Notepad of <Python for Data Analysis>

Pandas

Kitsu Liu
Graduate School of Environment and Information Sciences
Yokohama National University

目录

1 基础

[1]:
```
import pandas as pd
#看一下有没有装上 pandas 加载并简称为 pd
import numpy as np
```

Pandas 是一个数据分析处理库, 封装于 Numpy 之上

---

1. 读文件进 Pandas, 以及一些基本操作:

这些命令想用来生成数组的话，得用赋值的方式来新建一个数组才行

1.1 pd.read_csv 读取 csv 文件, 以及数值查询:
pd.read_csv('file_path&name', sep, header, names, index_col, usecols, dtype, keep_default_na, na_values, parse_date)
各参数说明:
file_path&name: 文件路径（与代码文件同文件夹下就写个文件名就行了）
sep: 区切り文字の指定, 省略可。省略時はカンマ「,」が指定されたものとして扱われる
header: ヘッダー行の指定。省略可。省略時は読み込んだ内容からヘッダー行と列名を推測する
names: 列名の指定。省略可
index_col: 行ラベル（行にアクセスするためのインデックス, 主キー的那种味道）となる列の指定。省略可。省略時は各行には整数値でアクセスする
usecols: 使用 usecols = x 只取出第 x【列】，使用 usecols = (x,y,z) 仅仅只取出第 x,y,z【列】
dtype: 数组内的元素类型，默认是 float 等兼容性大的类型，由 pandas 自行推测
keep_default_na:na_values パラメーターに欠損値と見なされる値が指定された場合に、デフォルトで欠損値として見なしている値を保持するかどうかの指定。省略可。省略時は True が指定されたものとして見なされる（デフォルトの欠損値を保持する）
na_values: 欠損値として見なされる値の追加指定。省略可
parse_date = True/False: 見た目が日付型のデータを datetime64 结构にするかどうか

1.2 data_name = pd.read_csv() 将读取进来的 csv 文件数据在 Pandas 中命名为 data_name
1.3 data_name.head(n) 从数据中仅抽取前 n 行, 不写的话默认是 5
1.4 data_name.info() 查看数据信息，各列的具体信息，数据类型
1.5 data_name.index 查看数据的索引 (主 key)
1.6 data_name.columns 查看数据的各列的名字

1.7 data_name.dtypes 查看各列的数据类型

1.8 data_name.values 近查看数据的各列的值, 注意：取出来的值会变成 ndarray 结构

1.9 data_name['column_name'] 单独取出某一列，大小写要注意

1.10 data_name.set_index('column_name') 将某一属性 (列) 设为 table 的索引 (index), 类似于主キー的感觉, 大小写要注意

1.11 data_name.describe(参数) 拉取 table 中的所有各种统计量，不写参数默认拉取整个 table，参数中可以写行的 index 去拉取单独某一行，或者是写 **'column_name'** 去单独拉取某一列

---

2. 创建数据：

这些命令想用来生成数组的话，得用赋值的方式来新建一个数组才行

2.1 pd.DataFrame(array_like_a_table) 创建 DataFrame 结构（把一组有列结构的 array 传进去，生成一个表 (table)）

但是传进去的 array 最好是这个形式'：

array_as_table = {'columns1':['aaa','bbb','ccc'],'columns2':['ddd','eee','fff'].....}

关于什么是DataFrame结构以及应用：点击这里

2.2 pd.Series(data=array_value, index = array_index, dtype=None, name=None,copy=False) 创建 Series 结构，类似表格中的一个列（column），类似于一维数组，可以保存任何数据类型,Series 由索引（index）和列组成

参数解释详细参考:第三方说明（click this）

Series 可以有 index 列的列名，只能在从 DataFrame 里抽出某一列时才有，因为 Series 可以理解成是 DataFrame 里的一列，另外使用 pd.MuitiIndex 命令生成 index 然后代入进 Series 也可以让 Series 有 index 列的列名。

关于什么是Series结构以及应用：点击这里

1.1 读文件进 Pandas 等基本操作

[5]:
```
data_test = pd.read_csv('titanic.csv')
#把数据读进来并命名为 data_test
data_test
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

[6]:
```
data_test.head(4)
#仅抽取前四行
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S

[7]:
```
data_test.info()
#查看数据各列的信息
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
```

```
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

[8]:
```
data_test.index
#查看数据的索引
```

[8]:
```
RangeIndex(start=0, stop=891, step=1)
```

[9]:
```
data_test.columns
#查看数据的各列的名字
```

[9]:
```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

[10]:
```
data_test.dtypes
#查看各列的数据类型
```

[10]:
```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

[11]:
```
data_test.values
#近查看数据的各列的值
```

[11]:
```
array([[1, 0, 3, …, 7.25, nan, 'S'],
       [2, 1, 1, …, 71.2833, 'C85', 'C'],
       [3, 1, 3, …, 7.925, nan, 'S'],
       …,
       [889, 0, 3, …, 23.45, nan, 'S'],
       [890, 1, 1, …, 30.0, 'C148', 'C'],
       [891, 0, 3, …, 7.75, nan, 'Q']], dtype=object)
```

[20]:
```
type(data_test.values)
#data_name.values 取出来的列会变成 ndarray 结构
```

[20]:
```
numpy.ndarray
```

[18]:
```
data_test['Age']
#单独取出 Age 列
```

[18]:
```
0      22.0
1      38.0
2      26.0
3      35.0
4      35.0
       …
886    27.0
887    19.0
888     NaN
889    26.0
890    32.0
Name: Age, Length: 891, dtype: float64
```

[19]:
```
Age = data_test['Age']
Age[:5]
#还可以切片
```

[19]:
```
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
```

```
Name: Age, dtype: float64
```

[21]:
```
#还可以对单独取出来的东西应用上面的
print(Age.index)
print(Age.values[:5])
#对用 data_name.values 直接可以切片
```

```
RangeIndex(start=0, stop=891, step=1)

[22. 38. 26. 35. 35.]
```

[27]:
```
data_test.head(4)
#仅抽取前四行看一下原来的 table
#table 最左边是一个个序号作为 index，作用类似于主キー，是默认设置出来的
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S

[28]:
```
data_test.set_index('Name').head(4)
#将 table 的 index 设置为 Name 那个属性
#但是如果不做赋值操作的话不会更改原数据
```

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Name											
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	A/5 21171	7.2500	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	PC 17599	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	113803	53.1000	C123	S

[30]:
```
Age = data_test.set_index('Name')['Age']
Age.head(4)
#设定完 index 后来看一下单独取出来的列
```

[30]:
```
Name
Braund, Mr. Owen Harris                             22.0
Cumings, Mrs. John Bradley (Florence Briggs Thayer) 38.0
Heikkinen, Miss. Laina                              26.0
Futrelle, Mrs. Jacques Heath (Lily May Peel)        35.0
Name: Age, dtype: float64
```

[33]:
```
#这样就可以直接查找到某个人的年龄单独取出来啦
Age['Braund, Mr. Owen Harris']
```

[33]: 22.0

[36]:
```
print(Age.head(4) + 10)
print(Age.head(4) * 10)
#可以直接做加减乘除
#数据太长，只取前4行来演示
```

```
Name
Braund, Mr. Owen Harris                             32.0
Cumings, Mrs. John Bradley (Florence Briggs Thayer)    48.0
Heikkinen, Miss. Laina                              36.0
Futrelle, Mrs. Jacques Heath (Lily May Peel)        45.0
Name: Age, dtype: float64


Name
Braund, Mr. Owen Harris                             220.0
Cumings, Mrs. John Bradley (Florence Briggs Thayer)    380.0
Heikkinen, Miss. Laina                              260.0
Futrelle, Mrs. Jacques Heath (Lily May Peel)        350.0
Name: Age, dtype: float64
```

[46]:
```
print("最大值:",Age.max())
print("最小值:",Age.min())
print("平均值:",Age.mean())
print("標準偏差:",Age.std())
print("分散:",Age.var())
#当然，还可以拉取其他统计量
```

```
最大值: 80.0

最小值: 0.42

平均值: 29.69911764705882

標準偏差: 14.526497332334044
```

分散：211.0191247463081

[47]:
```
data_test.describe()
#拉取全 table 的所有各种统计量
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

1.2 创建数据

[23]:
```
array_as_table = {'country':['aaa','bbb','ccc'],
                  'population':[10,20,30]}
#生成一个有完整结构的数组。跟写 sql 的 create 很像的
table_test1 = pd.DataFrame(array_as_table)
#把 array_as_table 传进去生成 table
print(array_as_table)
print(table_test1)
```

```
{'country': ['aaa', 'bbb', 'ccc'], 'population': [10, 20, 30]}

  country  population
0     aaa          10
1     bbb          20
2     ccc          30
```

[17]:
```
table_test1.info()
#用上面学的命令查一下我们创建的数据
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   country     3 non-null      object
 1   population  3 non-null      int64 
dtypes: int64(1), object(1)
memory usage: 176.0+ bytes
```

[49]:
```
array_value = [11,22,32,43,54]
array_index = ['1st','2nd','3rd','4th','5th']
pd.Series(array_value,array_index)
```

[49]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[47]:
```
array_value = [11,22,32,43,54]
array_index = ['1st','2nd','3rd','4th','5th']
a = pd.Series(array_value,array_index,name = 'Series')
print(a.index)
print("")
print(a.name)
#还有其他很多参数
#具体参考官方文稿: https://pandas.pydata.org/docs/reference/series.html
```

```
Index(['1st', '2nd', '3rd', '4th', '5th'], dtype='object')

Series
```

2 索引

2.1 一般索引

想要拉取‘列’的索引就跟之前一样，直接用括号写索引（列名）就好，但是想拉‘行’的索引的时候就要用下面这些函数了

1. **data_name.iloc[a:b,c:d]**，**找第 a 行到第 b 行 & 第 c 列到第 d 列的数据，当然可以只写行 or 列**，以数字（第几个）找数据

2. **data_name.loc['行索引名 1':'行索引名 2',c:d]**，**找‘行索引名 a’到‘行索引名 b’的数据中的第 c 列到第 d 列**，以‘文字’找数据

3. **data_name.column_name**，从 data_name 中取出名为 column_name 的列

4. **data_name.loc['index_name']**，从 data_name 中取出名为 index_name 的行

[2]:
```
data = pd.read_csv('titanic.csv')
```

[5]:
```
data['Age'][1:5]
#只拉一个列的索引
```

[5]:
```
1    38.0
2    26.0
3    35.0
4    35.0
Name: Age, dtype: float64
```

[7]:
```
data[['Age','Fare']][1:5]
#拉两个及以上的列的时候，需要双层括号，要注意!!
```

[7]:

	Age	Fare
1	38.0	71.2833
2	26.0	7.9250
3	35.0	53.1000
4	35.0	8.0500

[8]:
```
data[0]
#当我们想指定某一行数据的时候不能直接输入一个数进来
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)


