  C  D
0  9  4  0  1
1  9  0  1  8
2  9  0  8  6

0    4
1    0
2    0
Name: B, dtype: int32

   A  B  C  D
0  5  0 -4 -3
1  9  0  1  8
2  9  0  8  6
```

# 数值运算

- DataFrame与Series的运算要对齐指定方向上的索引

```
rng = np.random.RandomState(10)
A = pd.DataFrame(rng.randint(0,
                 columns=['A', '

print(A)
print(A.loc[0,['A', 'C']])
print(A - A.loc[0,['A', 'C']])
```

```
   A  B  C  D
0  9  4  0  1
1  9  0  1  8
2  9  0  8  6

A    9
C    0
Name: 0, dtype: int32

     A   B    C   D
0  0.0 NaN  0.0 NaN
1  0.0 NaN  1.0 NaN
2  0.0 NaN  8.0 NaN
```

# 缺失值处理

# 缺失值处理

- NumPy中，NaN与任何数运算，结果仍为NaN

```
var = np.array([1, 2, np.nan, 4])
print(np.nan + 1)
print(np.nan * 5)
print(var.sum())
print(var.max())
```

```
nan
nan
nan
nan
```

- 忽略NaN进行计算

```
print(np.nansum(var))
print(np.nanmax(var))
print(np.nanmin(var))
```

```
7.0
4.0
1.0
```

# 缺失值处理

- 在Pandas中， NaN表示缺失值时，类型是浮点数；表示缺失对象时，类型是object类型

```
var = pd.Series([1,2],dtype=int)
print(var)
var[0] = np.nan
print(var)
```

```
0     1
1     2
dtype: int32

0     NaN
1     2.0
dtype: float64
```

```
var = pd.Series(['aa','bb'])
print(var)
var[0] = np.nan
print(var)
```

```
0     aa
1     bb
dtype: object

0     NaN
1      bb
dtype: object
```

# 缺失值处理

- isnull()与notnull()方法

```python
var = pd.Series(['aa',np.nan,1,np.nan])
print(var.isnull())
print(var.notnull())
```

```
0    False
1     True
2    False
3     True
dtype: bool

0     True
1    False
2     True
3    False
dtype: bool
```

# 缺失值处理

- 丢弃缺失值

```
var = pd.Series([’aa’,np.nan,1,np.nan])
print(var.dropna())
```

```
0     aa
2      1
dtype: object
```

# 缺失值处理

```
df = pd.DataFrame([[1,np.nan,2],
                   [2,3,5],
                   [np.nan,4,6]])
print(df)                      #
print(df.dropna())             #
print(df.dropna(axis=1))       #
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6

     0    1  2
1  2.0  3.0  5

   2
0  2
1  5
2  6
```

# 缺失值处理

```
df = pd.DataFrame([[1, np.nan, 2, 4],
                   [2, 3, 5, 3],
                   [np.nan, np.nan, np.nan, np.nan]])
print(df)
print(df.dropna(how='any'))
print(df.dropna(how='all'))
```

```
     0    1    2    3
0  1.0  NaN  2.0  4.0
1  2.0  3.0  5.0  3.0
2  NaN  NaN  NaN  NaN

     0    1    2    3
1  2.0  3.0  5.0  3.0

     0    1    2    3
0  1.0  NaN  2.0  4.0
1  2.0  3.0  5.0  3.0
```

# 缺失值处理

```
df = pd.DataFrame([[1, np.nan, 2, 4],
                   [2, np.nan, np.nan, 3],
                   [np.nan, np.nan, np.nan, np.nan]])
print(df)
print(df.dropna(thresh=3))
```

```
     0    1    2    3
0  1.0  NaN  2.0  4.0
1  2.0  NaN  NaN  3.0
2  NaN  NaN  NaN  NaN

     0    1    2    3
0  1.0  NaN  2.0  4.0
```

表示留下该行（或列）时，非缺失值的个数至少需要thresh个

# 缺失值处理

- 填充缺失值

```
df = pd.DataFrame([[11,np.nan,22,44],
                   [22,np.nan,np.nan,33],
                   [np.nan,np.nan,np.nan,np.nan]])
print(df.fillna(0))
```

