自学报告：Pandas库

李帅 2016013270

周展平 2016013253

本文档说明:

> 本文档是针对Pandas 0.23.4的学习报告，主要是从实际的数据分析场景出发，以各个环节为线索学习Pandas的特性。其中介绍的方法只是最常用的一小部分，更多的方法在官方文档中可以找到。

一、Pandas 简介

> Pandas 是一个 Python 的开源项目，是数据分析的一个常用的工具。官方的文档中如下描述它:

> `pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

(linkage: http://pandas.pydata.org/pandas-docs/stable/overview.html)

> Pandas 数据结构的实现基于numpy，可视化方面则基于 matplotlib，因此 Pandas 对于它们均有很好的兼容性。

二、基本数据类型

*Pandas*有三种主要的数据结构：系列（`Series`），数据帧（`DataFrame`）以及面板（`Panel`），这三种数据结构分别表示1维数据，2维数据以及3维数据。除此以外，*Pandas*中还有`PanelND`数据结构以表示N维数据。这些数据结构中，较高维数据结构是较低维数据结构的容器。例如，`DataFrame`是`Series`的容器，`Panel`是`DataFrame`的容器。

系列（Series）

系列（`Series`）是一种类似于一维数组的对象，由一组数据（类型可以是各种*Numpy*中的数据类型）以及一组与之对应的数据标签（索引）组成。

`Series`可使用以下构造函数创建：

```
pandas.Series(data, index, dtype, copy)
```

参数如下：

参数	描述
`data`	系列中保存的数据，可以是任何类型，如整型、浮点型、字符串或其他Python对象类型，数据组织可采取各种形式，如`ndarray`，`list`，`dict`，`constants`等
`index`	对数据的索引，长度必须与数据相同。如果未传递该参数则数据的默认索引为`np.arange(n)`，n为`data`的长度
`dtype`	用于表明数据类型，若未传递该参数则将根据`data`推断数据类型
`copy`	是否进行数据复制，默认为`false`

根据传递的`data`的不同，`Series`的构造分为以下几种情况：

创建空系列

不传任何参数，直接调用`pandas.Series()`，可创建一个空系列，如

```
import pandas as pd
s = pd.Series()
print(s)
```

输出为

```
Series([], dtype: float64)
```

从ndarray创建一个系列

如果数据是`ndarray`，则传递的`index`必须具有相同的长度。如果未传递则默认索引为`np.arrange(n)`，n为data的长度。如

```
import pandas as pd
import numpy as np
data = np.array(['a', 'b', 'c', 'd'])
s1 = pd.Series(data)
s2 = pd.Series(data, index = [100, 101, 102, 103])
print (s1)
print (s2)
```

输出分别为

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

```
100    a
101    b
102    c
103    d
dtype: object
```

此外，pandas支持非唯一的索引值，如：

```
import pandas as pd
import numpy as np
print(pd.Series(np.random.randn(5), index = ['a', 'b', 'b', 'd', 'e']))
```

输出为：

```
a   -0.584994
b    0.858974
b   -1.472020
d   -1.142082
e   -0.415946
```

从字典创建一个系列

当数据是字典（`dict`）时，若未传递索引，则默认将以字典的键作为值的索引。若传递了索引，则将拉出与索引中的标签对应的数据的值。如：

```
import pandas as pd
d = {'b' : 1, 'a' : 0, 'c' : 2}
s1 = pd.Series(d)
s2 = pd.Series(d, index = ['b', 'c', 'd', 'a'])

print(s1)
print(s2)
```

输出分别为

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

和

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

从标量值创建一个系列

如果数据是标量值，则必须提供索引。如：

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

输出为：

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

`Series`的行为类似于`ndarray`，支持切片等操作；`Seires`的行为也类似于固定长度的`dict`，可通过键值对的方式获取数据。

数据帧（DataFrame）

数据帧（`DataFrame`）是一种类似于表格的二维数据结构，可作为`Series`的容器。

`DataFrame`可由以下构造函数创建：

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

参数如下：

参数	描述
`data`	可接收多种形式的输入，如`ndarray`，`series`，`map`，`lists`，`dict`，`constant`和另一个`DataFrame`等
`index`	对行的索引，可为省缺值`np.arange(n)`，n为行数
`columns`	对列的索引，可谓省缺值`np.arange(n)`，n为列数
`dtype`	每列的数据类型
`copy`	是否进行复制

根据传递的`data`的不同，`DataFrame`的构造分为以下几种：

创建空的数据帧

如：

```
import pandas as pd
df = pd.DataFrame()
print(df)
```

输出为：

```
Empty DataFrame
Columns: []
Index: []
```

从Series的字典创建数据帧

从`Series`的`dict`中创建`DataFrame`，如果未传递`index`参数，则默认`index`参数为各`Series`的`index`的并集。若未传递`columns`参数，则默认列为`dict`的键的有序列表，如：

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
```

输出为：

```
      one    two
a     1.0    1
b     2.0    2
c     3.0    3
d     NaN    4
```

如果传递了指定的`index`和`columns`参数，则`Series`中的`index`和`dict`中的键将会被覆盖，如：

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

输出为：

```
   two three
d    4   NaN
b    2   NaN
a    1   NaN
```

从ndarrays/lists的字典创建数据帧

ndarrays/lists必须是相同的长度，如果传递索引，则其长度需与数组长度相同。如果没有传递索引，则默认为`range(n)`，n为数组长度。如：

```
import pandas as pd
d = {'one' : [1., 2., 3., 4.], 'two' : [4., 3., 2., 1.]}
df = pd.DataFrame(d)
print(df)
```

输出为：

```
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
```

从dict的list创建数据帧

若为传递`index`参数和`columns`参数，则将以`range(n)`为`index`，`n`为`list`的长度，以`dict`的键为列。如：

```
import pandas as pd
d = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df1 = pd.DataFrame(d)
df2 = pd.DataFrame(d, index = ['first', 'second'])
df3 = pd.DataFrame(d, columns = ['a', 'b'])

print(df1)
print(df2)
print(df3)
```

输出分别为：

```
   a   b     c
0  1   2   NaN
1  5  10  20.0
```

```
        a   b     c
first   1   2   NaN
second  5  10  20.0
```

```
   a   b
0  1   2
1  5  10
```

列选择，添加，删除

`DataFrame`可看作是存储`Series`对象的`dict`，因此获取，设置，删除列的方法与`dict`相同。如：

```
import pandas as pd
d = {'one' : [1., 2., 3., 4.], 'two' : [4., 3., 2., 1.]}
df = pd.DataFrame(d)
print(df)
#输出为
   one  two
0  1.0  4.0
1  2.0  3.0
```