KeyError: 0
```

[9]:
```
data.iloc[0]
#拿第一行（第一个）数据
```

[9]:
```
PassengerId                     1
Survived                        0
Pclass                          3
Name            Braund, Mr. Owen Harris
Sex                            male
Age                            22.0
SibSp                             1
Parch                             0
Ticket                       A/5 21171
Fare                            7.25
Cabin                            NaN
Embarked                           S
Name: 0, dtype: object
```

[11]:
```
data.iloc[0:5,0:2]
#去第一行到第五行的数据中的第一列到第 2 列
```

[11]:

	PassengerId	Survived
0	1	0
1	2	1
2	3	1
3	4	1
4	5	0

[13]:
```
data2 = data.set_index('Name')  #将 Name 列作为索引 index
data2.loc['Braund, Mr. Owen Harris']   #拉出名为 Braund, Mr. Owen Harris 这个人的
那一行的数据
```

[13]:
```
PassengerId            1
Survived               0
Pclass                 3
Sex                 male
Age                 22.0
SibSp                  1
Parch                  0
Ticket         A/5 21171
Fare                7.25
Cabin                NaN
Embarked               S
Name: Braund, Mr. Owen Harris, dtype: object
```

[14]:
```
data2.iloc[0]
#因为 Braund, Mr. Owen Harris 是第 0 行，所以结果跟上面是一样的
```

[14]:
```
PassengerId            1
Survived               0
Pclass                 3
Sex                 male
Age                 22.0
SibSp                  1
Parch                  0
Ticket         A/5 21171
Fare                7.25
Cabin                NaN
Embarked               S
Name: Braund, Mr. Owen Harris, dtype: object
```

[15]:
```
data2.loc['Braund, Mr. Owen Harris','Fare']
#找【Braund, Mr. Owen Harris】行的【Fare】列的数据
```

[15]:
```
7.25
```

[16]:
```
data2.loc['Braund, Mr. Owen Harris':'Allen, Mr. William Henry',:]
#从【Braund, Mr. Owen Harris】行到【Allen, Mr. William Henry】行，并取其中所有列
```

Name	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	A/5 21171	7.2500	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	PC 17599	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	113803	53.1000	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	0	373450	8.0500	NaN	S

[17]:
```
data2.loc['Allen, Mr. William Henry','Fare'] = 1000
data2.loc['Allen, Mr. William Henry']
#当然也可以用索引抓取到某个属性取赋值
#看 Fare 那个属性变成 1000 了
```

[17]:
```
PassengerId         5
Survived            0
Pclass              3
Sex              male
Age              35.0
SibSp               0
Parch               0
Ticket         373450
Fare           1000.0
Cabin             NaN
Embarked            S
Name: Allen, Mr. William Henry, dtype: object
```

2.2 bool 布尔类型的索引

可以利用这个做到一些高级玩法

比如显示性别的男性的所有数据

求数据中的男性的年龄的平均值

数据中年龄大于 70 的有几个人

[4]:
```
data.iloc[0:5]['Fare'] > 30
#前五行的 Fare 列的值是否大于 30
```

[4]:
```
0    False
1     True
2    False
```

```
3     True
4    False
Name: Fare, dtype: bool
```

[8]:
```
data[data['Fare'] > 30].iloc[0:5]
#把大于 30 的判定结果的 bool 数组传回 data 里，就可以做到只筛选出 true 的部分了
#在 Fare>30 的结果中，只要前 5 个
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	237736	30.0708	NaN	C
13	14	0	3	Andersson, Mr. Anders Johan	male	39.0	1	5	347082	31.2750	NaN	S

2.3 高级查询/条件查询

可以利用这个做到一些高级玩法
比如显示性别的男性的所有数据
求数据中的男性的年龄的平均值
数据中年龄大于 70 的有几个人

[9]:
```
data['Sex'] == 'male'
#以 Sex 列进行判断，male 为 true
```

[9]:
```
0       True
1      False
2      False
3      False
4       True
       …
886     True
887    False
888    False
889     True
890     True
Name: Sex, Length: 891, dtype: bool
```

[10]:
```
data[data['Sex'] == 'male'].iloc[0:5]
#传回 data, 做到只取 sex 为 male 的行的前五个
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S

[17]:
```
data.loc[data['Sex'] == 'male'].head(5)
#用了.loc[]但是结果也是仅仅理解成 sex 为 male 的查询结果即可
#结果太长设置只显示前五个
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S

[18]:
```
data.loc[data['Sex'] == 'male','Age'].head(5)
#在上一条中加了 Age, 结果为仅抽出男性的年龄数据
#结果太长设置只显示前五个
```

[18]:
```
0    22.0
4    35.0
5     NaN
6    54.0
7     2.0
Name: Age, dtype: float64
```

[20]:
```
data.loc[data['Sex'] == 'male','Age'].mean()
#在上一条中加了.mean(), 结果为所有男性的年龄的平均
```

[20]:
```
30.72664459161148
```

[22]:
```
(data['Age'] > 70).sum()
#年龄大于 70 的有几个
#(data['Age'] > 70) 出来的是个 bool 类型的数组
#然后.sum 是统计了里面 true 的个数
```

[22]: 5

3 Groupby

3.1 基础

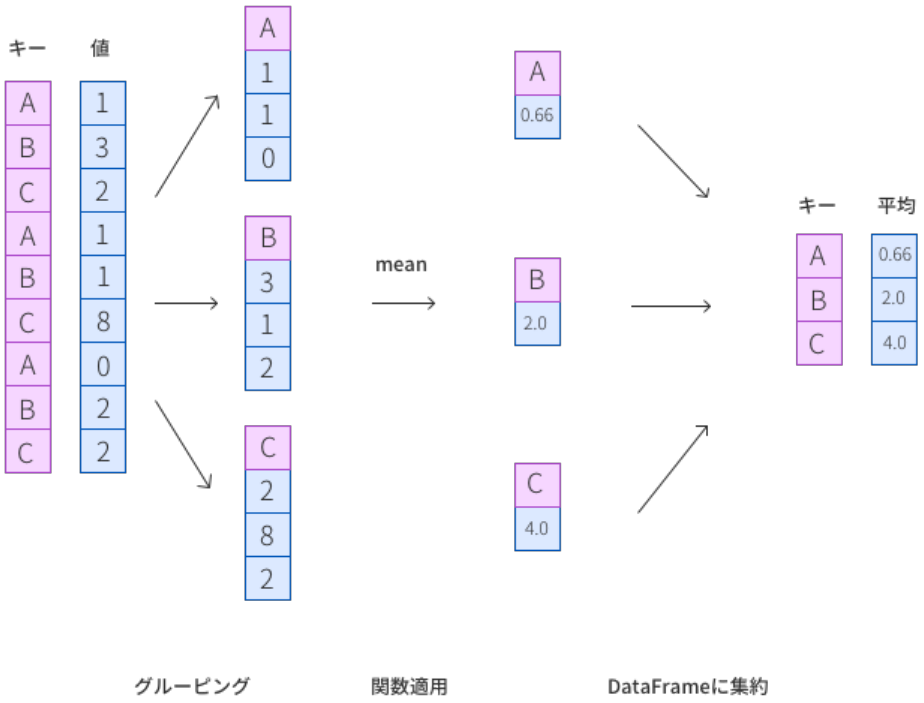
与 Relational DataBase 的 SQL 中的 groupby 一个意思

data_name.group('xxx')['列名']. 指令，根据【xxx 列】的值来给数据分组，这时的【xxx 列】的各个数据就变成了访问分组后各组的 key 值了

['列名'] 意为分组后抽出指定列 (跟上面作为分组指标的列不一样)，这一条可以不写，默认为选定全局

指令 可以是：**sum()**, **max()** 等等

示意图如下：



[24]:
```
# 新建数据
data = pd.DataFrame({'key':['a','b','c','a','b','c','a','b','c'],
                     'value':[0,5,10,5,10,15,10,15,20]})
# 注意 DataFrame 的 D 和 F 必须要大写
data
```

[24]:

	key	value
0	a	0

```
1   b      5
2   c     10
3   a      5
4   b     10
5   c     15
6   a     10
7   b     15
8   c     20
```

[38]:
```
data.groupby('key').sum()
#以【名为 key 的列】的值进行分组，并求出各组的值的和
```

[38]:
```
     value
key
a       15
b       30
c       45
```

[39]:
```
data.groupby('key').max()
#以【名为 key 的列】的值进行分组，并求出各组的值的最大值
```

[39]:
```
     value
key
a       10
b       15
c       20
```

[40]:
```
data.groupby('key').min()
#以【名为 key 的列】的值进行分组，并求出各组的值的最小值
```

[40]:
```
     value
key
a        0
b        5
c       10
```

[41]:
```
data.groupby('key').mean()
#以【名为 key 的列】的值进行分组，并求出各组的值的平均值
```

[41]:
```
      value
key
a         5
b        10
c        15
```

[42]:
```
data.groupby('key').std()
#以【名为 key 的列】的值进行分组，并求出各组的值的標準偏差
```

[42]:
```
      value
key
a       5.0
b       5.0
c       5.0
```

[43]:
```
data.groupby('key').var()
#以【名为 key 的列】的值进行分组，并求出各组的值的分散
```

[43]:
```
      value
key
a        25
b        25
c        25
```

[48]:
```
data = pd.read_csv('titanic.csv')
#读一下数据，用这个数据来进行接下来的演示
data.groupby('Sex')['Age'].mean()
#先以 Sex 的值来分组，然后分别根据组内的 age 的数据求平均
#简称：求各性别的年龄的平均值
```

[48]:
```
Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

[51]:
```
data.groupby('Sex')['Survived'].mean()
#原理同上
#查看不同性别的获救情况
#虽然这里求的是平均值，但是这个结果更像是获救的比例，女性的 74% 的人获救了
```

[51]:
```
Sex
female    0.742038
male      0.188908
Name: Survived, dtype: float64
```

3.2 GROUPBY 练习

再进行更详细讲解前，先需要了解一下什么是MultiIndex结构，字面意思是就是【多层 index】
其实可以理解成【多层主 key】，以下图为例：

				title
genre	year	month	day	
music	2017	2	12	The singer sold 1 million CDs.
sports	2017	3	12	Team A won the first game of the season.
		4	14	The athlete will retire by next month.
	2016	12	13	Team B lose for the first time in the season.
politics	2016	8	21	C on Trump tweet calling D a 'dog'.
	2017	5	29	E won the election.
health	2016	5	30	Running effectives.
	2017	2	21	Best foods for health.

このデータの場合、ラベルが genre, year, month, day の 4 つとなっており、genre > year > month > day の順にグループ分けがされています。
也就是说，它的主 key 是有“优先度”的，显按照最外面的优先度最高的进行分组后（这时各分组的“组名”就是某种意义上的主 key），再一层一层的进行分组。
详见：click this

1. groupby 的命令全貌

DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=Tru

参数解释：

by = 'xxx'：指定以谁为基准，这里可以写一个列名，也可以带进来一个函数，用于自定义想要构成的分组。

axis = 0(‘index’)/1(‘columns’)：可省略，默认 0，在【行】内还是【列】内,0 为行，1 为列

level = int, level name,default None：可省略，默认 None，如果 index 是多层的，也就是 MultiIndex 结构时，指定一下在哪一个层面进行 groupby 操作, 最外侧是 0

as_index = True/False：可省略，默认 True，是否把数据原来自己的 index（标签）用于 groupby 分组后的数据

sort = True/False：可省略，默认 True，是否把 groupby 分组后的组名（主 key）进行排序操作，设置关闭可以使得运算更快，而且这个虽然会对组名（主 key）进行排序操作，但是组的数据

的顺序并不会受到影响

group_keys = True/False: 可省略，默认 True，是否把 groupby 分组后的组名（主 key）进行排序操作，设置关闭可以使得运算更快，而且这个虽然会对组名（主 key）进行排序操作，但是组的数据的顺序并不会受到影响

dropna = True/False: 可省略，默认 True，If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

解释详见：官方文稿 (click this)

实际应用例子详见：click this

2. `grouped_data.groups`: 经过 groupeby 后的 obeject 可以使用.groups 来看一下グループの内訳

3. `grouped_data.get_group('xxx')`: 经过 groupeby 后的 obeject 可以使用.get_group 来看一下 xxx 这个分组内的所有数据元素

4. `grouped_data.agg()`:agg 同.aggregate，是用来对 DataFrame 结构进行函数带入的

イメージ: DataFrame と実行したい関数 →.agg() に代入 → 項目毎の関数結果からなる DataFrame

对数据里的每一列（默认）进行函数操作，比如求和，平均，等等。这个函数可以是自己写的

详细参考：click this

5. `grouped_data(或者是普通的 DataFrame).reset_index()`: 把 index（索引）重置为 0，1，2，3，4…这样的从 0 开始数组，如果是对经过 groupby 的结构的求和进行这个函数的话【形如：`grouped_data.sum().reset_index()`】，会还原成一个完全没有进行 groupby 操作的 DataFrame 结构，同时 index 会是从 0 开始一条一条的数据（因为某种意义上来讲，groupby 操作就是给重新做了个有分组感的 index）

6. `grouped_data(或者是普通的 DataFrame).size()`: 对于普通的 DataFrame 来说是看一下表里有几行数据，但是对 groupby 后的 DataFrame 使用的话，就是看看各个分组里有几个条目（行）的数据

[3]:
```
df = pd.DataFrame({'A':['foo','bar','foo','bar','foo','bar','foo','foo'],
                   'B':['one','one','two','three','two','two','one','three'],
                   'C':np.random.randint(1,200,8),
                   'D':np.random.randint(100,300,8)})
df
#生成一个数据用于演示
```

[3]:

	A	B	C	D
0	foo	one	92	290
1	bar	one	30	244
2	foo	two	197	243