```
      0    1     2     3
0  11.0  0.0  22.0  44.0
1  22.0  0.0   0.0  33.0
2   0.0  0.0   0.0   0.0
```

# 缺失值处理

```
data = pd.Series([1,np.nan,2,np.nan ('abcde'))
print(data)
print(data.fillna(method='ffill'))
print(data.fillna(method='bfill'))
```

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

a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64

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

用前后相邻的元素进行填充

# 缺失值处理

```
df = pd.DataFrame([[1,np.nan,2,4],
                   [2,3,5,3],
                   [np.nan,5,4,np.nan]])
print(df)
print(df.fillna(method='ffill', axis=1))
```

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1.0 | NaN | 2 | 4.0 |
| 1 | 2.0 | 3.0 | 5 | 3.0 |
| 2 | NaN | 5.0 | 4 | NaN |

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1.0 | 1.0 | 2.0 | 4.0 |
| 1 | 2.0 | 3.0 | 5.0 | 3.0 |
| 2 | NaN | 5.0 | 4.0 | 4.0 |

# 多级索引

# MultiIndex

考虑用Series来表示美国不同的州、不同年份的人口数据，这对Series来说就相当于有了两个索引值

```
index = [('California', 2008), ('California', 2018),
         ('New York', 2008), ('New York', 2018),
         ('Texas', 2008), ('Texas', 2018)]
mul_index = pd.MultiIndex.from_tuples(index)
print(mul_index)
```

```
MultiIndex(levels=[['California', 'New York', 'Texas'], [2008, 2018]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```python
index = [('California', 2008), ('California', 2018),
         ('New York', 2008), ('New York', 2018),
         ('Texas', 2008), ('Texas', 2018)]

mul_index = pd.MultiIndex.from_tuples(index)
population = [33870000, 37250000,
              18970000, 19370000,
              20850000, 25140000]

pop = pd.Series(population, index=mul_index)
print(pop)
```

```
California  2008    33870000
            2018    37250000
New York    2008    18970000
            2018    19370000
Texas       2008    20850000
            2018    25140000
dtype: int64
```

# MultiIndex

- 二维的Series与DataFrame转换

```
df_pop = pop.unstack()
print(df_pop)
print(df_pop.stack())
```

```
                2008      2018
California  33870000  37250000
New York    18970000  19370000
Texas       20850000  25140000

California  2008    33870000
            2018    37250000
New York    2008    18970000
            2018    19370000
Texas       2008    20850000
            2018    25140000
dtype: int64
```

# 使用DataFrame来表示三级索引

考虑表示上述三个州，2008/2018，总人口/18岁以下的人口，这里有三个维度的信息

```
index = [('California', 2008), ('California', 2018),
         ('New York', 2008), ('New York', 2018),
         ('Texas', 2008), ('Texas', 2018)]

mul_index = pd.MultiIndex.from_tuples(index)
population = [33870000, 37250000,18970000, 19370000, 20850000, 25140000]
under_18_pop = [9267089, 9284094, 4687374, 4318033, 5906301, 6879014]

pop = pd.Series(population, inc
pop_df = pd.DataFrame({'total'
                       'under1
print(pop_df)
```

```
                      total   under18
California 2008    33870000   9267089
           2018    37250000   9284094
New York   2008    18970000   4687374
           2018    19370000   4318033
Texas      2008    20850000   5906301
           2018    25140000   6879014
```

# 多级索引创建方法

- 嵌套列表

```
mul_index = pd.MultiIndex.from_arrays([['a', 'a', 'b',
'b'],[1,2,1,2]])
print(mul_index)
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

# 多级索引创建方法

- 元组列表

```
mul_index = pd.MultiIndex.from_tuples([('a', 1), ('a', 0),
('b', 1), ('b', 0)])
print(mul_index)
```

```
MultiIndex(levels=[['a', 'b'], [0, 1]],
           labels=[[0, 0, 1, 1], [1, 0, 1, 0]])
```