```
2  3.0  2.0
3  4.0  1.0

print(df['one'])
#输出为
0    1.0
1    2.0
2    3.0
3    4.0
Name: one, dtype: float64

del df['one']
print(df)
#输出为
   two
0  4.0
1  3.0
2  2.0
3  1.0
```

行选择，添加和删除

对`DataFrame`中的行可通过以下函数进行操作：

- loc()：传递行标签来选择行
- iloc()：传递整数位置来选择行
- append()：将新行添加到`DataFrame`
- drop()：通过行标签来删除行

可通过将行标签传递给`loc()`函数来选择行，也可通过将整数位置传递给`iloc()`函数来选择行。如：

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)


print(df.loc['b'])
#输出为
```

```
one 2.0
two 2.0
Name: b, dtype: float64

print(df.iloc[2])
#输出为
one   3.0
two   3.0
Name: c, dtype: float64

df2 = pd.DataFrame([[5, 6]], columns = ['one', 'two'])
df.append(df2)
print(df)
#输出为
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4
0  5.0    6

df.drop(a)
print(df)
#输出为
   one  two
b  2.0    2
c  3.0    3
d  NaN    4
```

三、文件读写

读取数据

*Pandas*提供一些用于将表格数据读取为`DataFrame`对象的函数，如：

- `read_csv`: 从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号。
- `read_table`: 从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符（“\t”）。

以上两个函数接受的参数基本相同，常见参数为：

```
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',
names=None, index_col=None, dtype=None)
```

`filepath_or_buffer`：指向文件的路径或URL或带有`read`方法的对象

`sep`：分隔符或正则表达式

`deliter`：是否将空格用作分隔符

`header`：用作列名的行号。默认为0（第一行）

`names`：用作列名的列表

`index_col`：用作行索引的列编号或列名

`dtype`：将某列的数据类型进行强制转换

示例：

将以下数据保存为`data.csv`：

```
S.No,Name,Age,City,Salary
1,Tom,28,Toronto,20000
2,Lee,32,HongKong,3000
3,Steven,43,Bay Area,8300
4,Ram,38,Hyderabad,3900
```

使用`read_csv`从csv文件中读取数据，创建`DataFrame`对象：

```
import pandas as pd
import numpy as np
df = pd.read_csv("data.csv")
print(df)
#输出为
   S.No    Name  Age       City  Salary
0     1     Tom   28    Toronto   20000
1     2     Lee   32   HongKong    3000
2     3  Steven   43   Bay Area    8300
3     4     Ram   38  Hyderabad    3900

#使用index_col来指定索引的行号
df = pd.read_csv("temp.csv",index_col=['S.No'])
print(df)
```

```
#输出为
        Name  Age       City  Salary
S.No
1        Tom   28    Toronto   20000
2        Lee   32   HongKong    3000
3     Steven   43   Bay Area    8300
4        Ram   38  Hyderabad    3900

#将Salary列的类型转换为np.float64
df = pd.read_csv("temp.csv", dtype={'Salary': np.float64})
print(df)
#输出为
  S.No   Name   Age       City    Salary
0   1     Tom   28    Toronto   20000.0
1   2     Lee   32   HongKong    3000.0
2   3  Steven   43   Bay Area    8300.0
3   4     Ram   38  Hyderabad    3900.0
print(df.dtypes)
#输出为
S.No        int64
Name       object
Age         int64
City       object
Salary    float64
dtype: object

#使用names参数来设置列名
df = pd.read_csv("temp.csv", names=['a', 'b', 'c','d','e'])
print(df)
#输出为
      a       b    c          d       e
0  S.No    Name  Age       City  Salary
1     1     Tom   28    Toronto   20000
2     2     Lee   32   HongKong    3000
3     3  Steven   43   Bay Area    8300
4     4     Ram   38  Hyderabad    3900

#使用header参数来指定某行作为列名，若此时设置了names参数，则header指定的行将被删除
df = pd.read_csv("temp.csv",names=['a','b','c','d','e'],header=0)
print(df)
#输出为
   a       b   c          d      e
```

```
0  1     Tom  28    Toronto  20000
1  2     Lee  32   HongKong   3000
2  3  Steven  43   Bay Area   8300
3  4     Ram  38  Hyderabad   3900
```

写出数据

*Pandas*可通过`to_csv`和`to_json`函数来将数据分别输出为csv文件和json文件。

`to_csv`的常用参数为：

- `path_or_buf`：要写入的文件或StringIO的字符串路径
- `sep`：输出文件的字段分隔符（默认为“，”）
- `cols`：要写入的列（默认为None）
- `header`：是否写出列名（默认为True）
- `index`：是否写出行（索引）名称（默认为True）

`to_json`的常用参数为：

- `path_or_buf`：用于写入输出的路径名或缓冲区。如果需要返回json字符串的话该参数可为None
- `orient`：json字符串的格式，常用输入为：
  - split：输出格式为{index->[index],columns-> [columns], data->[values]}的`dict`
  - records：输出格式为[{column -> value}, … , {column -> value}]的`list`
  - index：输出格式为{index -> {column -> value}}的`dict`
  - columns：输出格式为 {column -> {index -> value}}的`dict`
  - values：输出序列化数组

示例：

```
import pandas as pd
dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7,
10)),columns=list('ABC'), index=list('xyz'))
print(dfjo)
#输出为
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

print(dfjo.to_json(orient="columns"))
#输出为
```

```
'{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

print(dfjo.to_json(orient="index"))
#输出为
'{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

print(dfjo.to_json(orient="records"))
#输出为
'[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

print(dfjo.to_json(orient="split"))
#输出为
'{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],
[3,6,9]]}'

print(dfjo.to_json(orient="values"))
#输出为
'[[1,4,7],[2,5,8],[3,6,9]]'
```

四、数据索引(index)、排序(sort)

索引

*pandas*中的数据对象支持*Python*和*NumPy*中的`[]`和`.`索引访问符，此处不再赘述。这里只要介绍*Pandas*的多轴索引：

- `loc`：基于标签进行索引
- `iloc`：基于整数进行索引

loc

`loc`支持多种访问方式，如

- 单个标签，如5或'a'
- 列表或标签数组，如['a', 'b', 'c']
- 带标签的切片对象，如'a':'f'
- 布尔数组
- 可调用的函数

`loc`需要两个单/列表/范围运算符，用`","`分隔。第一个表示行，第二个表示列。

示例：

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])
print(df)
#输出为
          A         B         C         D
a -1.592179  0.076265 -0.923470 -0.794027
b -0.851854  0.443821  0.843279  0.144215
c -0.448750 -0.198909 -0.654940 -1.318762
d -2.439036  0.321245 -0.199227  0.050368
e  1.600584 -0.698589 -1.572020 -1.114553
f -0.260017  0.421532  0.826567 -0.372232
g  0.100750 -1.172052  1.616512  1.090774
h -0.370828  0.166327 -1.121000 -0.631396
print(df.loc[:, 'A'])#输出'A'列的所有行
#输出为
a   -1.592179
b   -0.851854
c   -0.448750
d   -2.439036
e    1.600584
f   -0.260017
g    0.100750
h   -0.370828

print(df.loc[:, ['A', 'C']])#输出'A', 'C'列的所有行
#输出为
          A         C
a -1.592179 -0.923470
b -0.851854  0.843279
c -0.448750 -0.654940
d -2.439036 -0.199227
e  1.600584 -1.572020
f -0.260017  0.826567
g  0.100750  1.616512
h -0.370828 -1.121000

print (df.loc[['a','b','f','h'],['A','C']])
#输出为
```

```
          A         C
a -1.592179 -0.923470
b -0.851854  0.843279
f -0.260017  0.826567
h -0.370828 -1.121000

print(df.loc['a':'c'])#切片输出
#输出为
          A         B         C         D
a -1.592179  0.076265 -0.923470 -0.794027
b -0.851854  0.443821  0.843279  0.144215
c -0.448750 -0.198909 -0.654940 -1.318762

print(df.loc['a']>0)#使用布尔数组来过滤数据
#输出为
A    False
B     True
C    False
D    False

print(df.A.loc[lambda s: s > 0])#使用匿名函数作为数据选择器
#输出为
e    1.600584
g    0.100750
```

iloc

`iloc`与`loc`类似，只是将标签改为序列中对应的整数，其余操作基本相同，此处不再赘述。

此外，*Pandas*可通过`sample`函数来返回随机样本。如：