```
3  bar  three  171  238
4  foo    two   13  129
5  bar    two  195  196
6  foo    one  153  181
7  foo  three   35  175
```

[4]:
```
grouped = df.groupby('A')
#以 A 里的分类为基准去分组
grouped.count()
```

[4]:
```
     B  C  D
A
bar  3  3  3
foo  5  5  5
```

[5]:
```
grouped2 = df.groupby(['A','B'])
grouped2.count()
#以 A 列 +B 列作为 groupby 依据去分组
```

[5]:
```
            C  D
A   B
bar one     1  1
    three   1  1
    two     1  1
foo one     2  2
    three   1  1
    two     2  2
```

[77]:
```
def check_if_it_includes_vowel(word):
        if word.lower() in 'abcd':
            return 'Yes'
        else:
            return 'No'

grouped3 = df.groupby(by = check_if_it_includes_vowel,axis = 1)
grouped3.count()
#如果 by 写的是函数的话，传进去的默认是【行 label(行名)】(axis = 0) 和【列名】
(axis=1)
```

```
#并不是里面的数据
```

[77]:
```
   Yes
0    4
1    4
2    4
3    4
4    4
5    4
6    4
7    4
```

[122]:
```
grouped3.groups
#经过 groupeby 后的 obeject 可以使用 .groups 来看一下グループの内訳
```

[122]:
```
{'Yes': ['A', 'B', 'C', 'D']}
```

[30]:
```
s = pd.Series([9,8,7,6,4,3],[0,1,2,0,1,2])
s
```

[30]:
```
0    9
1    8
2    7
0    6
1    4
2    3
dtype: int64
```

[32]:
```
grouped = s.groupby(by = '0')
#这里想写第 0 列发现没办法表达出来，就会报错了
#但是这个 Series 结构里它没定义【列】名这种东西，所以没法写 by=xx 这个参数
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/yk/g6zs19fd5jz847sc8_ndgs240000gn/T/ipykernel_8924/2777108394.py i
 ↪<module>
----> 1 grouped = s.groupby(by = '0')
      2
```

```
KeyError: '0'
```

[33]:
```
#这时候我们就可以用 level 这个命令来指定使用最外面的那层主 key，也就是 Series 的 index 列了
grouped = s.groupby(level = 0)
#用最外层的 index 进行 groupby 操作，这样就可以代替指定 by= 了
grouped4.sum()
#做一下各个分组的求和
```

[33]:
```
0    15
1    12
2    10
dtype: int64
```

[34]:
```
df2 = pd.DataFrame({'X':['A','B','A','B'],
                    'Y':np.arange(1,5,1)})
df2
```

[34]:

	X	Y
0	A	1
1	B	2
2	A	3
3	B	4

[35]:
```
df2.groupby(['X']).get_group('A')
#看一下分组 A 里的的所有元素
```

[35]:

	X	Y
0	A	1
2	A	3

[36]:
```
df2.groupby(['X']).get_group('B')
#看一下 B 里的
```

[36]:

	X	Y
1	B	2
3	B	4

[42]:
```
array_index = [['foo','foo','bar','bar','foo','foo','bar','bar'],
               ['one','two','two','one','one','two','one','two']]
index = pd.MultiIndex.from_arrays(array_index,names= ['first','second'])
s = pd.Series(np.random.randint(1,100,8),index = index)
s
```

[42]:
```
first  second
foo    one       66
       two       55
bar    two       16
       one       42
foo    one        9
       two       50
bar    one       90
       two       94
dtype: int64
```

[51]:
```
grouped = s.groupby(level = 0)
print(grouped.groups)
print(grouped.get_group('bar'))
print(grouped.get_group('foo'))
grouped2 = s.groupby(level = 1)
print(grouped2.groups)
print(grouped2.get_group('one'))
print(grouped2.get_group('two'))
#在不同的 level 进行 groupby 的结果是不一样的
#尽管分组的样子不太一样，但是会发现对实质上的最后一列的值的顺序并没有影响
```

```
{'bar': [('bar', 'two'), ('bar', 'one'), ('bar', 'one'), ('bar', 'two')], 'foo':
[('foo', 'one'), ('foo', 'two'), ('foo', 'one'), ('foo', 'two')]}

first  second
bar    two       16
       one       42
       one       90
       two       94
dtype: int64
```

```
first  second
foo    one       66
       two       55
       one        9
       two       50
dtype: int64

{'one': [('foo', 'one'), ('bar', 'one'), ('foo', 'one'), ('bar', 'one')], 'two':
[('foo', 'two'), ('bar', 'two'), ('foo', 'two'), ('bar', 'two')]}

first  second
foo    one       66
bar    one       42
foo    one        9
bar    one       90
dtype: int64

first  second
foo    two       55
bar    two       16
foo    two       50
bar    two       94
dtype: int64
```

[62]:
```
df = pd.DataFrame({'A':['foo','bar','foo','bar','foo','bar','foo','foo'],
                   'B':['one','one','two','three','two','two','one','three'],
                   'C':np.random.randint(1,200,8),
                   'D':np.random.randint(100,300,8)})
df
#生成一个数据用于演示
```

[62]:

	A	B	C	D
0	foo	one	182	135
1	bar	one	28	293
2	foo	two	105	153
3	bar	three	166	232
4	foo	two	51	196

```
5  bar    two   17  243
6  foo    one   71  240
7  foo  three   70  121
```

[64]:
```
grouped = df.groupby(['A','B'])
grouped.groups
#注意，index是多值的
```

[64]:
```
{('bar', 'one'): [1], ('bar', 'three'): [3], ('bar', 'two'): [5], ('foo',
'one'): [0, 6], ('foo', 'three'): [7], ('foo', 'two'): [2, 4]}
```

[65]:
```
grouped.get_group(('bar','one'))
#所以写.get_group时要代入的index也是MultiIndex结构
```

[65]:
```
     A    B   C    D
1  bar  one  28  293
```

[66]:
```
grouped.sum().reset_index()
#使用.reset_index()把index（索引）重置为0，1，2，3，4...这样的从0开始数组
#如果是对经过groupby的结构的求和进行这个函数的话【形如：grouped_data.sum().
↪reset_index()】
#会还原成一个完全没有进行groupby操作的DataFrame结构
#同时index会是从0开始一条一条的数据
#（因为某种意义上来讲，groupby操作就是给重新做了个有分组感的index）
```

[66]:
```
     A      B    C    D
0  bar    one   28  293
1  bar  three  166  232
2  bar    two   17  243
3  foo    one  253  375
4  foo  three   70  121
5  foo    two  156  349
```

[67]:
```
grouped.size()
#本来.size是用来看一下有几行数据的
#对groupby后的DataFrame使用的话，就是看看各个分组里有几个条目的数据
```

		C								D						
		count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
A	B															
	one	1.0	28.0	NaN	28.0	28.00	28.0	28.00	28.0	1.0	293.0	NaN	293.0	293.00	293.0	293.00	293.0
bar	three	1.0	166.0	NaN	166.0	166.00	166.0	166.00	166.0	1.0	232.0	NaN	232.0	232.00	232.0	232.00	232.0
	two	1.0	17.0	NaN	17.0	17.00	17.0	17.00	17.0	1.0	243.0	NaN	243.0	243.00	243.0	243.00	243.0
	one	2.0	126.5	78.488853	71.0	98.75	126.5	154.25	182.0	2.0	187.5	74.246212	135.0	161.25	187.5	213.75	240.0
foo	three	1.0	70.0	NaN	70.0	70.00	70.0	70.00	70.0	1.0	121.0	NaN	121.0	121.00	121.0	121.00	121.0
	two	2.0	78.0	38.183766	51.0	64.50	78.0	91.50	105.0	2.0	174.5	30.405592	153.0	163.75	174.5	185.25	196.0

[63]:
```
grouped.agg(np.sum)
#.agg 同 .aggregate，是用来对 DataFrame 结构进行函数带入的
#イメージ：DataFrame と実行したい関数 →.agg() に代入 → 項目毎の関数結果からなる
DataFrame
#对数据里的每一列（默认）进行函数操作，比如求和，平均，等等。这个函数可以是自己写的
#上面这个例子里，因为进行了 groupby 操作，所以 .agg 的默认对象就变成了各个分组
#同时因为 bar 分组到最后一级的条目里的数据是单独一条数据，没有重复的
#那么自然每个小组里就只有一条数据
#所以 bar 分组中 sum 出来的值就是对单个数据求和，那么输出的结果也自然是单个数据
#因此对于 bar 分组来说形成的结果就是把 groupby 操作后的表给打印出来了
#但是 foo 分组里是有重复的，也就是说各个小分组里是有多个条目的
#因此就是变成了求和
```

[63]:

		C	D
A	B		
bar	one	28	293
	three	166	232
	two	17	243
foo	one	253	375
	three	70	121
	two	156	349

[78]:
```
grouped2 = df.groupby('A')
grouped2[['C','D']].agg([np.sum,np.mean,np.std])
#agg 的一个标准用法，对 DataFrame 的每一个小部分使用函数后，把结果统合成 DataFrame 结构
```

[78]:

	C			D		
	sum	mean	std	sum	mean	std