# 多级索引创建方法

- levels和labels标签

```
mul_index = pd.MultiIndex(levels=[['a', 'b'], [0, 1]],
              labels=[[0, 0, 1, 1], [1, 0, 1, 0]])
print(mul_index)
```

```
MultiIndex(levels=[['a', 'b'], [0, 1]],
              labels=[[0, 0, 1, 1], [1, 0, 1, 0]])
```

# 多级索引创建方法

- 相乘方法

```
mul_index = pd.MultiIndex.from_product([[2008,2018],[1,2]])
print(mul_index)
```

```
MultiIndex(levels=[[2008, 2018], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

# 行索引和列索引均可以多级

```
index = pd.MultiIndex.from_product([['May','June'],[1,2]],
                                    names=['month','visit'])
columns = pd.MultiIndex.from_product([['Tom','Bill'],[22,18]],
                                      names=['name','age'])
data = np.random.randn(4,4)
df_data = pd.DataFrame(data,index=index,columns=columns)
print(df_data)
```

```
name              Tom                 Bill
age                22        18         22        18
month visit
May   1      0.700432 -2.102260 -0.180631  0.547632
      2     -1.420270 -0.228577 -0.620010 -0.811929
June  1      0.998547 -0.454578 -0.775359 -0.239387
      2     -0.291106  0.008166  1.238868 -0.180516
```

# 多级索引取值与运算

# 多级索引取值与运算

- 与之前介绍的Series，DataFrame的索引本质一致，只是增加了索引的维度

```
California  2008    33870000
            2018    37250000
New York    2008    18970000
            2018    19370000
Texas       2008    20850000
            2018    25140000
dtype: int64
```

# 多级索引取值与运算

```
print(pop['New York',2008])
```

18970000

```
print(pop['New York'])
```

```
2008      18970000
2018      19370000
dtype: int64
```

```
print(pop[:,2008])
```

```
California    33870000
New York      18970000
Texas         20850000
dtype: int64
```

# 多级索引取值与运算

```
print(pop.loc['California':'New York'])
```

```
California  2008    33870000
            2018    37250000
New York    2008    18970000
            2018    19370000
dtype: int64
```

```
print(pop.loc[:, 2008:2018])
```

```
California  2008    33870000
            2018    37250000
New York    2008    18970000
            2018    19370000
Texas       2008    20850000
            2018    25140000
dtype: int64
```

# 多级索引取值与运算

- 四维DataFrame例子

```
name              Tom                 Bill
age                22        18         22        18
month visit
May   1      0.700432 -2.102260 -0.180631  0.547632
      2     -1.420270 -0.228577 -0.620010 -0.811929
June  1      0.998547 -0.454578 -0.775359 -0.239387
      2     -0.291106  0.008166  1.238868 -0.180516
```

```
print(df_data['Tom'])   # 访问Tom的数据
```

```
age                22        18
month visit
May   1      0.700432 -2.102260
      2     -1.420270 -0.228577
June  1      0.998547 -0.454578
      2     -0.291106  0.008166
```

```
print(df_data['Tom',22]) # Tom年龄为22岁的数据
```

```
month  visit
May    1        0.700432
       2       -1.420270
June   1        0.998547
       2       -0.291106
Name: (Tom, 22), dtype: float64
```

```
print(df_data['Tom',22]['May']) # Tom 22岁 5月份的数据
```

```
visit
1    0.700432
2   -1.420270
Name: (Tom, 22), dtype: float64
```

# 多级索引取值与运算

```
print(df_data.iloc[:2, :3])
```

```
name                  Tom                       Bill
age                    22         18              22
month visit
May   1          0.700432  -2.102260  -0.180631
      2         -1.420270  -0.228577  -0.620010
```

# DataFrame求和、求平均

```
print(df_data.mean(level='month'))
```

```
name         Tom                 Bill
age           22        18        22        18
month
May    -0.359919 -1.165418 -0.400320 -0.132148
June    0.353720 -0.223206  0.231754 -0.209951
```

```
print(df_data.sum(axis=1, level='age'))
```

```
age                 22        18
month visit
May   1       0.519801 -1.554628
      2      -2.040279 -1.040506
June  1       0.223188 -0.693965
      2       0.947762 -0.172350
```

# 数据合并

# concat