```
#使用上文数据
print(df.sample())
#输出为
          A         B        C         D
c -0.44875 -0.198909 -0.65494 -1.318762
```

排序

*pandas*常用的排序函数有：

`sort_index`：将`DataFrame`中的数据按标签排序

常用参数：

`axis`：参数值为0或1，默认情况下为0，表示对行标签进行排列；传入1表示对列标签进行排序

`ascending`：传入布尔值，默认情况下为True，表示升序排序；传入False则表示降序排序

`kind`：用来排序的算法，可从{'quicksort', 'mergesort', 'heapsort'}中选择，默认为'quicksort'

`sort_values`：将`DataFrame`中的数据按值排序

常用参数：

`by`：指定对某一列或某一行进行排序

`axis`：同上`sort_index`中的参数

`ascending`：同上`sort_index`中的参数

`kind`：同上`sort_index`中的参数

示例：

```
import pandas as pd
import numpy as np

unsorted_df = pd.DataFrame(np.random.randn(10,2),index=
[1,4,6,2,3,5,9,8,0,7],columns = ['col2','col1'])
print(unsorted_df)
#输出为
       col2      col1
1 -0.795173  1.379910
4  1.314992 -0.086624
6 -0.561880 -1.932147
2 -1.160174 -0.654222
3  0.519920  0.306030
```

```
5 -2.124039  0.074746
9 -1.028733 -0.023074
8  1.818023  1.962943
0  2.198657  0.740556
7 -1.181086  0.861429

print(unsorted_df.sort_index())#默认对行升序排序
#输出为
       col2      col1
0  2.198657  0.740556
1 -0.795173  1.379910
2 -1.160174 -0.654222
3  0.519920  0.306030
4  1.314992 -0.086624
5 -2.124039  0.074746
6 -0.561880 -1.932147
7 -1.181086  0.861429
8  1.818023  1.962943
9 -1.028733 -0.023074

print(unsorted_df.sort_index(ascending=False))#对行进行降序排序
#输出为
       col2      col1
9 -1.028733 -0.023074
8  1.818023  1.962943
7 -1.181086  0.861429
6 -0.561880 -1.932147
5 -2.124039  0.074746
4  1.314992 -0.086624
3  0.519920  0.306030
2 -1.160174 -0.654222
1 -0.795173  1.379910
0  2.198657  0.740556

print(unsorted_df.sort_index(axis=1))#对列进行排序
#输出为
       col1      col2
1  1.379910 -0.795173
4 -0.086624  1.314992
6 -1.932147 -0.561880
2 -0.654222 -1.160174
3  0.306030  0.519920
```

```
5  0.074746 -2.124039
9 -0.023074 -1.028733
8  1.962943  1.818023
0  0.740556  2.198657
7  0.861429 -1.181086

print(unsorted_df.sort_values(by='col1'))#对col1的数据进行排序
#输出为
       col2      col1
6 -0.561880 -1.932147
2 -1.160174 -0.654222
4  1.314992 -0.086624
9 -1.028733 -0.023074
5 -2.124039  0.074746
3  0.519920  0.306030
0  2.198657  0.740556
7 -1.181086  0.861429
1 -0.795173  1.379910
8  1.818023  1.962943
```

五、数据分组(groupby)

数据分组的含义包含3个方面：

1. **Split**：将数据按照一定的标准进行分类。
2. **Apply**：对于每一个类别的数据，进行特定的操作。其中包括：
   （1）**Aggregation**：计算类别内数据的总体特征，如和、均值、维数
   （2）**Transformation**：对类内数据总体进行处理，如标准化、填补NA
   （3）**Filtration**：对某些类别的数据进行丢弃、筛选等
3. **Combine**：将经过操作的所有类别的数据重新按照某种方式组合起来 。这一部分内容将在
   下一节介绍。

下面简单介绍具体的方法：

1. Split

常用方法：

`grouped = obj.groupby()` 方法：接受的参数包括：

- 1个python函数，对指定方向(axis)的标签(label)进行处理，用来进行分类
- 1个列表或numpy数组对象，对应于指定方向(axis)的标签(label)
- 1个字典或Series对象，用于制定标签(label)到类名(group name)的映射关系
- 如果调用者是1个DataFrame对象，字符串指定了分类用到的列(colomn)

`grouped.get_group(key)` 方法：从所有分组中获得指定key的组。

`grouped.filter()` 方法：筛选分组

`grouped.count()` 方法：输出每个分组中的元素个数

常用属性：

`grouped.groups` 属性：获得所有分组

迭代：`for name, group in grouped:`

实例：

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                   'C' : np.random.randn(8),
                   'D' : np.random.randn(8),
                   'B' : ['one', 'one', 'two', 'three',
                         'two', 'two', 'one', 'three']
                   })

# 可以指定colomn
grouped = df.groupby('A')
grouped = df.groupby(['A', 'B'])
print(grouped.groups)
# 输出结果：
{('bar', 'one'): Int64Index([1], dtype='int64'), ('bar', 'three'):
Int64Index([3], dtype='int64'), ('bar', 'two'): Int64Index([5],
dtype='int64'), ('foo', 'one'): Int64Index([0, 6], dtype='int64'), ('foo',
'three'): Int64Index([7], dtype='int64'), ('foo', 'two'): Int64Index([2, 4],
dtype='int64')}

# 输出每个分组中的元素个数
```

```
print(grouped.count())
# 输出结果
           C  D
A   B
bar one    1  1
    three  1  1
    two    1  1
foo one    2  2
    three  1  1
    two    2  2

# 可以指定方法by以及方向axis
# 下面是按照colomn名是否为元音字母来进行分组
def get_letter_type(letter):
    if letter.lower() in 'aeiou':
       return 'vowel'
    else:
       return 'consonant'

grouped = df.groupby(get_letter_type, axis=1)
for group in grouped:
    print(group)
# 输出结果:
('consonant',         B         C         D
0    one  0.350096  1.082806
1    one  0.324642  0.611490
2    two  0.075571  0.654428
3  three  0.028201  0.024015
4    two  1.286625  1.217880
5    two  0.975612 -0.914238
6    one  0.282143  1.009814
7  three  1.181507  0.257534)
('vowel',      A
0  foo
1  bar
2  foo
3  bar
4  foo
5  bar
6  foo
7  foo)
```

```
# 取消默认的将关键字排序
grouped = df.groupby(get_letter_type, axis=1,sort=False)
print(grouped.get_group('consonant'))
# 输出结果：此时第二组的输出顺序为C, D, B
          C         D      B
0 -0.612971  0.758547    one
1 -0.601868  0.605106    one
2 -0.307514 -0.638541    two
3 -0.100397 -0.728291  three
4  0.399796  1.092549    two
5  0.555295  0.849624    two
6  0.139331  1.831764    one
7 -0.194881  0.244773  three
```