```
A
bar  211  70.333333  83.032122  768  256.0  32.511536
foo  479  95.800000  51.968260  845  169.0  48.698049
```

[90]:
```
grouped2.agg({'C':['sum','mean','std'],'D':['sum','mean','std']})
#使用字典结构的写法，同时使用了 pd 的自带的函数，而不是传进去 numpy 的函数
```

[90]:
```
       C                          D
     sum       mean        std  sum   mean        std
A
bar  211  70.333333  83.032122  768  256.0  32.511536
foo  479  95.800000  51.968260  845  169.0  48.698049
```

4 运算

4.1 单纯数学运算

以下所有命令想用来生成数组的话，得用赋值的方式来新建一个数组才行

1. **data_name.sum(参数)，求和**
如果不写参数则默认是以【列】为单位求和，把一列的所有元素加起来

2. **data_name.means(参数)，求平均值**
如果不写参数则默认是以【列】为单位求平均值

3. **data_name.min(参数)，求最小值**
如果不写参数则默认是以【列】为单位求最小值

4. **data_name.max(参数)，求最大值**
如果不写参数则默认是以【列】为单位求最大值

5. **data_name.median(参数)，求中位数**
如果不写参数则默认是以【列】为单位求中位数

参数可写：如果写成 **axis = 0** 是指的第 0 维，还可以写成 **axis = 'columns'** 这样直接指定成在以【行】为单位做运算。如果不写参数则默认是以【列】为单位做运算

[14]:
```
#创建一个数据用来做演示
data = pd.DataFrame([[1,2,3],[4,5,6]],index = ['第一行','第二行'],columns = ['第一列','第二列','第三列'])
#index 是行名，columns 是列名
data
```

[14]:

	第一列	第二列	第三列
第一行	1	2	3
第二行	4	5	6

4.1.1 求和

[9]:
```
data.sum()
#以【列】为单位求和
```

[9]:
```
第一列    5
第二列    7
第三列    9
dtype: int64
```

[10]:
```
data.sum(axis = 0)
#以【列】为单位求和
```

[10]:
```
第一列    5
第二列    7
第三列    9
dtype: int64
```

[11]:
```
data.sum(axis = 1)
#以【行】为单位求和
```

[11]:
```
第一行     6
第二行    15
dtype: int64
```

[15]:
```
data.sum(axis = 'columns')
#以【行】为单位求和
```

[15]:
```
第一行     6
第二行    15
dtype: int64
```

[16]:
```
data.sum(axis = 'index')
#以【列】为单位求和
```

[16]:
```
第一列    5
第二列    7
第三列    9
dtype: int64
```

4.1.2 求平均值

[17]:
```
data.mean()
#以【列】为单位求平均值
```

[17]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[18]:
```
data.mean(axis = 0)
#以【列】为单位求平均值
```

[18]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[19]:
```
data.mean(axis = 1)
#以【行】为单位求平均值
```

[19]:
```
第一行    2.0
第二行    5.0
dtype: float64
```

[20]:
```
data.mean(axis = 'index')
#以【列】为单位求平均值
```

[20]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[21]:
```
data.mean(axis = 'columns')
#以【行】为单位求平均值
```

[21]:
```
第一行    2.0
第二行    5.0
dtype: float64
```

4.1.3 求最小值

[22]:
```
data.min()
#以【列】为单位求最小值
```

[22]:
```
第一列    1
第二列    2
第三列    3
dtype: int64
```

[23]:
```
data.min(axis = 0)
#以【列】为单位求最小值
```

[23]:
```
第一列    1
第二列    2
第三列    3
dtype: int64
```

[24]:
```
data.min(axis = 1)
#以【行】为单位求最小值
```

[24]:
```
第一行    1
第二行    4
dtype: int64
```

[25]:
```
data.min(axis = 'index')
#以【列】为单位求最小值
```

[25]:
```
第一列    1
第二列    2
第三列    3
dtype: int64
```

[26]:
```
data.min(axis = 'columns')
#以【行】为单位求最小值
```

[26]:
```
第一行    1
第二行    4
dtype: int64
```

4.1.4 求最大值

[27]:
```
data.max()
#以【列】为单位求最大值
```

[27]:
```
第一列    4
第二列    5
第三列    6
dtype: int64
```

[28]:
```
data.max(axis = 0)
#以【列】为单位求最大值
```

[28]:
```
第一列    4
第二列    5
第三列    6
dtype: int64
```

[29]:
```
data.max(axis = 1)
#以【行】为单位求最大值
```

[29]:
```
第一行    3
第二行    6
dtype: int64
```

[30]:
```
data.max(axis = 'index')
#以【列】为单位求最大值
```

[30]:
```
第一列    4
第二列    5
第三列    6
dtype: int64
```

[31]:
```
data.max(axis = 'columns')
#以【行】为单位求最大值
```

[31]:
```
第一行    3
第二行    6
dtype: int64
```

4.1.5 求中位数

[32]:
```
data.median()
#以【列】为单位求中位数
```

[32]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[33]:
```
data.median(axis = 0)
#以【列】为单位求中位数
```

[33]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[34]:
```
data.median(axis = 1)
#以【行】为单位求中位数
```

[34]:
```
第一行    2.0
第二行    5.0
dtype: float64
```

[35]:
```
data.median(axis = 'index')
#以【列】为单位求中位数
```

[35]:
```
第一列    2.5
第二列    3.5
第三列    4.5
dtype: float64
```

[36]:
```
data.median(axis = 'columns')
#以【行】为单位求中位数
```

[36]:
```
第一行    2.0
第二行    5.0
dtype: float64
```

4.2 统计型运算

以下所有命令想用来生成数组的话，得用赋值的方式来新建一个数组才行

1. `data_name.cov()`，求协方差矩阵 (共分散行列)
生成的矩阵一定是 n*n 的，矩阵中的对应位置的元素表达某两个列数据之间的协方差

2. `data_name.corr()`，求相关矩阵 (相関行列)
生成的矩阵一定是 **n*n** 的，矩阵中的对应位置的元素表达某两个列数据之间的相关系数
相关系数是协方差经过标准化得出的量，在-1 到 1 之间

3. `data_name['列名'].value_counts(参数)`，指定某一【列】，看数据中各元素的个数（出现频率）
参数可写：
ascendign = True 是按个数的升序排列，不写默认是按降序
bins = x 自动分成 x 个区间，并统计区间内有多少个元素（不过这个分组很机械化，各组区间的大小相等，平均算出来的临界值看起来没有那么整齐）

4. `data_name['列名'].count()`，数一下这个【列】有几个数据 (元素)

[39]:
```
data = pd.read_csv('titanic.csv')
#把用来做例子的数据导入进来
data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

[40]:
```
data.cov()
#求协方差矩阵（共分散行列）
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
PassengerId	66231.000000	-0.626966	-7.561798	138.696504	-16.325843	-0.342697	161.883369
Survived	-0.626966	0.236772	-0.137703	-0.551296	-0.018954	0.032017	6.221787
Pclass	-7.561798	-0.137703	0.699015	-4.496004	0.076599	0.012429	-22.830196
Age	138.696504	-0.551296	-4.496004	211.019125	-4.163334	-2.344191	73.849030
SibSp	-16.325843	-0.018954	0.076599	-4.163334	1.216043	0.368739	8.748734
Parch	-0.342697	0.032017	0.012429	-2.344191	0.368739	0.649728	8.661052
Fare	161.883369	6.221787	-22.830196	73.849030	8.748734	8.661052	2469.436846

[41]:
```
data.corr()
#求相关矩阵（相関行列）
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
PassengerId	1.000000	-0.005007	-0.035144	0.036847	-0.057527	-0.001652	0.012658
Survived	-0.005007	1.000000	-0.338481	-0.077221	-0.035322	0.081629	0.257307
Pclass	-0.035144	-0.338481	1.000000	-0.369226	0.083081	0.018443	-0.549500
Age	0.036847	-0.077221	-0.369226	1.000000	-0.308247	-0.189119	0.096067
SibSp	-0.057527	-0.035322	0.083081	-0.308247	1.000000	0.414838	0.159651
Parch	-0.001652	0.081629	0.018443	-0.189119	0.414838	1.000000	0.216225
Fare	0.012658	0.257307	-0.549500	0.096067	0.159651	0.216225	1.000000

[42]:
```
data['Age'].value_counts()
#看 Age 这列都有什么元素，每个元素有几个
#简称看看不同年龄分别有多少人
#默认按的降序
```

[42]:
```
24.00    30
22.00    27
18.00    26
28.00    25
19.00    25
         ..
55.50     1
74.00     1
0.92      1
```

```
70.50     1
12.00     1
Name: Age, Length: 88, dtype: int64
```

[43]:
```
data['Age'].value_counts(ascending = True)
#升序排列
```

[43]:
```
0.67      1
14.50     1
36.50     1
74.00     1
24.50     1
         ..
19.00    25
30.00    25
18.00    26
22.00    27
24.00    30
Name: Age, Length: 88, dtype: int64
```

[45]:
```
data['Age'].value_counts(ascending = True,bins = 5)
#给分成5组，按照数据值来平均分（不是各数据的个数）
#最大值是80，最小值是0，基本按照16岁一个区间来切的。
```

[45]:
```
(64.084, 80.0]       11
(48.168, 64.084]     69
(0.339, 16.336]     100
(32.252, 48.168]    188
(16.336, 32.252]    346
Name: Age, dtype: int64
```

[47]:
```
data['Age'].count()
#数一下Age列里有几个数据（元素）
```

[47]:
```
714
```

5 一些对象操作

5.1 对某个对象进行增删改查

要注意：跟 Numpy 第 14 页一样，在比如做 arraye2 = array1 这样用一个数组给另一个数组赋值时，因为 python 中走的是类似 C 里面的指针的感觉，不会再单独在内存里开一个新的位置给后面的数组，是共享内存中同一块位置的。所以对后面的新数组做更改时，原数组也会被更改
因此要使用 **data/array_name.copy()** 这个命令最好

1. data_name.replace() 批量替换数据的值:
data_name.replace(to_replace=None,value=None,inplace=False,limit=None)
各参数说明： 具体参考 (click)
to_replace：置換したい値を指定する
value:to_replace で指定された値を置換する値をここで指定する
inplace:bool 値 (省略可能) 初期値は False である。True の時は変更を元のデータに反映させる
limit:int 値 (省略可能) 初期値 None。値を補完する時の最大繰り返し回数を指定する
df.columns = df.columns.str.replace() 可以修改列名做到 rename 的效果（在 5.5 章中介绍）

2. data_name.rename() 批量更改/替换数据的索引:
data_name.rename(mapper=None,index=None,columns=None,axis=None,inplace=False)
各参数说明： 具体参考 (click)
mapper = {'想更改的值':'换成什么值'}：指定想换的 mapper(行名 index + 列名 column 的合集) 改成什么, 会【同时对行名以及列名】进行影响
index = {'想更改的值':'换成什么值'}：仅对【行名】进行影响
columns = {'想更改的值':'换成什么值'}：仅对【列名】进行影响
axis = xx：可写：axis = 0 或者是 axis = 'index' 这样的，指定对哪一个维度进行影响，配合 mapper 进行使用，0 等于 index，1 等于 columns
inplace:bool 値 (省略可能) 初期値は False である。True の時は変更を元のデータに反映させる

3. data_name.append() 拼接数据（Series 和 DataFrame 均可）：
3.1 data_name.append(data_name_another,ignore_index = False)
各参数说明：
data_name：基于哪个 Series/DataFrame，也就是 base，会在拼接后排在上面
data_name_another：作为材料，被添加到 base 里的
gnore_index = False：【省略可】默认 False，写 True 会去掉之前的所有索引 index，改为默认的从 0 开始的数字索引
对 Series 使用：效果是上下的简单拼接
对 DataFrame 使用：拼接成一个新的表，虽然是拼接了，但是因为行数行列数都增加了，所以会

产生一些值没有经过定义，因此写为 NaN，但是这个 NaN 也是一个‘值’，可以被查询出来，不是摆设

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

+

	d	e	f
0	11	22	33
1	44	55	66
2	77	88	99

=

	a	b	c	d	e	f
0	1.0	2.0	3.0	NaN	NaN	NaN
1	4.0	5.0	6.0	NaN	NaN	NaN
2	7.0	8.0	9.0	NaN	NaN	NaN
0	NaN	NaN	NaN	11.0	22.0	33.0
1	NaN	NaN	NaN	44.0	55.0	66.0
2	NaN	NaN	NaN	77.0	88.0	99.0

3.2 pd.concat([data_1,data_2,....data_n],axis = 0)

各参数说明：

[data_1,data_2,....data_n]：指定数据名，将多个数据按顺序拼在一起

axis = 0：【省略可】默认 axis = 0【行 index】，写 1 为【列 index】

对 DataFrame 使用：把几个表去一行一行的拼时，如果两个表的列名完全一样，就会简单的上下去一行一行的拼接，不一样就会形成交叉错位式的插入【拼列时同理】，就向上面那张图（3.1）一样

4. 删除数据:

4.1 `del data_name['index_name']`, 删除索引为 index_name 的数据

Series 里的表现为很普通的删除对应索引的数据 (毕竟不是表结构)

DataFrame 里面应用的话，就是只能用来删除列，不能用来删除行

4.2 `data_name.drop(['index_1','index_2',....,'index_n'],inplace = False,axis = 0)`

各参数说明：

`['index_1','index_2',....,'index_n']`: 删除哪些 index, 可以一次性写多个

`inplace = False`:【省略可】默认 False，写 True 会将更改应用到原数据

`axis = 0`:【省略可】默认 axis = 0【行 index】，写 1 为【列 index】

对 Series 使用：单纯的删除某个索引的数据，不用指定 axis

对 DataFrame 使用：注意需要指定 axis 来说明想要删除的是行还是列

5.1.1 基本操作

[50]:
```
data = pd.Series(data = [10,11,12],index = ['a','b','c'])
#生成一个 Series 结构的数据
data
```