```
df1 = pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'}})
df2 = pd.DataFrame({'C':{'1':'C1','2':'C4'},'D':{'1':'D1','2':'D2'}})
print(df1)
print(df2)
print(pd.concat([df1,df2], axis=1))
```

```
    A   B
1  A1  B1
2  A2  B2

    C   D
1  C1  D1
2  C4  D2

    A   B   C   D
1  A1  B1  C1  D1
2  A2  B2  C4  D2
```

```
df1 = pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'}})
df2 = pd.DataFrame({'A':{'1':'A3','2':'A4'},'B':{'1':'B3','2':'B4'}})
print(df1)
print(df2)
print(pd.concat([df1,df2], ignore_index=True))
```

该参数可以消除重复索引

```
   A   B
1  A1  B1
2  A2  B2

   A   B
1  A3  B3
2  A4  B4

   A   B
0  A1  B1
1  A2  B2
2  A3  B3
3  A4  B4
```

```python
df1 = pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'}})
df2 = pd.DataFrame({'A':{'1':'A3','2':'A4'},'B':{'1':'B3','2':'B4'}})
print(df1)
print(df2)
print(pd.concat([df1,df2], keys=['x','y']))
```

如果需要保留原来的索引，我们可以添加一层索引进行区别

```
   A   B
1  A1  B1
2  A2  B2

   A   B
1  A3  B3
2  A4  B4

     A   B
x 1  A1  B1
  2  A2  B2
y 1  A3  B3
  2  A4  B4
```

# 列名不一致

```
df1 = 
pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'},'C':{'1':'C1','2':'C2'}})
df2 = 
pd.DataFrame({'B':{'3':'B3','4':'B4'},'C':{'3':'C3','4':'C4'},'D':{'3':'D3','4':'D4'}})
print(df1)
print(df2)
print(pd.concat([df1,df2]))
```

```
    A    B    C
1   A1   B1   C1
2   A2   B2   C2

    B    C    D
3   B3   C3   D3
4   B4   C4   D4

     A    B    C      D
1    A1   B1   C1   NaN
2    A2   B2   C2   NaN
3   NaN   B3   C3    D3
4   NaN   B4   C4    D4
```

```python
df1 =
pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'},'C':{'1':'C1','2':'C2'}})
df2 =
pd.DataFrame({'B':{'3':'B3','4':'B4'},'C':{'3':'C3','4':'C4'},'D':{'3':'D3','4':'D4'}})
print(df1)
print(df2)
print(pd.concat([df1,df2], join='inner'))
```

对输入列取交集

```
    A   B   C
1   A1  B1  C1
2   A2  B2  C2

    B   C   D
3   B3  C3  D3
4   B4  C4  D4

    B   C
1   B1  C1
2   B2  C2
3   B3  C3
4   B4  C4
```

```
df1 = 
pd.DataFrame({'A':{'1':'A1','2':'A2'},'B':{'1':'B1','2':'B2'},'C':{'1':'C1','2':'C2'}})
df2 = 
pd.DataFrame({'B':{'3':'B3','4':'B4'},'C':{'3':'C3','4':'C4'},'D':{'3':'D3','4':'D4'}})
print(df1)
print(df2)
print(pd.concat([df1, df2], join_axes=[df1.columns]))
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2

    B   C   D
3  B3  C3  D3
4  B4  C4  D4

     A   B   C
1   A1  B1  C1
2   A2  B2  C2
3  NaN  B3  C3
4  NaN  B4  C4
```

指定采用哪个合并项的列

# merge

```
df1 = pd.DataFrame({'employ                                        ue'],
                    'group'
df2 = pd.DataFrame({'employ                                        sa'],
                    'hire_d
print(df1)
print(df2)
df3 = pd.merge(df1, df2)
print(df3)
```

一对一连接

```
  employee group
0      Bob   MGR
1     Jake   R&D
2     Lisa    HR
3      Sue   R&D

  employee  hire_date
0      Bob       2004
1     Jake       2009
2      Sue       2013
3     Lisa       2010

  employee group  hire_date
0      Bob   MGR       2004
1     Jake   R&D       2009
2     Lisa    HR       2010
3      Sue   R&D       2013
```