2. Apply

aggregation:

常用方法:

`grouped.aggregate(method)` 方法：使用method参数处理grouped中的每一个分组。

`grouped.agg([method1[,method2[,...]]])` 方法：一次性用多个方法进行处理。

`grouped.agg({colomn:method,...})` 方法：用不同方法处理不同列的数据。

实例:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                   'C' : np.random.randn(8),
                   'D' : np.random.randn(8),
                   'B' : ['one', 'one', 'two', 'three',
                          'two', 'two', 'one', 'three']
                   })

grouped = df.groupby('A',as_index=False)

# 使用np.sum函数对每组求和
print(grouped.aggregate(np.sum))
```

```
# 输出结果:
     A         C         D
0  bar -1.000992 -0.081658
1  foo -1.340381  3.082666
# 效果与sum()函数相同:
print(grouped.sum())
# 输出结果:
     A         C         D
0  bar -1.000992 -0.081658
1  foo -1.340381  3.082666

# 多个方法处理
print(grouped['C'].agg([np.sum, np.mean, np.std]))
# 输出结果:
          sum      mean       std
A
bar -0.467876 -0.155959  0.300311
foo  2.787968  0.557594  0.748056

# 用不同方法处理不同列的数据
print(grouped.agg({'C' : 'sum', 'D' : 'std'}))
# 输出结果:
     A         C         D
0  bar  0.613597  0.415764
1  foo -2.231638  0.773108
```

常用方法:

`grouped.describe()` 方法：计算一系列的统计量，这些统计量也有专门的函数可以单独调用。

实例:

```
# 计算统计量
print(grouped.describe())
# 输出结果:
      C
    D
  count      mean       std       min       25%       50%       75%       max
count      mean       std       min       25%       50%       75%       max
0   1.0  0.254161       NaN  0.254161  0.254161  0.254161  0.254161  0.254161
  1.0  1.511763       NaN  1.511763  1.511763  1.511763  1.511763  1.511763
1   1.0  0.215897       NaN  0.215897  0.215897  0.215897  0.215897  0.215897
  1.0 -0.990582       NaN -0.990582 -0.990582 -0.990582 -0.990582 -0.990582
2   1.0 -0.077118       NaN -0.077118 -0.077118 -0.077118 -0.077118 -0.077118
  1.0  1.211526       NaN  1.211526  1.211526  1.211526  1.211526  1.211526
3   2.0 -0.491888  0.117887 -0.575247 -0.533567 -0.491888 -0.450209 -0.408530
  2.0  0.807291  0.761937  0.268520  0.537905  0.807291  1.076676  1.346061
4   1.0 -0.862495       NaN -0.862495 -0.862495 -0.862495 -0.862495 -0.862495
  1.0  0.024580       NaN  0.024580  0.024580  0.024580  0.024580  0.024580
5   2.0  0.024925  1.652692 -1.143704 -0.559389  0.024925  0.609240  1.193555
  2.0  0.592714  1.462816 -0.441652  0.075531  0.592714  1.109898  1.627081
```

transformation

常用方法:

`grouped.transform(method)` 方法：使用method方法对每组中的数据进行处理

实例:

```
index = pd.date_range('10/1/1999', periods=1100)
ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
ts = ts.rolling(window=100,min_periods=100).mean().dropna()

key = lambda x: x.year
zscore = lambda x: (x - x.mean()) / x.std() #用于transform的函数，进行归一化
transformed = ts.groupby(key).transform(zscore)

grouped_trans = transformed.groupby(key)
print(grouped_trans.mean()) #期望
print(grouped_trans.std()) #标准差
# 输出结果
2000   -2.699790e-16
2001    1.861525e-16
```

```
2002   -6.561138e-16
dtype: float64
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

filtration:

常用方法:

`grouped.filter(method)` 方法：使用method方法对每组中的数据进行判断，返回True或False，从而筛选出返回True的数据

实例:

```
sf = pd.Series([1, 1, 1, 1, 2, 3, 4])
print(sf.groupby(sf).filter(lambda x: x.sum() > 3)) # 筛选出数据的和大于3的分组
# 输出结果
0    1
1    1
2    1
3    1
6    4
dtype: int64
```

apply:

常用方法:

`grouped.apply(method)` 方法：对每一个group应用apply方法

实例:

```
sf = pd.Series(np.random.normal(0.5, 2, 10), index=np.arange(10))
def f(x):
   return pd.Series([ x, x**2 ], index = ['x', 'x^2'])
print(sf.apply(f))
# 输出结果
          x        x^2
0  1.465540   2.147806
1  1.018819   1.037991
2  0.927661   0.860555
```

```
3  3.959855  15.680452
4  0.750213   0.562820
5  0.784470   0.615393
6  3.476535  12.086294
7  0.099393   0.009879
8 -1.783570   3.181121
9  0.427524   0.182777
```

（本节参考资料：http://pandas.pydata.org/pandas-docs/stable/groupby.html）

六、合并(concatenate,merge,join)

1. concatenate:

常用方法:

`pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,keys=None, levels=None, names=None, verify_integrity=False,copy=True)` 方法：

- `objs`：需要合并的pandas对象的序列或者字典，其中字典的键值会被当做keys参数
- `axis`：合并的方向
- `join`：如何处理其他方向的index的合并，outer代表取并集，inner代表取交集
- `ignore_index`：是否忽略以前的index
- `keys`：作为新的index添加到合并后的元素中，形成多级的索引结构
- `levels`：一个列表序列，用来建立 multiIndex
- `names`：为新产生的index指定名字
- `verify_integrity`：查看合并方向上是否有重复
- `copy`：是否复制元素（深拷贝）