[50]:
```
a    10
b    11
c    12
dtype: int64
```

[51]:
```
data[0]
#找第 n 个值
```

[51]:
```
10
```

[52]:
```
data[0:2]
#切片
```

[52]:
```
a    10
b    11
dtype: int64
```

[54]:
```
mask = [True,False,True]
data[mask]
#用 bool 值的数组来取出我们想要的值
#原理为：把 mask 这个 bool 数组穿传进去，对应位置为 true 的值才会被取出来
```

[54]:
```
a    10
c    12
dtype: int64
```

[55]:
```
data.loc['b']
#用‘行索引’的手法找值，找名为 b 的行的值
```

[55]:
```
11
```

[58]:
```
data.iloc[1]
#用‘行索引’的手法找值，第 1 行的值（计数的开始是第 0 行）
```

[58]:
```
11
```

[68]:
```
data1 = data.copy()
data1['a'] = 100
print(data)
print(data1)
```

```
#更改数据值，但是运用 data_name.copy() 去避免原数据值被改变
```

```
a    10
b    11
c    12
dtype: int64
a    100
b     11
c     12
dtype: int64
```

[74]:
```
data.index = ['c','d','e']
data
#更改 index
```

[74]:
```
c    10
d    11
e    12
dtype: int64
```

5.1.2 改数据值

对于 Series 结构

[54]:
```
series1 = pd.Series(
    data = [11,22,32,43,54],
    index = ['1st','2nd','3rd','4th','5th']
)
series1
#生成一个 series1 用于演示
```

[54]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[58]:
```
series1['1st']=500
#把索引为 1st 的数据值改为 500
series1
```

[58]:
```
1st    500
2nd     22
3rd     32
4th     43
5th     54
dtype: int64
```

[60]:
```
series1.loc['1st']=600
#把索引为 1st 的数据值改为 600
series1
```

[60]:
```
1st    600
2nd     22
3rd     32
4th     43
5th     54
dtype: int64
```

[61]:
```
series1.iloc[0]=700
#把第 0 个数据值改为 700
series1
```

[61]:
```
1st    700
2nd     22
3rd     32
4th     43
5th     54
dtype: int64
```

对于 Data_Frame 结构

[62]:
```
data1 = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=['a','b','c'])
data1
#创造一个数据用来做演示
```

[62]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[65]:
```
data1['a'] = [2,5,8]
#把索引为 a 的【列】改为 2, 5, 8
data1
```

[65]:

	a	b	c
0	2	2	3
1	5	5	6
2	8	8	9

[66]:
```
data1['a'] = data1['c']
#把索引为 a 的【列】改的和索引为 c 的【列】一样
data1
```

[66]:

	a	b	c
0	3	2	3
1	6	5	6
2	9	8	9

[68]:
```
data1.loc[:,'a'] = [1,3,5]
#把索引为 a 的【列】改为 1,3,5
data1
```

[68]:

	a	b	c
0	1	2	3
1	3	5	6
2	5	8	9

[69]:
```
data1.iloc[:,0] = [2,4,6]
#把第 0【列】改为 2,4,6
data1
```

[69]:

	a	b	c
0	2	2	3
1	4	5	6

```
2  6  8  9
```

[70]:
```
data1.loc[0] = [1,3,5]
#把索引为 0 的【行】改为 1,3,5
data1
```

[70]:
```
   a  b  c
0  1  3  5
1  4  5  6
2  6  8  9
```

[71]:
```
data1.iloc[1] = [2,5,8]
#把第 1【行】改为 2,5,8
data1
```

[71]:
```
   a  b  c
0  1  3  5
1  2  5  8
2  6  8  9
```

[77]:
```
data1.iloc[1,1] = 100
#把第 1【行】第 1【列】的元素改为 100
data1
```

[77]:
```
   a    b  c
0  1    3  5
1  2  100  8
2  6    8  9
```

5.1.3 批量修改数据值

[70]:
```
print(data1)
print(data1.replace(to_replace = 100,value = 300,inplace = False))
#将数据值为 100 的全改为 300，但是不去影响原数据
```

原数据为:

```
a    100
b     11
```

```
c     12
dtype: int64
```

被更改数据为:

```
a    300
b     11
c     12
dtype: int64
```

[72]:
```
print(data1.replace(to_replace = 100,value = 300,inplace = True))
#将数据值为 100 的全改为 300, 并将修改的内容去影响原数据
print(data1)
```

数据 data 为:

```
a    300
b     11
c     12
dtype: int64
```

原数据为:

```
a    300
b     11
c     12
dtype: int64
```

编译成 latex 的时候把上一条的:【数据 data 为】这几个字改成:【被更改数据为:】

5.1.4 批量修改索引

[2]:
```
df = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=['a','b','c'])
df
#创造一个数据用来做演示
```

[2]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[80]:
```
df.rename({0:11})
#更改索引,【行名】
```

[80]:
```
    a  b  c
11  1  2  3
1   4  5  6
2   7  8  9
```

[81]:
```
df.rename({0:11},axis=0)
#更改索引,【行名】
```

[81]:
```
    a  b  c
11  1  2  3
1   4  5  6
2   7  8  9
```

[82]:
```
df.rename({'a':'aa'},axis=1)
#更改索引,【列名】
```

[82]:
```
   aa  b  c
0   1  2  3
1   4  5  6
2   7  8  9
```

[83]:
```
df.rename({'a':'aa'},axis='columns')
#更改索引,【列名】
```

[83]:
```
   aa  b  c
0   1  2  3
1   4  5  6
2   7  8  9
```

[84]:
```
df.rename(index={0:11},columns={"a":"aa"})
#index と columns を使って別々に指定してみる
```

[84]:
```
    aa  b  c
11   1  2  3
1    4  5  6
2    7  8  9
```

5.1.5 增添/插入数据

对于 Series 结构

[52]:
```
series1 = pd.Series(
    data = [11,22,32,43,54],
    index = ['1st','2nd','3rd','4th','5th']
)
series1
#生成一个 series1 用于演示
```

[52]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[53]:
```
series1.loc['6th'] = 66
series1
```

[53]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
6th    66
dtype: int64
```

对于 Data_Frame 结构

[81]:
```
data1 = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=['a','b','c'])
data1
#生成两个 DataFrame 结构用于演示
```

[81]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[42]:
```
data2 = pd.DataFrame([[11,22,33],[44,55,66],[77,88,99]], columns=['d','e','f'])
data2
```

[42]:

	d	e	f
0	11	22	33
1	44	55	66
2	77	88	99

[82]:
```
data1.loc[5] = [10,11,12]
#增加一【行】，索引写为 5
data1
```

[82]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9
5	10	11	12

[83]:
```
data1.loc[:,'d'] = [4,7,10,13]
#增加一【列】，索引写为 d
data1
```

[83]:

	a	b	c	d
0	1	2	3	4
1	4	5	6	7
2	7	8	9	10
5	10	11	12	13

[86]:
```
data1.loc[:,'e'] = np.random.randint(0,10,4)
#增加一【列】，索引写为 e，列的值为用 numpy 随机生成的整数
data1
```

[86]:

	a	b	c	d	e
0	1	2	3	4	0
1	4	5	6	7	1
2	7	8	9	10	0
5	10	11	12	13	9

[87]:
```
data1['f'] = np.random.normal(0,1,4)
#增加一【列】，索引写为 f，列的值为用 numpy 随机生成的服从标准正态分布的数
data1
```

[87]:

	a	b	c	d	e	f
0	1	2	3	4	0	0.230914
1	4	5	6	7	1	0.431042
2	7	8	9	10	0	0.023902
5	10	11	12	13	9	-0.739259

5.1.6 拼接

对于 Series 结构

[3]:
```
series1 = pd.Series(
    data = [11,22,32,43,54],
    index = ['1st','2nd','3rd','4th','5th']
)
series1
#生成一个 series1 用于演示
```

[3]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[4]:
```
series2 = pd.Series(
    data = [45,89,15,15,17],
    index = ['6th','7th','8th','9th','10th']
)
series2
#生成一个 series2 用于演示
```

[4]:
```
6th    45
7th    89
8th    15
9th    15
```

```
10th    17
dtype: int64
```

[8]:
```
series1.append(series2)
#往 series1 里面增加 series2，给拼到一起
```

[8]:
```
1st     11
2nd     22
3rd     32
4th     43
5th     54
6th     45
7th     89
8th     15
9th     15
10th    17
dtype: int64
```

[10]:
```
series1.append(series2,ignore_index = True)
#拼接以后去掉老的索引，换成默认的数字索引
#默认是 False
```

[10]:
```
0    11
1    22
2    32
3    43
4    54
5    45
6    89
7    15
8    15
9    17
dtype: int64
```

对于 Data_Frame 结构

[88]:
```
data1 = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=['a','b','c'])
data1
```

```
# 生成两个 DataFrame 结构用于演示
```

[88]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[89]:
```
data2 = pd.DataFrame([[11,22,33],[44,55,66],[77,88,99]], columns=['d','e','f'])
data2
```

[89]:

	d	e	f
0	11	22	33
1	44	55	66
2	77	88	99

[28]:
```
data1.append(data2)
# 拼接两个 DataFrame
```

[28]:

	a	b	c	d	e	f
0	1.0	2.0	3.0	NaN	NaN	NaN
1	4.0	5.0	6.0	NaN	NaN	NaN
2	7.0	8.0	9.0	NaN	NaN	NaN
0	NaN	NaN	NaN	11.0	22.0	33.0
1	NaN	NaN	NaN	44.0	55.0	66.0
2	NaN	NaN	NaN	77.0	88.0	99.0

[19]:
```
data1.append(data2).iloc[0:3,3:]
# 取出的右上角的 3*3 的块，可以看出来他的逻辑，
# 虽然是拼接了，但是因为行数行列数都增加了，所以会产生一些值没有经过定义
# 因此写为 NaN，但是这个 NaN 也是一个‘值’，可以被查询出来，不是摆设
```

[19]:

	d	e	f
0	NaN	NaN	NaN
1	NaN	NaN	NaN
2	NaN	NaN	NaN

对于 pd.concat 命令

拼行去一行一行的拼时，如果两个表的列名完全一样，就会简单的上下去一行一行的拼接，不一样

就会形成交叉错位式的插入【拼列时同理】

[92]:
```
data3 = data1
pd.concat([data1,data3])
#直接竖着拼接在一起（一行一行的拼接）
#axis参数省略时默认为 axis = 0，指定为在行的维度进行操作
```

[92]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9
0	1	2	3
1	4	5	6
2	7	8	9

[93]:
```
pd.concat([data1,data3],axis = 1)
#一列一列的拼接（axis = 1指定为在列的维度进行操作）
```

[93]:

	a	b	c	a	b	c
0	1	2	3	1	2	3
1	4	5	6	4	5	6
2	7	8	9	7	8	9

[94]:
```
pd.concat([data1,data2])
#列名不一样情况下的【一行一行的拼接】
```

[94]:

	a	b	c	d	e	f
0	1.0	2.0	3.0	NaN	NaN	NaN
1	4.0	5.0	6.0	NaN	NaN	NaN
2	7.0	8.0	9.0	NaN	NaN	NaN
0	NaN	NaN	NaN	11.0	22.0	33.0
1	NaN	NaN	NaN	44.0	55.0	66.0
2	NaN	NaN	NaN	77.0	88.0	99.0

[95]:
```
pd.concat([data1,data2],axis = 1)
#行名一样情况下的【一列一列的拼接】
```

[95]:

	a	b	c	d	e	f
0	1	2	3	11	22	33