```python
df4 = pd.DataFrame({'group': ['MGR', 'R&D', 'HR'],
                    'supervisor': ['Bill', 'Tom', 'Bob']})
print(df3)
print(df4)
df5 = pd.merge(df3, df4)
print(df5)
```

多对一连接

```
  employee group  hire_date
0      Bob   MGR       2004
1     Jake   R&D       2009
2     Lisa    HR       2010
3      Sue   R&D       2013

  group supervisor
0   MGR       Bill
1   R&D        Tom
2    HR        Bob

  employee group  hire_date supervisor
0      Bob   MGR       2004       Bill
1     Jake   R&D       2009        Tom
2      Sue   R&D       2013        Tom
3     Lisa    HR       2010        Bob
```

```
df6 = pd.DataFrame({'group':['MGR', 'R&D', 'R&D','HR','HR'],

'skill':['management','CS','math','office','english']})
df7 = pd.merge(df1, df6)
print(df1)
print(df6)
print(df7)
```

多对多连接

```
  employee group
0      Bob   MGR
1     Jake   R&D
2     Lisa    HR
3      Sue   R&D

  group       skill
0   MGR  management
1   R&D          CS
2   R&D        math
3    HR      office
4    HR     english

  employee group       skill
0      Bob   MGR  management
1     Jake   R&D          CS
2     Jake   R&D        math
3      Sue   R&D          CS
4      Sue   R&D        math
5     Lisa    HR      office
6     Lisa    HR     english
```

# 合并列名称不一致

- 两个待连接的DataFrame对

```
df1 = pd.DataFrame({'employee':                                   ],
                   'group': ['MG
df2 = pd.DataFrame({'name': ['Bob
                   'hire_date':
print(df1)
print(df2)
df3 = pd.merge(df1, df2, left_on=
right_on='name').drop('name', axi
print(df3)
```

| | employee | group |
|---|---|---|
| 0 | Bob | MGR |
| 1 | Jake | R&D |
| 2 | Lisa | HR |
| 3 | Sue | R&D |

| | name | hire_date |
|---|---|---|
| 0 | Bob | 2004 |
| 1 | Jake | 2009 |
| 2 | Sue | 2013 |
| 3 | Lisa | 2010 |

| | employee | group | hire_date |
|---|---|---|---|
| 0 | Bob | MGR | 2004 |
| 1 | Jake | R&D | 2009 |
| 2 | Lisa | HR | 2010 |
| 3 | Sue | R&D | 2013 |

# 合并索引列

```
df1 = pd.DataFrame({'employee'                    a', 'Sue'],
                    'group': [                    &D']})
df2 = pd.DataFrame({'name': ['                    Lisa'],
                    'hire_date                    2010]})

df1_a = df1.set_index('employe
df2_a = df2.set_index('name')
df3 = pd.merge(df1_a, df2_a, l                    index=True)
print(df1_a)
print(df2_a)
print(df3)
```

```
         group
employee
Bob        MGR
Jake       R&D
Lisa        HR
Sue        R&D

      hire_date
name
Bob        2004
Jake       2009
Sue        2013
Lisa       2010

     group  hire_date
Bob    MGR       2004
Jake   R&D       2009
Lisa    HR       2010
Sue    R&D       2013
```

# join

```
df1_a = df1.set_index('employee')
df2_a = df2.set_index('name')
print(df1_a.join(df2_a))
```

```
         group  hire_date
employee
Bob        MGR       2004
Jake       R&D       2009
Lisa        HR       2010
Sue        R&D       2013
```

# DataFrame对象的合并列一个是索引列，另一个是数据列

```
df1 = pd.DataFrame({'employee'                                   'Sue'],
                    'group': ['                                  })
df2 = pd.DataFrame({'name': ['B                                  '],
                    'hire_date'                                  ]})

df1_a = df1.set_index('employe
df3 = pd.merge(df1_a, df2, left                                  me')
print(df1_a)
print(df2)
print(df3)
```