实例:

```
import numpy as np
import pandas as pd

df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])
```

```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'E': ['E4', 'E5', 'E6', 'E7'],
                    'F': ['F4', 'F5', 'F6', 'F7']},
                     index=[4, 5, 6, 7])


df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])

# concantenate展示1: join='inner',keys=['x', 'y', 'z'],names=['level1']
concatenated1 = pd.concat([df1,df2,df3],join='inner',keys=['x', 'y',
'z'],names=['level1'])
print(concatenated1)
# 输出结果:
             A    B
level1
x      0    A0   B0
       1    A1   B1
       2    A2   B2
       3    A3   B3
y      4    A4   B4
       5    A5   B5
       6    A6   B6
       7    A7   B7
z      8    A8   B8
       9    A9   B9
       10  A10  B10
       11  A11  B11

# concantenate展示2: join='outer'
concatenated2 = pd.concat([df1,df2,df3],join='outer')
print(concatenated2)
# 输出结果:
      A    B    C    D    E    F
0    A0   B0   C0   D0  NaN  NaN
1    A1   B1   C1   D1  NaN  NaN
2    A2   B2   C2   D2  NaN  NaN
```

```
3    A3   B3   C3   D3  NaN  NaN
4    A4   B4  NaN  NaN   E4   F4
5    A5   B5  NaN  NaN   E5   F5
6    A6   B6  NaN  NaN   E6   F6
7    A7   B7  NaN  NaN   E7   F7
8    A8   B8   C8   D8  NaN  NaN
9    A9   B9   C9   D9  NaN  NaN
10  A10  B10  C10  D10  NaN  NaN
11  A11  B11  C11  D11  NaN  NaN

# concantenate展示3: axis=1,join='outer',ignore_index=True
concatenated3 =
pd.concat([df1,df2,df3],axis=1,join='outer',ignore_index=True)
print(concatenated3)
# 输出结果:
     0    1    2    3    4    5    6    7    8    9    10   11
0    A0   B0   C0   D0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
1    A1   B1   C1   D1  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
2    A2   B2   C2   D2  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
3    A3   B3   C3   D3  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
4   NaN  NaN  NaN  NaN   A4   B4   E4   F4  NaN  NaN  NaN  NaN
5   NaN  NaN  NaN  NaN   A5   B5   E5   F5  NaN  NaN  NaN  NaN
6   NaN  NaN  NaN  NaN   A6   B6   E6   F6  NaN  NaN  NaN  NaN
7   NaN  NaN  NaN  NaN   A7   B7   E7   F7  NaN  NaN  NaN  NaN
8   NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN   A8   B8   C8   D8
9   NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN   A9   B9   C9   D9
10  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  A10  B10  C10  D10
11  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  A11  B11  C11  D11
```

2. merge:

此方法是针对两个 `DataFrame` 对象的、类似于数据库方法的高效合并的方法。

常用方法:

`pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=True, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)` 方法:

- `left` 、 `right`: 需要合并的 `DataFrame` 对象
- `how` : 当左右的键值集合不相同时如何解决，可以取的值为 `'left'`, `'right'`, `'outer'`, `'inner'`

- `on`：选择按照哪些键值进行merge操作
- `left_on` ： 从 `left` 对象中选择的作为键值的column或index
- `right_on` ： 从 `right` 对象中选择的作为键值的column或index
- `sort`：是否将keys按照自字典序排序
- `suffixes`：当column名相同时用以加以区分的后缀
- `copy`：是否复制元素（深拷贝）
- `indicator`：用来指示每一行的数据的来源，在结果中额外增加`_merge_` 列，值可以为 `'left_only'`, `'right_only'`, `'both'`
- `validate` ：验证合并的两元素的关系，取值为 '1:1' ,'1:m','m:1','m:m'

实例:

```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                      'key2': ['K0', 'K0', 'K0', 'K0'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})

# merge展示1: on=['key1', 'key2']
merged1 = pd.merge(left, right, on=['key1', 'key2'])
print(merged1)
# 输出结果:
  key1 key2   A   B   C   D
0   K0   K0  A0  B0  C0  D0
1   K1   K0  A2  B2  C1  D1
2   K1   K0  A2  B2  C2  D2

# merge展示2: how='left', on=['key1', 'key2'], indicator=True, 键值组合集合以
left为准
merged2 = pd.merge(left, right, how='left', on=['key1', 'key2'],
indicator=True)
print(merged2)
# 输出结果:
  key1 key2   A   B    C    D      _merge
0   K0   K0  A0  B0   C0   D0        both
1   K0   K1  A1  B1  NaN  NaN   left_only
2   K1   K0  A2  B2   C1   D1        both
```

```
3   K1   K0  A2  B2   C2   D2       both
4   K2   K1  A3  B3  NaN  NaN  left_only
# merge展示3: how='right', on=['key1', 'key2'], indicator=True, 键值组合集合以right为准
merged3 = pd.merge(left, right, how='right', on=['key1', 'key2'], indicator=True)
print(merged3)
# 输出结果:
  key1 key2    A    B   C   D      _merge
0   K0   K0   A0   B0  C0  D0        both
1   K1   K0   A2   B2  C1  D1        both
2   K1   K0   A2   B2  C2  D2        both
3   K2   K0  NaN  NaN  C3  D3  right_only
# merge展示4: how='outer', on=['key1', 'key2'], indicator=True, 键值组合集合求并集
merged4 = pd.merge(left, right, how='outer', on=['key1', 'key2'], indicator=True)
print(merged4)
# 输出结果:
  key1 key2    A    B    C    D      _merge
0   K0   K0   A0   B0   C0   D0        both
1   K0   K1   A1   B1  NaN  NaN   left_only
2   K1   K0   A2   B2   C1   D1        both
3   K1   K0   A2   B2   C2   D2        both
4   K2   K1   A3   B3  NaN  NaN   left_only
5   K2   K0  NaN  NaN   C3   D3  right_only
# merge展示5: how='inner', on=['key1', 'key2'], indicator=True, 键值组合集合求交集
merged5 = pd.merge(left, right, how='inner', on=['key1', 'key2'], indicator=True)
print(merged5)
# 输出结果:
  key1 key2   A   B   C   D _merge
0   K0   K0  A0  B0  C0  D0   both
1   K1   K0  A2  B2  C1  D1   both
2   K1   K0  A2  B2  C2  D2   both
```

3. join:

此方法是针对两个 `DataFrame` 对象，与 `merge` 方法类似。

常用方法:

`DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)` 方法：

- `other`：`DataFrame` 对象、带有 `name` 属性的`Series` 对象或者`DataFrame` 对象的列表
- `on`：选择进行合并时选用的列
- `how`：可以取的值为`'left'`, `'right'`, `'outer'`, `'inner'`，与merge方法的含义类似
- `lsuffix`、`rsuffix`：当column名相同时用以加以区分的后缀
- `sort`：是否对键值对按照字典序进行排序

事实上，以下两个函数是等价的：

```
left.join(right, on=key_or_keys)
pd.merge(left, right, left_on=key_or_keys, right_index=True,
      how='left', sort=False)
```