```
1  4  5  6  44  55  66
2  7  8  9  77  88  99
```

5.1.7 删除操作

对于 Series 结构

[7]:
```
series1
#看一下原数据
```

[7]:
```
1st    11
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[11]:
```
del series1['1st']
#删掉以 1st 为索引的数据
#注意!! 这个命令会更改原数据!!!
#最好用赋值或者是.copy()的操作去做
series1
```

[11]:
```
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[17]:
```
series1.drop(['2nd','3rd'],inplace = False)
print(series1.drop(['2nd','3rd'],inplace = False))
print(series1)
#设定 inplace = False 后，这个命令不会更改原数组
#可以不写 inplace = ，默认是 False
```

```
4th    43
5th    54
dtype: int64
```

原数据为：

```
2nd    22
3rd    32
4th    43
5th    54
dtype: int64
```

[18]:
```
series1.drop(['2nd','3rd'],inplace = True)
#写 inplace = True 后就对更改原数据了
series1
```

[18]:
```
4th    43
5th    54
dtype: int64
```

对于 Data_Frame 结构

[19]:
```
data1
#看一下原数据
```

[19]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[22]:
```
del data1['a']
#删除 a 列的数据
data1
```

[22]:

	b	c
0	2	3
1	5	6
2	8	9

[25]:
```
del data1[0]
#不可用 del 命令删除某一行的数据，会报错
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

```
~/opt/anaconda3/lib/python3.8/site-packages/pandas/core/indexes/base.py in␣
↪get_loc(self, key, method, tolerance)
   3079             try:
-> 3080                 return self._engine.get_loc(casted_key)
   3081             except KeyError as err:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


KeyError: 0
```

[34]:
```
data1.drop(['a','c'],inplace = False,axis = 1)
#删除 a,c 列
print(data1.drop(['a','c'],inplace = False,axis = 1))
print(data1)
#设定 inplace = False 后，这个命令不会更改原数组
#可以不写 inplace = ，默认是 False
#这里用了 axis = 1 来指定的 index 为【列】的索引（不写的话默认 axis = 0，也就是行）
```

更改后的数据:

	b
0	2
1	5
2	8

原数据

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

[35]:
```
data1.drop(['a','c'],inplace = True,axis = 1)
data1
#写 inplace = True 后就对更改原数据了
```

[35]:

	b
0	2

```
1  5
2  8
```

[38]:
```
data1.drop([1,2])
#删除【行】
```

[38]:
```
   a  b  c
0  1  2  3
```

5.2 Merge 操作

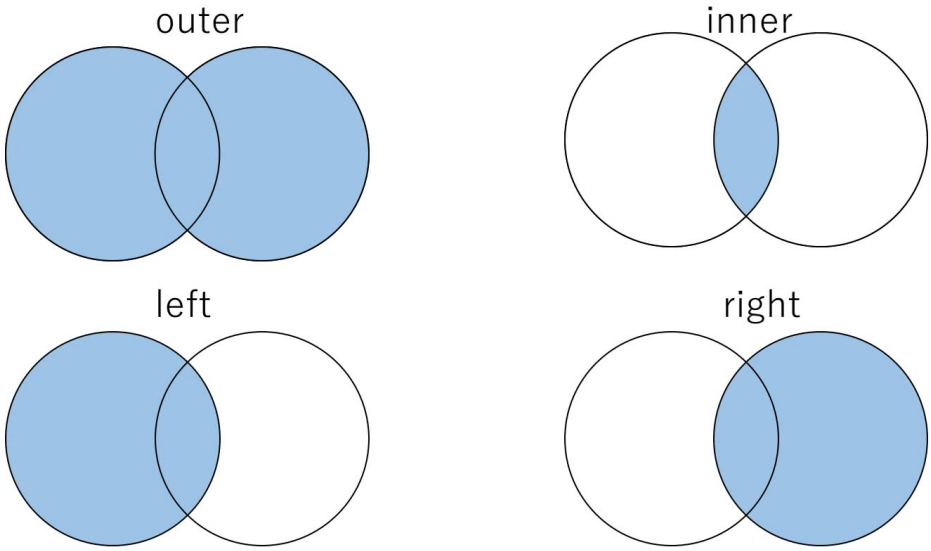
等同于 Relational Database 的 JOIN, 把两个表结合在一起

pd.merge(left_DataFrame,right_DataFrame,how='inner', on=None, left_on=None, right_on=N
参数解读:
left_DataFrame: 结合时放在【左边】的表的名称
right_DataFrame: 结合时放在【右边】的表的名称
how = 'xx':【可省略】默认为 **inner**, xx 参数可写: **left** 以左表为基准, **right** 以右表为基准, **inner** 内部结合（取交集）, **outer** 外部结合（取并集）
on = xxx:【可省略】默认为 **None**, 以 xxx 列为新合并出来的表的主 key（xxx 列必须为左右两表里同时存在的列）
left_on=xxx:【可省略】默认为 **None**, 以【左表】的 xxx 列为新合并出来的表的主 key
right_on=xxx:【可省略】默认为 **None**, 以【右表】的 xxx 列为新合并出来的表的主 key
left_index = True/False:【可省略】默认为 **False**, 合并出来的新表直接使用【左表】的主 key
right_index = True/False:【可省略】默认为 **False**, 合并出来的新表直接使用【右表】的主 key
sort = True/False:【可省略】默认为 **False**, 合并后是否对新表根据主 key 进行 sort 操作（从小到大进行排序）
suffixes=('_x', '_y'):【可省略】默认为 **'_x', '_y'**, 如果合并后非主 key 的列名发生重复的话, 指定一下如何重命名
indicator = True/False:【可省略】默认为 **False**, 合并后添加一个 _merge 列来显示各行在合并时执行的是什么操作（left? right?）
validate=xxx:【可省略】默认为 **None**, 查看一下合并后的主 key 与表内元素的对应关系, **'one_to_one'** 一对一, **'one_to_many'** 一对多, **'many_to_one'** 多对一, **'many_to_many'** 多对多

详见: click this

結合の仕方の違い
merge 関数で使われる引数 how のオプションについて解説します。
SQL を扱ったことのある人には INNER JOIN や LEFT JOIN など結合時に使用する場面が多いので馴染みのあるワードですが、Pandas でも 2 つのデータを結合するとき、結合の基準となるキーをどの範囲まで使うかを指定することができます。
例えば、inner を使用すると両方のデータに存在するキーのみで構成されます。outer ではどちらか片方に存在するキーも使われます。
結合される DataFrame は左側にあると考えられているので、left を使用すると結合される側の DataFrame のキーが全て使われます。right を使用すると結合する側の DataFrame のキーが全て

使われます。
これを図に表すと以下のようになります。



[2]:
```
left = pd.DataFrame({'key':['K0','K1','K2','K3'],
                     'A':['A0','A1','A2','A3'],
                     'B':['B0','B1','B2','B3']})

right = pd.DataFrame({'key':['K0','K1','K2','K3'],
                      'C':['C0','C1','C2','C3'],
                      'D':['D0','D1','D2','D3']})
#创建两个 DataFrame 结构来做演示
```

[3]:
```
#right 的结构与 left 有着极高的相似度，因此可以用上面学的替换命令去写
right = left.copy()
for i in range(0,4) :
    right.replace(to_replace = 'A%d'%(i),value = 'C%d'%(i),inplace = True)
    right.replace(to_replace = 'B%d'%(i),value = 'D%d'%(i),inplace = True)
    #用 i 带入进 Ai 以及 Ci 循环指定 A1 到 A3，替换为 C1 到 C3，B 与 D 也是同理
right.rename(columns = {'B':'D'},inplace = True)
right.rename(columns = {'A':'C'},inplace = True)
```

[81]:
```
print(left)
print(right)
#展示一下两个表
```

```
  key   A   B
0  K0  A0  B0
1  K1  A1  B1
2  K2  A2  B2
3  K3  A3  B3

  key   C   D
0  K0  C0  D0
1  K1  C1  D1
2  K2  C2  D2
3  K3  C3  D3
```

[87]:
```
pd.merge(left,right)
```

[87]:
```
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

[88]:
```
pd.merge(left,right,on = 'key')
#合并表，以 key1 为主 key
```

[88]:
```
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

[90]:
```
left.rename(columns = {'key':'key1'},inplace = True)
left['key2'] = ('K0','K1','K2','K3')
right.rename(columns = {'key':'key1'},inplace = True)
right['key2'] = ('K0','K1','K2','K3')
#操作两个表给 key 改成 key1，然后增加 key2 这个列
```

[91]:
```
print(left)
print(right)
#展示一下两个表
```

```
  key1   A   B key2
0   K0  A0  B0   K0
1   K1  A1  B1   K1
2   K2  A2  B2   K2
3   K3  A3  B3   K3

  key1   C   D key2
0   K0  C0  D0   K0
1   K1  C1  D1   K1
2   K2  C2  D2   K2
3   K3  C3  D3   K3
```

[92]:
```
pd.merge(left,right,on = 'key1')
#以 key1 为主 key 合并，但是因为两个表里都有‘key2’这个名字的列
#所以合并后会自动给两个列的 key2 重命名，加后缀
```

[92]:
```
  key1   A   B key2_x   C   D key2_y
0   K0  A0  B0      K0  C0  D0      K0
1   K1  A1  B1      K1  C1  D1      K1
2   K2  A2  B2      K2  C2  D2      K2
3   K3  A3  B3      K3  C3  D3      K3
```

[93]:
```
pd.merge(left,right,on = ['key1','key2'])
#以 key1 加上 key2 两个列作为主 key 来合并
#这次因为 key1 和 key2 在两个表里都存在，且值相等，所以就直接能这么合并
#不会出现上次那样的自动重命名的情况
```

[93]:
```
  key1   A   B key2   C   D
0   K0  A0  B0   K0  C0  D0
1   K1  A1  B1   K1  C1  D1
2   K2  A2  B2   K2  C2  D2
3   K3  A3  B3   K3  C3  D3
```

[103]:
```
right.iloc[3,3] = 'K8'
#稍微改一下 right 表，使得 left 表和 right 表的 key2 列的值有区别
print(right)
print(pd.merge(left,right,on = ['key1','key2']))
#我们会发现，因为两个表的 key2 的第 3 行的值（从第 0 行开始）不一样
```

```
#同时 key2 又被指定为主 key，所以只合并值一样的第 0,1,2 行，第 3 行直接被 drop 掉了
#只能 merge 主 key 一样的部分，不一样的会被过滤掉
#等于求了个交集
```

```
  key1   C   D key2
0   K0  C0  D0   K0
1   K1  C1  D1   K1
2   K2  C2  D2   K2
3   K3  C3  D3   K8
```

```
  key1   A   B key2   C   D
0   K0  A0  B0   K0  C0  D0
1   K1  A1  B1   K1  C1  D1
2   K2  A2  B2   K2  C2  D2
```

[104]:
```
pd.merge(left,right,on = ['key1','key2'],how = 'outer')
#定义一下 how 参数，写为 outer 就可以保留主 key 不一样的部分，留为 NaN
#就是合并一个交集出来了
```

[104]:
```
  key1    A    B key2    C    D
0   K0   A0   B0   K0   C0   D0
1   K1   A1   B1   K1   C1   D1
2   K2   A2   B2   K2   C2   D2
3   K3   A3   B3   K3  NaN  NaN
4   K3  NaN  NaN   K8   C3   D3
```

[105]:
```
pd.merge(left,right,on = ['key1','key2'],how = 'outer',indicator = True)
#加上 indicator 参数可以查看合并执行的是什么操作
#看 _merge 列
#left_only 意为只有 left 表里有值
```

[105]:
```
  key1    A    B key2    C    D      _merge
0   K0   A0   B0   K0   C0   D0        both
1   K1   A1   B1   K1   C1   D1        both
2   K2   A2   B2   K2   C2   D2        both
3   K3   A3   B3   K3  NaN  NaN   left_only
4   K3  NaN  NaN   K8   C3   D3  right_only
```