```
         group
employee
Bob        MGR
Jake       R&D
Lisa        HR
Sue        R&D

   name  hire_date
0   Bob       2004
1  Jake       2009
2   Sue       2013
3  Lisa       2010

  group  name  hire_date
0   MGR   Bob       2004
1   R&D  Jake       2009
3    HR  Lisa       2010
2   R&D   Sue       2013
```

# 数据连接操作中的集合操作规则

```
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df2 = pd.DataFrame({'name': ['Mary', '
                    'drink': ['wine', '
                   columns=['name', 'dr


print(df1)
print(df2)
print(pd.merge(df1, df2, how='inner'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread

     name drink
0    Mary  wine
1  Joseph  beer

   name   food drink
0  Mary  bread  wine
```

指定连接方式：内连接

```python
print(df1)
print(df2)
print(pd.merge(df1, df2, how='left'))
```

指定连接方式：左连接

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread

     name drink
0    Mary  wine
1  Joseph  beer

    name   food drink
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  wine
```

# 多个共同列

```
df1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'rank': [1, 2, 3]})
df2 = pd.                  ', 'Mary', 'Paul'],
                                   1]})
print(df1)
print(df2)
print(pd.m                          )
```

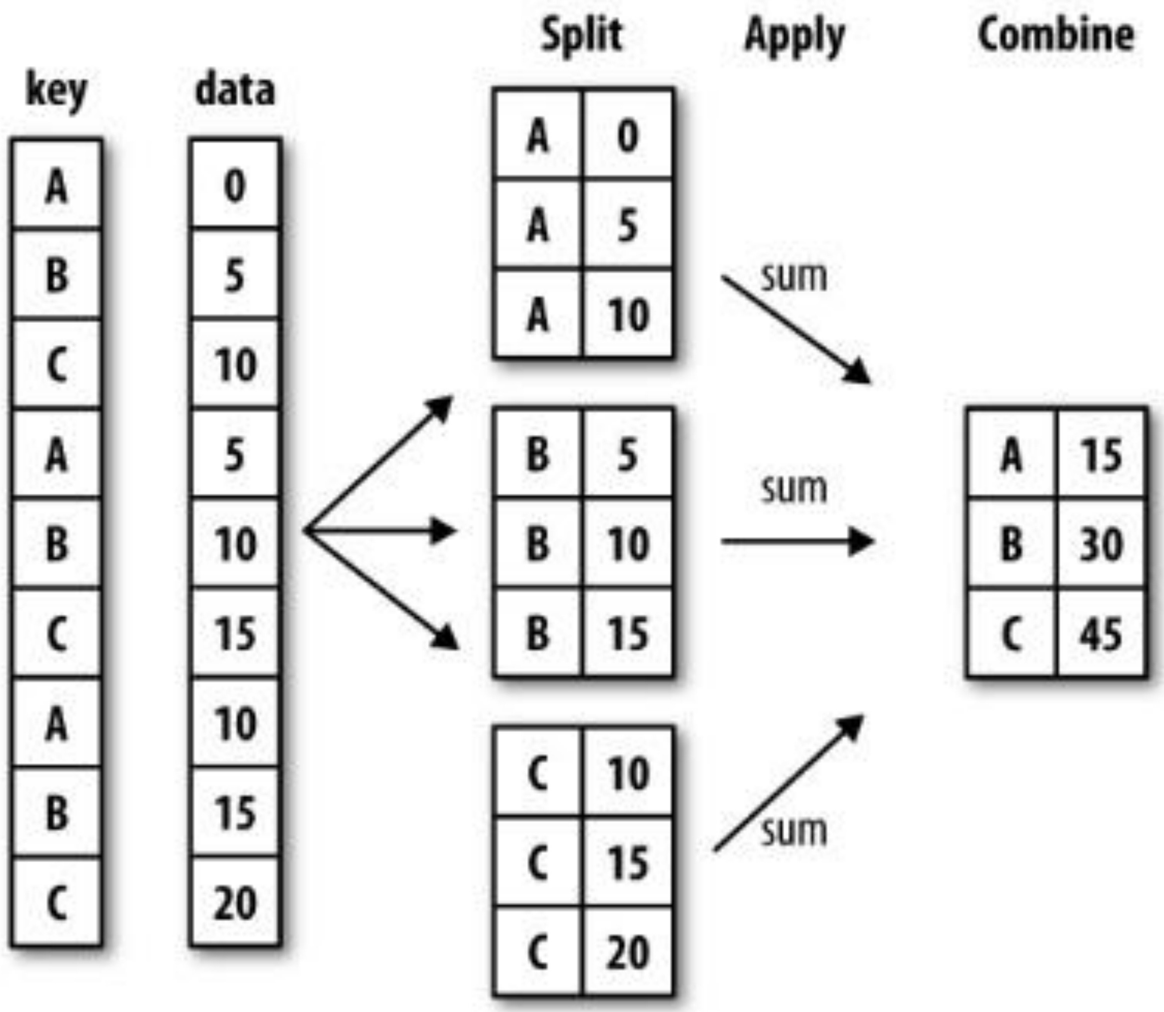
```
    name  rank
0  Peter     1
1   Paul     2
2   Mary     3

    name  rank
0  Peter     4
1   Mary     7
2   Paul     1

    name  rank_x  rank_y
0  Peter       1       4
1   Paul       2       1
2   Mary       3       7
```