因此，理解了 `merge` 函数也就理解了 `join` 函数，此处不再举例。

（本节参考资料：http://pandas.pydata.org/pandas-docs/stable/merging.html）

七、可视化

在这里，我们针对数据集 `MovieLens` (https://grouplens.org/datasets/movielens/) 进行可视化分析。

1. 基本绘图：

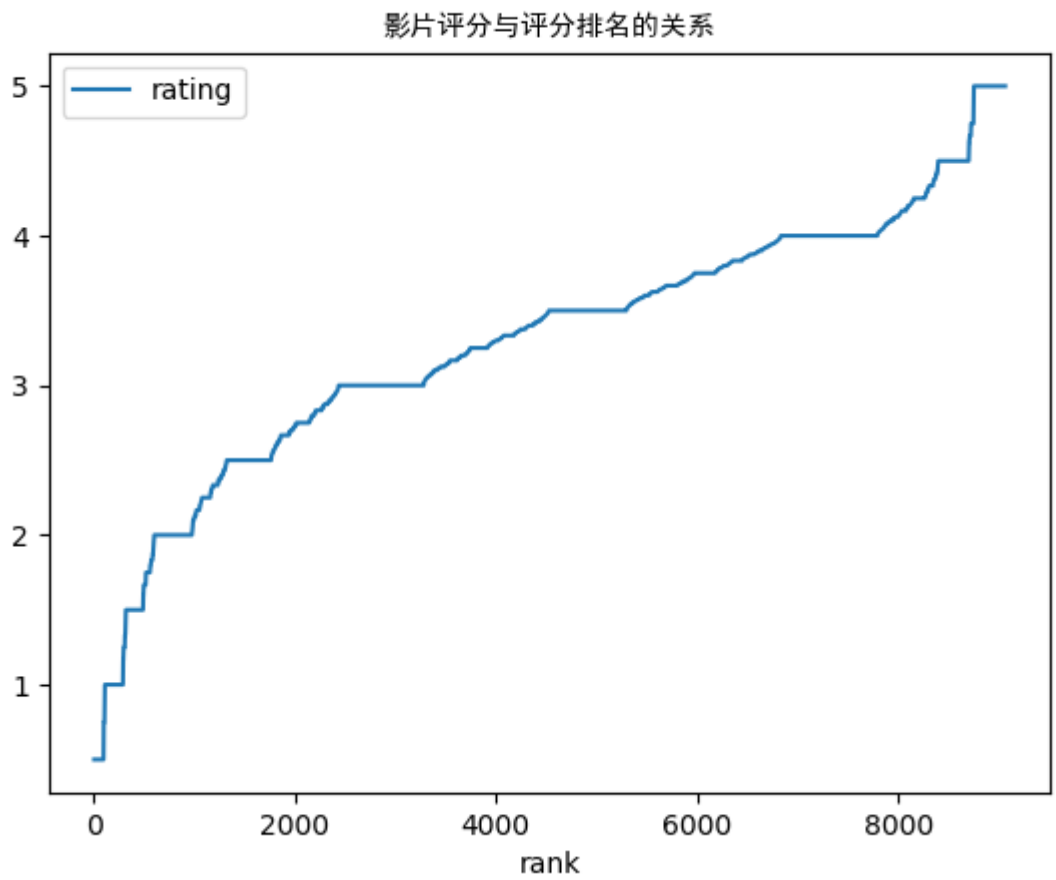
方法： `series_obj.plot()` 或 `dataframe_obj.plot()`，

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# 读入数据
path = '../dataset/ml-latest-small/'
links = pd.read_csv(path+'links.csv', encoding='latin-1')
movies = pd.read_csv(path+'movies.csv', encoding='latin-1')
ratings = pd.read_csv(path+'ratings.csv', encoding='latin-1')
tags = pd.read_csv(path+'tags.csv', encoding='latin-1')
```

```
# 综合各表数据
movie_ratings = pd.merge(movies, ratings)
# 计算每部影片的评分的平均值
ratings_mean = ratings.groupby(by='movieId').mean()
# 按照平均分排序
ratings_mean_sorted = ratings_mean.sort_values(by='rating')
ratings_mean_sorted['rank'] =
ratings_mean_sorted['rating'].rank(ascending=1,method='first')
# 按照评分从低到高绘图
ratings_mean_sorted.plot(x='rank',y='rating')
plt.title(u"影片评分与评分排名的关系", fontproperties="SimHei")
plt.show()
```

输出如图:



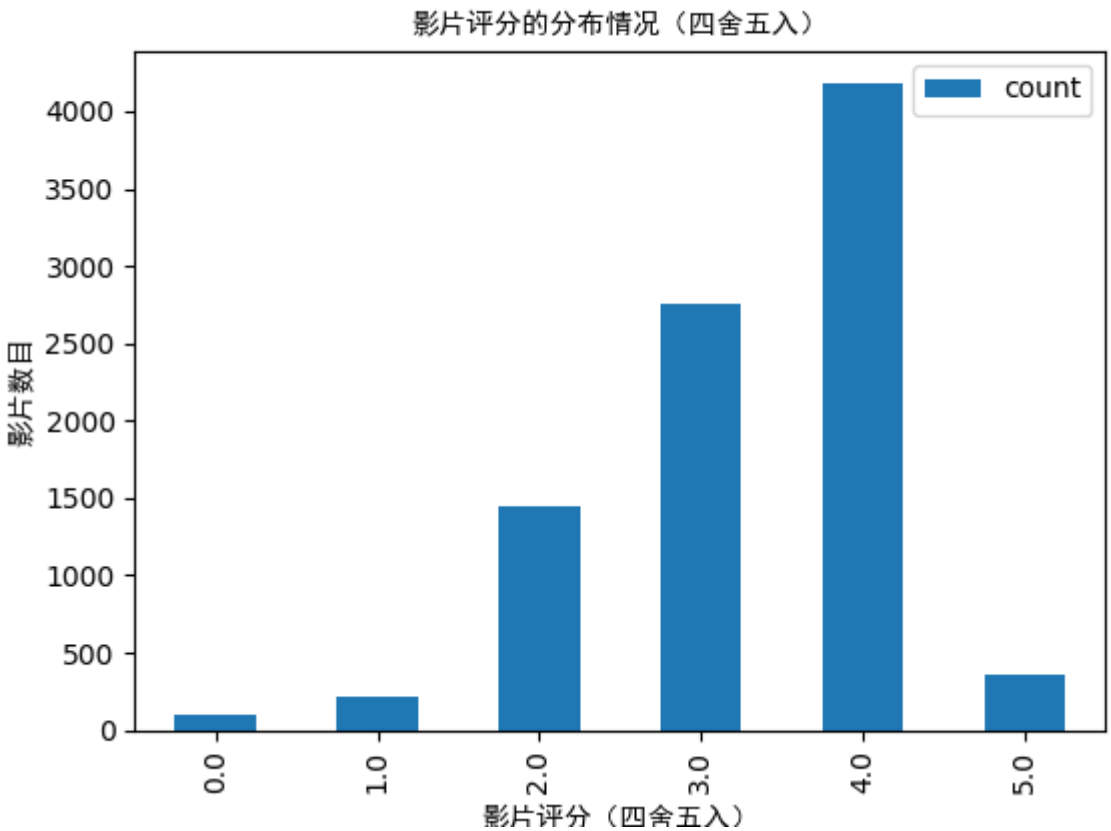
2. 条形图

方法: `series_obj.plot.bar()` 或 `dataframe_obj.plot.bar()` ,