[106]:
```
pd.merge(left,right,on = ['key1','key2'],how = 'left')
#how参数还可以指定以哪个表为基准去合并，去处理主 key 的矛盾问题时优先谁
#以左表为基准，左表的主 key 里有但是右表里没有的会在相应位置写生 NaN
```

[106]:

	key1	A	B	key2	C	D
0	K0	A0	B0	K0	C0	D0
1	K1	A1	B1	K1	C1	D1
2	K2	A2	B2	K2	C2	D2
3	K3	A3	B3	K3	NaN	NaN

[107]:
```
pd.merge(left,right,on = ['key1','key2'],how = 'right')
#以右表为基准
```

[107]:

	key1	A	B	key2	C	D
0	K0	A0	B0	K0	C0	D0
1	K1	A1	B1	K1	C1	D1
2	K2	A2	B2	K2	C2	D2
3	K3	NaN	NaN	K8	C3	D3

5.3 全局的 display（显示, 打印）设置

1. **pd.get_option(参数)，看一下一些默认设置**
2. **pd.set_option(参数, 数值)，设定变更全局的 display 设置**

参数可写：
'display.max_rows'：默认设置打印多少行
'display.max_columns'：默认设置打印多少列
'display.max_colwidth'：默认设置打印的数据里的单独一个值的长度（字符串长度）
'display.precision'：默认设置打印的数据值的精度（四舍五入到小数点后几位）

数值可写：int 格式的数字就行

具体参考：官方文稿 | 解释的很具体的第三方文稿

[2]:
```
pd.get_option('display.max_rows')
#看一下默认设置打印多少行
```

[2]:
```
60
```

[7]:
```
pd.set_option('display.max_rows',10)
#为了节省空间，设置成默认打印 10 行
```

[11]:
```
pd.Series(index = range(0,100),dtype = 'float64')
#设置一个 100 行的 series，但是并不会被全打印下来
#中间有省略号，只会打印出来 10 行
```

[11]:
```
0    NaN
1    NaN
2    NaN
3    NaN
4    NaN
     ..
95   NaN
96   NaN
97   NaN
98   NaN
99   NaN
Length: 100, dtype: float64
```

[12]:
```
pd.get_option('display.max_columns')
#看一下默认设置打印多少列
```

[12]: 20

[13]:
```
pd.set_option('display.max_columns',5)
#为了节省空间，设置成默认打印 5 列
```

[14]:
```
pd.DataFrame(columns = range(0,30),index = range(0,100))
#设置一个 30 列 100 行的 DataFrame，但是并不会被全打印下来
#中间有省略号，只会打印出来 10 行，5 列
```

[14]:

	0	1	…	28	29
0	NaN	NaN	…	NaN	NaN
1	NaN	NaN	…	NaN	NaN
2	NaN	NaN	…	NaN	NaN
3	NaN	NaN	…	NaN	NaN
4	NaN	NaN	…	NaN	NaN
..	…	…	…	…	…
95	NaN	NaN	…	NaN	NaN
96	NaN	NaN	…	NaN	NaN
97	NaN	NaN	…	NaN	NaN
98	NaN	NaN	…	NaN	NaN
99	NaN	NaN	…	NaN	NaN

[100 rows x 30 columns]

[15]:
```
pd.get_option('display.max_colwidth')
#看一下默认设置打印的数据里的单独一个值的长度（字符串长度）
```

[15]: 50

[16]:
```
pd.set_option('display.max_colwidth',10)
#为了节省空间，设置成默认打印长度为 10
```

[18]:
```
pd.Series(index = ['a'],data = ['x'*80])
#设置一个 Series，只有一行，但是数据有 80 的长度
#但是不会被全打印出来
```

[18]:
```
a    xxxxxx…
dtype: object
```

[19]:
```
pd.get_option('display.precision')
#看一下默认设置打印的数据值的精度（四舍五入到小数点后几位）
```

[19]:
```
6
```

[20]:
```
pd.set_option('display.precision',2)
#设置成四舍五入到小数点后 2 位
```

[22]:
```
pd.Series(index = ['x'],data = [2.1576164])
#原值是 2.1576164，但是设置了四舍五入到小数点后 2 位，所以那个值显示成了 2.16
```

[22]:
```
x    2.16
dtype: float64
```

5.4 pivot 操作（数据透视表）

数据透视表（pivot table, ピボットテーブル）简单来说就是把明细表进行分类汇总的过程，你可以按照不同的组合方式进行数据计算

1. 用已有的表作为原材料生成新的透视表用于分析具体事例

pd.DataFrame_name.pivot(index=None,columns=None,values=None)

(前面的 pd. 可以省略)

各参数说明：具体参考 (click)

index = 'xxx': 省略可, 设置透视表中的的主 key 为 DataFrame_name 表中的哪一列，列中的元素的种类会成为新表的【行名】

columns = 'xxx': 省略可，设置生成新的分析中，成为主 key 的那一列与 DataFrame_name 表中的哪一列一起用呢，列中的元素的种类会成为新表的【列名】

values = 'xxx': 省略可，设置在新生成的分析中哪一列的元素来作为数据值

2. 用已有的表作为原材料生成新的透视表用于分析具体事例 (跟 1 效果差不多的另一种写法):

DataFrame_name.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')

各参数说明：具体参考 (click)

index = 'xxx': 省略可, 设置透视表中的的主 key 为 DataFrame_name 表中的哪一列，列中的元素的种类会成为新表的【行名】

columns = 'xxx': 省略可，设置生成新的分析中，成为主 key 的那一列与 DataFrame_name 表中的哪一列一起用呢，列中的元素的种类会成为新表的【列名】

values = 'xxx': 省略可，设置在新生成的分析中哪一列的元素来作为数据值

aggfunc='xxx': 省略可, 默认为 'mean', データを集計する手法を指定する, 还可以设置最大值最小值等，相当于 Numpy 笔记中的 7.2 统计量的运算中的都可以写

fill_value = 'xxx': 省略可，默认为 'None', 欠損値を補完する値を指定する

margins = True/False: 省略可，默认为 False，True の時列ごとの合計と行ごとの合計を表示する

dropna = True/False: 省略可，默认为 False，全てのデータが NaN 値である列データを集計に含まないようにする

margins_name = 'xxx': 省略可，默认为 'All'，小計を表示する行と列のラベル名を指定

3. 用已有的表作为原材料生成新的透视表用于分析具体事例 (跟 1 和 2 效果差不多的另一种写法):

pd.pivot_table(DataFrame_name,values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')

各参数说明：具体参考 (click)

DataFrame_name：做为原材料的那个表的名字
index = 'xxx'：省略可，设置透视表中的的主 key 为 DataFrame_name 表中的哪一列，列中的元素的种类会成为新表的【行名】
columns = 'xxx'：省略可，设置生成新的分析中，成为主 key 的那一列与 DataFrame_name 表中的哪一列一起用呢，列中的元素的种类会成为新表的【列名】
values = 'xxx'：省略可，设置在新生成的分析中哪一列的元素来作为数据值
aggfunc='xxx'：省略可，默认为 'mean'，データを集計する手法を指定する，还可以设置最大值最小值等，相当于 Numpy 笔记中的 7.2 统计量的运算中的都可以写
fill_value = 'xxx'：省略可，默认为 'None'，欠損値を補完する値を指定する
margins = True/False：省略可，默认为 False，True の時列ごとの合計と行ごとの合計を表示する
dropna = True/False：省略可，默认为 False，全てのデータが NaN 値である列データを集計に含まないようにする
margins_name = 'xxx'：省略可，默认为 'All'，小計を表示する行と列のラベル名を指定

4. 对某个表的某两个列提出来做成透视表，对里面的元素进行计数统计分布：
pd.crosstab(index , columns, rownames=None, colnames=None, aggfunc=None, dropna=True, normalize=False)
各参数说明：具体参考 (click)
index = DataFrame_name['xx']：省略可，设置透视表中的的主 key 为 DataFrame_name 表中的 xx 列，列中的元素的种类会成为新表的【行名】
columns = DataFrame_name['xx']：省略可，设置生成新的分析中，成为主 key 的那一列与 DataFrame_name 表中的 xx 一起用，列中的元素的种类会成为新表的【列名】
rownames = 'xxx'：省略可，行ラベルを新たに指定する際に使う
colnames = 'xxx'：省略可，列ラベルを新たに指定する際に使う
dropna = True/False：省略可，默认为 False，全てのデータが NaN 値である列データを集計に含まないようにする
normalize = True/False：省略可，默认为 False，是否对所有值进行归一化（正規化）

[33]:
```
import numpy as np
example = pd.DataFrame({'Month':["January","January","January","January",
                                 "February","February","February","February",
                                 "March","March","March","March"],
                        'Category':
↪["Transportation","Grocery","Household","Entertainment",
                                 ␣
↪"Transportation","Grocery","Household","Entertainment",
```

```
                                    ␣
↪"Transportation","Grocery","Household","Entertainment"],
                        })
example['Amount'] = np.random.randint(50,300,12)
example
#生成一个 DataFrame 的例子用来操作
#是一个家庭账本，一条一条的
```

[33]:

	Month	Category	Amount
0	January	Transportation	118
1	January	Grocery	195
2	January	Household	260
3	January	Entertainment	138
4	February	Transportation	131
5	February	Grocery	146
6	February	Household	141
7	February	Entertainment	155
8	March	Transportation	199
9	March	Grocery	118
10	March	Household	98
11	March	Entertainment	270

[34]:
```
#如果我们想统计一下在各个品类（大概哪一类事上）上每月花了多少钱
#那么，行就设置成各个品类，那么列就是月份了，数据值就是上面的金额了
example_pivot = example.pivot(index = 'Category',
                              columns = 'Month',values = 'Amount')


example_pivot
```

[34]:

Month	February	January	March
Category			
Entertainment	155	138	270
Grocery	146	195	118
Household	141	260	98
Transportation	131	118	199

[35]:
```
example_pivot.sum(axis = 1)
#统计各个品类的合计花费
```

[35]:
```
Category
Entertainment     563
Grocery           459
Household         499
Transportation    448
dtype: int64
```