可以在merge函数参数中指定suffixes参数来指定后缀

# GroupBy


Split
Apply
Combine
key
data
A
B
C
A
B
C
A
B
C
0
5
10
5
10
15
10
15
20
A 0
A 5
A 10
B 5
B 10
B 15
C 10
C 15
C 20
sum
sum
sum
A 15
B 30
C 45

# GroupBy

```
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
    'key2' : ['one', 'two', 'one', 'two', 'one'],
    'data1' : np.random.randn(5),
    'data2' : np.random.randn(5)})
print(df)
```

```
  key1 key2     data1     data2
0    a  one -1.220379 -0.614543
1    a  two -0.284783  0.901116
2    b  one  0.339224 -0.512699
3    b  two  0.532306  0.300087
4    a  one -0.929099  1.680894
```

# GroupBy

语法糖：df.groupby('key1')['data1']

```
grouped = df['data1'].groupby(df['key1'])
print(grouped)

grouped.mean()
```

```
<pandas.core.groupby.groupby.SeriesGroupBy object at 0x013587B0>

key1
a   -0.811420
b    0.435765
Name: data1, dtype: float64
```

# GroupBy

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
print(means)
means.unstack()
```

```
key1  key2
a     one     -1.074739
      two     -0.284783
b     one      0.339224
      two      0.532306
Name: data1, dtype: float64
```

```
key2        one       two
key1
a     -1.074739 -0.284783
b      0.339224  0.532306
```

# 遍历各分组

GroupBy对象支持迭代，会生成一个包含组名和数据块的2维元组序列

```
for name, group in df.groupby('key1'):
    print(name)
    print(group)
```

```
a
  key1 key2     data1     data2
0    a  one -1.220379 -0.614543
1    a  two -0.284783  0.901116
4    a  one -0.929099  1.680894
b
  key1 key2     data1     data2
2    b  one  0.339224 -0.512699
3    b  two  0.532306  0.300087
```

```python
grouped = df.groupby(df.dtypes, axis=1)
for dtype, group in grouped:
    print(dtype)
    print(group)
```

groupby默认情况下在axis=0的轴向上分组，我们也可以其他轴向上进行分组

```
float64
      data1     data2
0 -1.220379 -0.614543
1 -0.284783  0.901116
2  0.339224 -0.512699
3  0.532306  0.300087
4 -0.929099  1.680894
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

# aggregate

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key':['A','B','C','A','B','C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)})
print(df)
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

# aggregate

```
print(df.groupby('key').aggregate(['min', np.median, 'max']))
```

```
    data1              data2
      min median max     min median max
key
A       0    1.5   3       3    4.0   5
B       1    2.5   4       0    3.5   7
C       2    3.5   5       3    6.0   9
```

# aggregate

```
print(df.groupby('key').aggregate({'data1':'min','data2':'max'}))
```

```
     data1  data2
key
A        0      5
B        1      7
C        2      9
```