```
# 四舍五入
```

```
ratings_mean['rating'] = ratings_mean['rating'].apply(np.round)
# 按照四舍五入后的评分进行分组
grouped = ratings_mean.groupby('rating')
count = []
names = []
for name, group in grouped:
    names.append(name)
    count.append(len(group))
rel = pd.DataFrame({'count':count},index=names)
# 条形图
rel.plot.bar()
plt.title(u"影片评分的分布情况（四舍五入）", fontproperties="SimHei")
plt.xlabel(u"影片评分（四舍五入）", fontproperties="SimHei")
plt.ylabel(u"影片数目", fontproperties="SimHei")
plt.show()
```

输出如图：

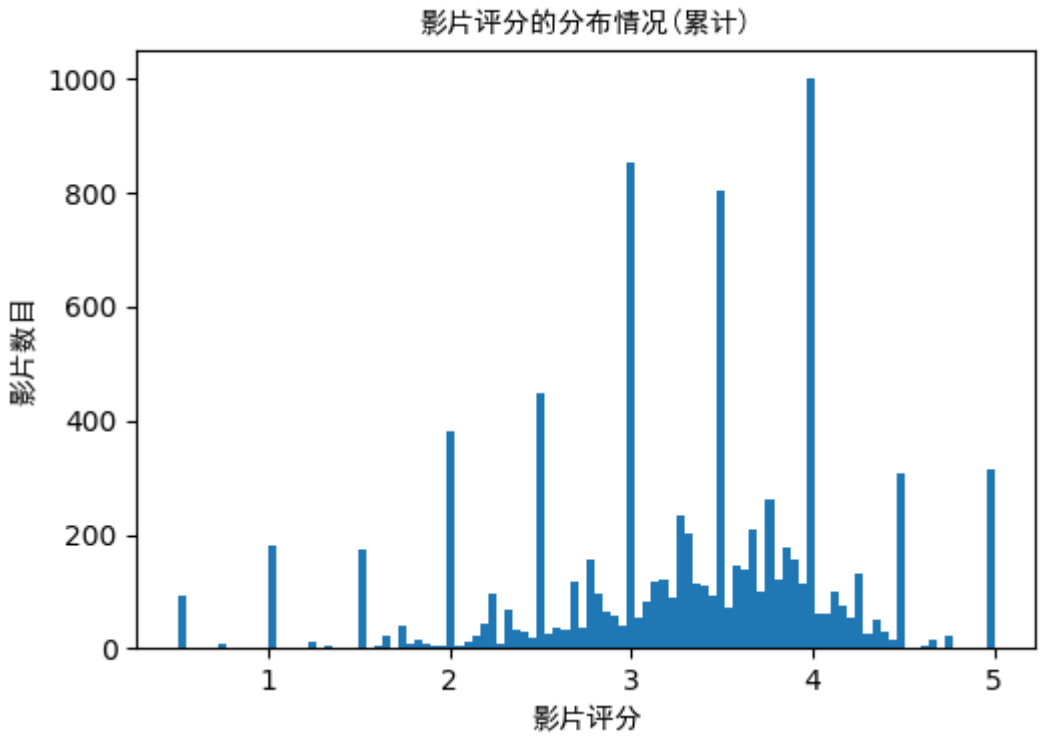


3. 直方图

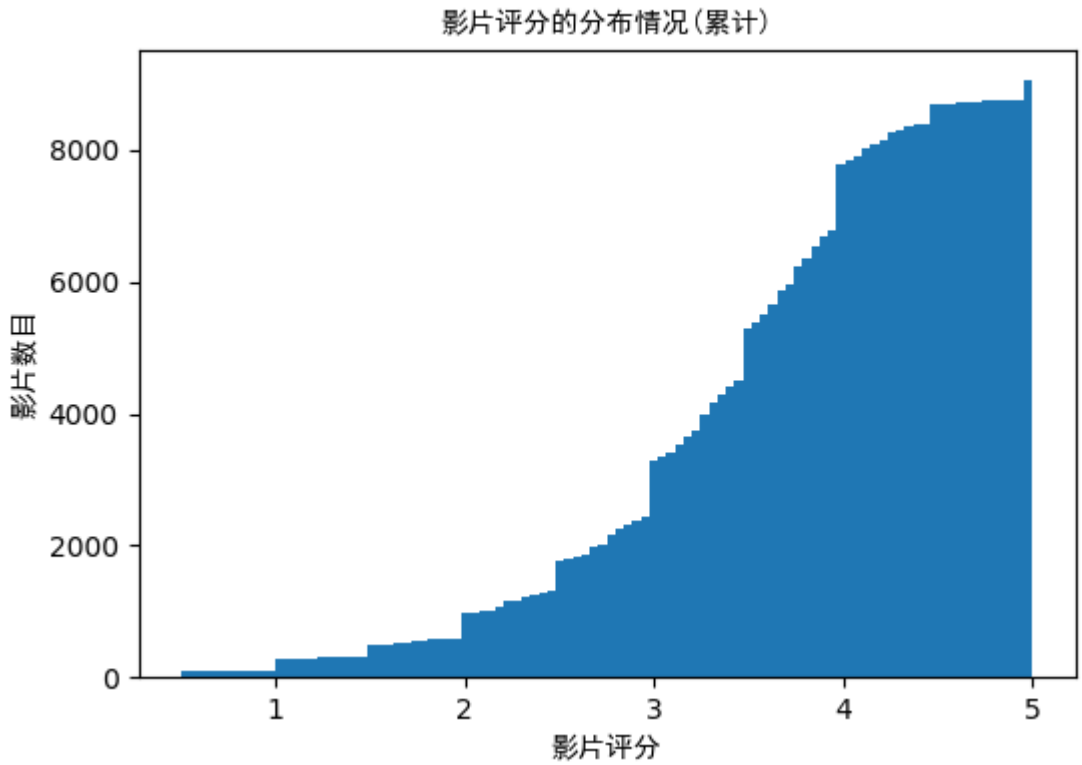
方法： `series_obj.plot.hist([alpha=])` 或 `dataframe_obj.plot.hist([alpha=])` ，

```
# 直方图
ratings_mean['rating'].plot.hist(bins=100)
plt.title(u"影片评分的分布情况", fontproperties="SimHei")
plt.xlabel(u"影片评分", fontproperties="SimHei")
plt.ylabel(u"影片数目", fontproperties="SimHei")
plt.show()
```

输出如图：



设置 cumulative = True 后，得到累计图：

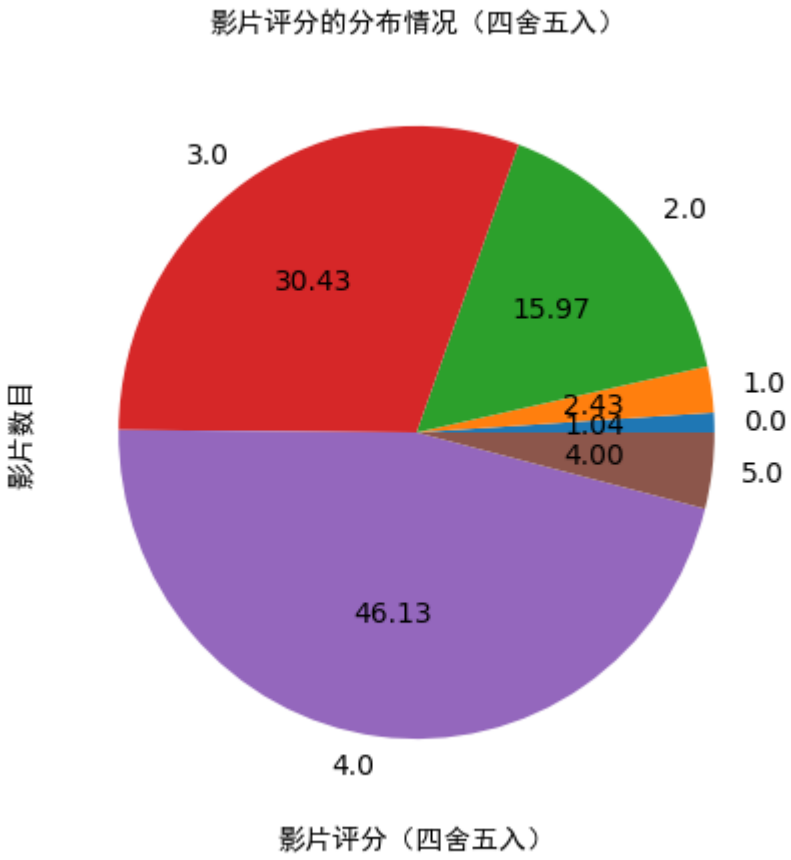


4. 扇形图

方法： `series_obj.plot.pie(figsize=(,))`或`dataframe_obj.plot.pie(figsize=(,))`，