# filter

```
print(df.groupby('key').std())
def filter_func(x):
    return x['data2'].std() > 4

print(df.groupby('key').filter(filter_func))
```

```
        data1     data2
key
A     2.12132  1.414214
B     2.12132  4.949747
C     2.12132  4.242641

  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

# transform

```
print(df)
print(df.groupby('key').transform(lambda x: x - x.mean()))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

   data1  data2
0   -1.5    1.0
1   -1.5   -3.5
2   -1.5   -3.0
3    1.5   -1.0
4    1.5    3.5
5    1.5    3.0
```

# apply

```
def norm_by_data2(x):
    x['data1'] /= x['data2'].sum()
    return x
print(df)
print(df.groupby('key').apply(norm_by_data2))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
  key     data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

# 数据透视表

- 数据透视表根据一个或多个键聚合一张表的数据，将数据在矩形格式中排列，其中一些分组键是沿着行的，另一些是沿着列的。
- DataFrame拥有pivot_table方法，为groupby工具以及分层索引等操作提供一个便捷接口。

```python
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')

print(titanic.head())
print(titanic.columns)
```

```
   survived  pclass     sex   age  ...  deck  embark_town alive  alone
0         0       3    male  22.0  ...   NaN  Southampton    no  False
1         1       1  female  38.0  ...     C    Cherbourg   yes  False
2         1       3  female  26.0  ...   NaN  Southampton   yes   True
3         1       1  female  35.0  ...     C  Southampton   yes  False
4         0       3    male  35.0  ...   NaN  Southampton    no   True
[5 rows x 15 columns]

Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',
       'embarked', 'class', 'who', 'adult_male', 'deck', 'embark_town',
       'alive', 'alone'],
      dtype='object')
```

查看不同性别和船舱等级的生还情况

```
print(titanic.groupby(['sex','class'])['survived'].mean().unstack())
```

```
class        First      Second     Third
sex
female    0.968085   0.921053   0.500000
male      0.368852   0.157407   0.135447
```

```
print(titanic.pivot_table('survived', index='sex', columns='class'))
```

## 透视表在表达更复杂的分组时有优势

查看不同性别和船舱等级的生还情况，同时考虑年龄划分

```
age = pd.cut(titanic['age'],[0, 18, 80])
print(titanic.pivot_table('survived',['sex',age],'class'))
```

```
class             First     Second    Third
sex    age
female (0, 18]    0.909091  1.000000  0.511628
       (18, 80]   0.972973  0.900000  0.423729
male   (0, 18]    0.800000  0.600000  0.215686
       (18, 80]   0.375000  0.071429  0.133663
```

再增加一个维度：对票价分组，其分割点是最低价和最高价的平均值

```
age = pd.cut(titanic['age'],[0, 18, 80])
fare = pd.qcut(titanic['fare'],2)
print(titanic.pivot_table('survived', ['sex',age],[fare,'class']))
```

```
fare            (-0.001, 14.454]                     (14.454, 512.329]  \
class                      First    Second     Third              First
sex    age
female (0, 18]               NaN  1.000000  0.714286           0.909091
       (18, 80]              NaN  0.880000  0.444444           0.972973
male   (0, 18]               NaN  0.000000  0.260870           0.800000
       (18, 80]              0.0  0.098039  0.125000           0.391304

fare
class              Second     Third
sex    age
female (0, 18]   1.000000  0.318182
       (18, 80]  0.914286  0.391304
male   (0, 18]   0.818182  0.178571
       (18, 80]  0.030303  0.192308
```

根据性别和船舱等级分组，观测各分组里票价fare的均值和生还人数survived的总数

```
print(titanic.pivot_table(index='sex', columns='class',
                          aggfunc={'survived':'sum', 'fare':'mean'}))
```

| | fare | | | survived | | |
|---|---|---|---|---|---|---|
| **class** | First | Second | Third | First | Second | Third |
| sex | | | | | | |
| female | 106.125798 | 21.970121 | 16.118810 | 91 | 70 | 72 |
| male | 67.226127 | 19.741782 | 12.661633 | 45 | 17 | 47 |