```
# 扇形图
rel['count'].plot.pie(figsize=(6,6),autopct='%.2f')
```

输出结果：

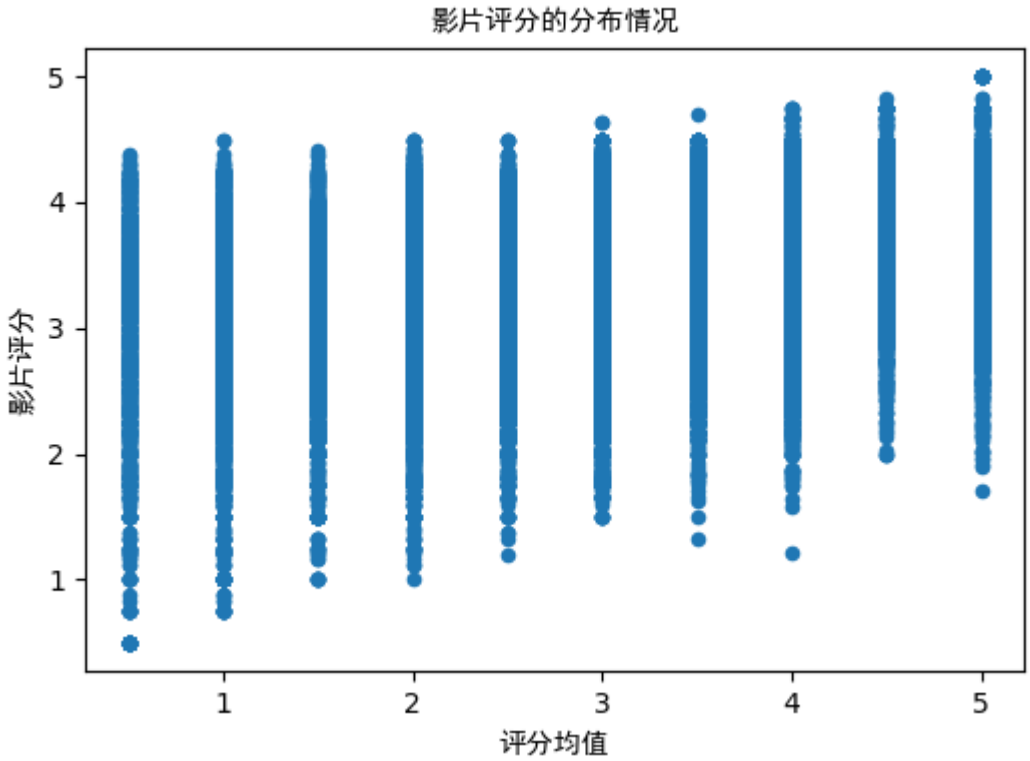


5. 散点图

方法： `series_obj.plot.scatter(x=, y= [,c= [,...]])` 或 `dataframe_obj.plot.scatter(x=,y= [,c= [,...]])` ，

```
# 将实际评分与平均分结合在一起
scatter = pd.merge(movie_ratings,ratings_mean,on='movieId')
scatter.plot.scatter(x='rating_x',y='rating_y')
# 散点图
plt.title(u"影片评分的分布情况", fontproperties="SimHei")
plt.xlabel(u"评分均值", fontproperties="SimHei")
plt.ylabel(u"影片评分", fontproperties="SimHei")
```

输出结果：

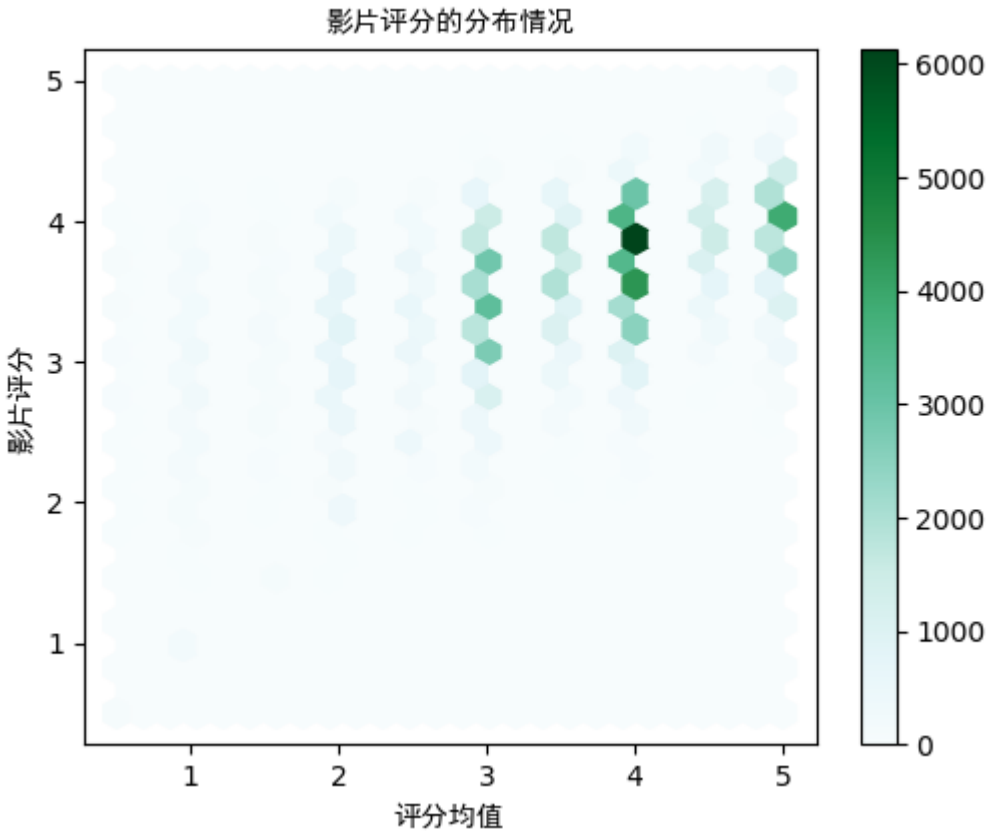


6. Hexagonal Bin图：

方法： `series_obj.plot.hexbin(x=, y= [,gridsize= [,...]])` 或 `dataframe_obj.plot.hexbin(x=,y= [,gridsize= [,...]])` ，

```
# Hexagonal Bin图:
scatter.plot.hexbin(x='rating_x',y='rating_y',gridsize=25)
```

输出结果：



(本节参考资料：http://pandas.pydata.org/pandas-docs/stable/visualization.html)

八、参考资料

- pandas 官方文档：http://pandas.pydata.org/pandas-docs/stable/index.html
- 易百教程-Pandas教程：https://www.yiibai.com/pandas/python_pandas_sorting.html