[36]:
```
example_pivot.sum(axis = 0)
#统计各个月的合计花费
```

[36]:
```
Month
February    573
January     711
March       685
dtype: int64
```

[38]:
```
#请出老嘉宾 titanic.csv 来做接下来的演示
test = pd.read_csv('titanic.csv')
test.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen...	male	22.0	1	0	A/5 21171	7.25	NaN	S
1	2	1	1	Cumings, Mrs. Jo...	female	38.0	1	0	PC 17599	71.28	C85	C
2	3	1	3	Heikkinen, Miss....	female	26.0	0	0	STON/O2. 3101282	7.92	NaN	S
3	4	1	1	Futrelle, Mrs. J...	female	35.0	1	0	113803	53.10	C123	S
4	5	0	3	Allen, Mr. Willi...	male	35.0	0	0	373450	8.05	NaN	S

[40]:
```
test.pivot_table(index = 'Sex',columns = 'Pclass',values = 'Fare')
#想看一看两个性别在各个等级的船舱上大概花了多少钱（一个期待值）
#因为是离散性的变量，所以期待值就是平均值
#.pivot_table 这个命令求出来的默认就是平均值
```

[40]:
```
Pclass        1      2      3
Sex
female   106.13  21.97  16.12
male      67.23  19.74  12.66
```

[41]:
```
test.pivot_table(index = 'Sex',columns = 'Pclass',values = 'Fare',
                 aggfunc = 'max')
#这次是看看不同性别在各个等级的船舱上的花费最大值，加了 aggfunc = 'max'参数
```

[41]:
```
Pclass       1     2      3
Sex
female  512.33  65.0  69.55
male    512.33  73.5  69.55
```

[42]:
```
test.pivot_table(index = 'Sex',columns = 'Pclass',values = 'Fare',
                 aggfunc = 'count')
#看看不同性别在不同等级船舱上的计数分布
```

[42]:
```
Pclass    1    2    3
Sex
female   94   76  144
male    122  108  347
```

[45]:
```
pd.crosstab(index = test['Sex'],columns = test['Pclass'])
#跟上面一样的效果，通过.crosstab 这个计数命令实现的
```

[45]:
```
Pclass    1    2    3
Sex
female   94   76  144
male    122  108  347
```

[46]:
```
test.pivot_table(index = 'Pclass',columns = 'Sex',values = 'Survived',
                 aggfunc = 'mean')
#统计不用船舱等级中两个性别的存活的平均值（期望值）
#因为存活这个数据是一个二元量（只有 0 和 1），所以平均值是等于默认被归一化（正規化）了的
#所以这个期待值可以解释为存活率
```

[46]:
```
Sex     female  male
Pclass
1         0.97  0.37
2         0.92  0.16
3         0.50  0.14
```

[47]:
```
test['未成年'] = test['Age'] <= 18
#添加一列来判断是否为未成年
test.pivot_table(index = '未成年',columns = 'Sex',values = 'Survived',
                 aggfunc = 'mean')
#统计未成年人的不同性别的获救率
#未成年 False 意味着已经成年
```

[47]:

Sex	female	male
未成年		
False	0.76	0.17
True	0.68	0.34

5.5 时间操作

0. 如果不考虑 Pandas 的话，Python 中的时间处理组件
import datetime as dt 使用这个组件
dt.datetime(year = 2017, month = 11,day=24,hour = 10,minute = 30)
生成一个时间结构
记得用 ***print()*** 打印出来才像是时间的结构，不然直接输出的话不像是时间结构

1. Pandas 中生成时间结构（Timestamp）的工具
pd.Timestamp('yyyy-mm-dd')
'yyyy-mm-dd' 就是比如 **'2022-03-21'**

2. 另一种工具，不仅可以生成，还可以把 Series 中的元素转化为 datetime64 结构:
pd.to_datetime(xxx)
生成一个时间结构的元素：**pd.to_datetime('yyyy-mm-dd')**
转化 Series 中的元素：**pd.to_datetime(Series_name)**
* 但是要是用于转化的话 Series 中的元素不能是简单的数字，必须也是类似于时间结构的写法，而且并不是 int64 类型转化为 datetime64 这样，而是从 object 转化的，因为 Series 中元素是类似于 '2018-06-28' 这样的东西，不是一个单纯的数字，具体看下面的例子

3. 时间数据的切片:
data_name[('yyyy-mm-dd'):('yyyy-mm-dd')], 从 A 日期到 B 日期
data_name['yyyy'], 单独调取某一年的
data_name[bool_array], 用布尔数组取出特定条件的数据，比如所有年份中一月的数据

4. 数据的重新采样（集約）:具体参考 (click)
data_name.resample(参数). 条件命令
参数可写：**'D'** 天，**'H'** 小时等等，具体的去参考上面贴出来的链接
条件命令可写：省略可，默认为 **mean()**，还可写 **max()** 等等，具体的去链接

5. 抽取最后 xx 条数据: data_name.tail(xx)
6. 抽取每一天的几点到几点的数据: data_name.between_time('HH:mm','HH:mm')

5.5.1 基础生成操作

[3]:
```
import datetime as dt
testtime = dt.datetime(year = 2017, month = 11,
                       day=24,hour = 10,minute = 30)
testtime
```

```
#这样打印出来看起来不像是时间
```

[3]: datetime.datetime(2017, 11, 24, 10, 30)

[4]:
```
print(testtime)
#这样打印一下就是时间结构了
```

2017-11-24 10:30:00

[5]:
```
testtime2 = pd.Timestamp('2017-11-24')
testtime2
#用 pandas 来创建时间结构
```

[5]: Timestamp('2017-11-24 00:00:00')

[6]:
```
testtime2.month
#输出时间结构中的月份
```

[6]: 11

[7]:
```
testtime2.day
```

[7]: 24

[8]:
```
testtime2 + pd.Timedelta('5 days')
#可以直接做加减这样的计算
```

[8]: Timestamp('2017-11-29 00:00:00')

[9]:
```
pd.to_datetime('2017-11-24')
#用另一个命令也可以
```

[9]: Timestamp('2017-11-24 00:00:00')

[10]:
```
pd.to_datetime('24/11/2017')
#另一种日期中的分隔符，但是这种的话时间的写法就不一样了
```

[10]: Timestamp('2017-11-24 00:00:00')

[12]:
```
s = pd.Series(['2017-11-29 00:00:00','2017-11-24 00:00:00',
               '2017-11-25 00:00:00'])
s
```

```
#用时间来构造 Series 结构的数据
#构造出来的注意，是 object 结构，不是 datetime64 结构
```

[12]:
```
0    2017-11-29 00:00:00
1    2017-11-24 00:00:00
2    2017-11-25 00:00:00
dtype: object
```

[14]:
```
ts = pd.to_datetime(s)
ts
#用 pd.to_datetime 命令还可以转换数据类型
#注意下面的 dtype 的值
```

[14]:
```
0   2017-11-29
1   2017-11-24
2   2017-11-25
dtype: datetime64[ns]
```

[15]:
```
ts.dt.day
#写全了是：ts.datetime.day
#抽出这个 Series 的日的数据
```

[15]:
```
0    29
1    24
2    25
dtype: int64
```

[17]:
```
ts.dt.weekday
#抽出星期几
#当然，还可以抽出小时啊，分钟等等
```

[17]:
```
0    2
1    4
2    5
dtype: int64
```

[18]:
```
pd.Series(pd.date_range(start = '2022-3-21',periods = 10,freq = '12H'))
#使用 pd.date_range 来构建时间规律性的多个时间数据，时间结构版的 range 命令
```

[18]:
```
0   2022-03-21 00:00:00
1   2022-03-21 12:00:00
2   2022-03-22 00:00:00
3   2022-03-22 12:00:00
4   2022-03-23 00:00:00
5   2022-03-23 12:00:00
6   2022-03-24 00:00:00
7   2022-03-24 12:00:00
8   2022-03-25 00:00:00
9   2022-03-25 12:00:00
dtype: datetime64[ns]
```

[22]:
```
pd.date_range(start = '2022-3-21 15:00:01',periods = 10,freq = '3D',
              normalize = False)
#不进行 normalize，开始时间就会从 15:00:01 开始
```

[22]:
```
DatetimeIndex(['2022-03-21 15:00:01', '2022-03-24 15:00:01',
               '2022-03-27 15:00:01', '2022-03-30 15:00:01',
               '2022-04-02 15:00:01', '2022-04-05 15:00:01',
               '2022-04-08 15:00:01', '2022-04-11 15:00:01',
               '2022-04-14 15:00:01', '2022-04-17 15:00:01'],
              dtype='datetime64[ns]', freq='3D')
```

[23]:
```
pd.date_range(start = '2022-3-21 15:00:01',periods = 10,freq = '3D',
              normalize = True)
#设定了了 normalize 后，开始时间就会对齐到 00:00:00
```

[23]:
```
DatetimeIndex(['2022-03-21', '2022-03-24', '2022-03-27', '2022-03-30',
               '2022-04-02', '2022-04-05', '2022-04-08', '2022-04-11',
               '2022-04-14', '2022-04-17'],
              dtype='datetime64[ns]', freq='3D')
```

5.5.2 查询与切片

[26]:
```
data = pd.read_csv('./data/flowdata.csv')
data.head()
#读一个 csv 进来做示例
```

[26]:

	Time	L06_347	LS06_347	LS06_348
0	2009-01-01 00:00:00	0.137417	0.097500	0.016833
1	2009-01-01 03:00:00	0.131250	0.088833	0.016417
2	2009-01-01 06:00:00	0.113500	0.091250	0.016750
3	2009-01-01 09:00:00	0.135750	0.091500	0.016250
4	2009-01-01 12:00:00	0.140917	0.096167	0.017000

[27]:
```
data['Time'] = pd.to_datetime(data['Time'])
data = data.set_index('Time')
data
#把 Time 列的数据转换为 datetime64 结构
#然后设置 Time 列为主 key
```

[27]:

Time	L06_347	LS06_347	LS06_348
2009-01-01 00:00:00	0.137417	0.097500	0.016833
2009-01-01 03:00:00	0.131250	0.088833	0.016417
2009-01-01 06:00:00	0.113500	0.091250	0.016750
2009-01-01 09:00:00	0.135750	0.091500	0.016250
2009-01-01 12:00:00	0.140917	0.096167	0.017000
…	…	…	…
2013-01-01 12:00:00	1.710000	1.710000	0.129583
2013-01-01 15:00:00	1.420000	1.420000	0.096333
2013-01-01 18:00:00	1.178583	1.178583	0.083083
2013-01-01 21:00:00	0.898250	0.898250	0.077167
2013-01-02 00:00:00	0.860000	0.860000	0.075000

[11697 rows x 3 columns]

[29]:
```
data = pd.read_csv('./data/flowdata.csv',
                   index_col = 'Time',parse_dates = True)
data
#直接这样写就跟上面会一样了
```

[29]:

Time	L06_347	LS06_347	LS06_348
2009-01-01 00:00:00	0.137417	0.097500	0.016833
2009-01-01 03:00:00	0.131250	0.088833	0.016417

```
2009-01-01 06:00:00  0.113500  0.091250  0.016750
2009-01-01 09:00:00  0.135750  0.091500  0.016250
2009-01-01 12:00:00  0.140917  0.096167  0.017000
…                         …         …         …
2013-01-01 12:00:00  1.710000  1.710000  0.129583
2013-01-01 15:00:00  1.420000  1.420000  0.096333
2013-01-01 18:00:00  1.178583  1.178583  0.083083
2013-01-01 21:00:00  0.898250  0.898250  0.077167
2013-01-02 00:00:00  0.860000  0.860000  0.075000

[11697 rows x 3 columns]
```

[33]:
```
data[pd.Timestamp('2012-01-10 12:00'):pd.Timestamp('2012-01-11 9:00')]
#因为使用了 datatime64 格式的数据作为了主 key
#所以在作切片的时候也要用 datatime64 格式的数据去查询
#所以使用了 pd.Timestamp() 来做转化
```

[33]:
```
                     L06_347  LS06_347  LS06_348
Time
2012-01-10 12:00:00  0.220417  0.227000  0.025000
2012-01-10 15:00:00  0.203583  0.224750  0.023917
2012-01-10 18:00:00  0.203833  0.222167  0.024250
2012-01-10 21:00:00  0.201750  0.219333  0.024167
2012-01-11 00:00:00  0.195250  0.215833  0.022750
2012-01-11 03:00:00  0.192917  0.213917  0.023667
2012-01-11 06:00:00  0.178750  0.210333  0.021667
2012-01-11 09:00:00  0.190833  0.208833  0.022000
```

[34]:
```
data[('2012-01-10 12:00'):('2012-01-11 9:00')]
#不写 pd.Timestamp() 其实也会自己做转化
#结果是跟上面一样
```

[34]:
```
                     L06_347  LS06_347  LS06_348
Time
2012-01-10 12:00:00  0.220417  0.227000  0.025000
2012-01-10 15:00:00  0.203583  0.224750  0.023917
2012-01-10 18:00:00  0.203833  0.222167  0.024250
```

```
2012-01-10 21:00:00  0.201750  0.219333  0.024167
2012-01-11 00:00:00  0.195250  0.215833  0.022750
2012-01-11 03:00:00  0.192917  0.213917  0.023667
2012-01-11 06:00:00  0.178750  0.210333  0.021667
2012-01-11 09:00:00  0.190833  0.208833  0.022000
```

[35]:
```
data.tail(10)
#取最后 10 个数据
```

[35]:
```
                     L06_347  LS06_347  LS06_348
Time
2012-12-31 21:00:00  0.846500  0.846500  0.170167
2013-01-01 00:00:00  1.688333  1.688333  0.207333
2013-01-01 03:00:00  2.693333  2.693333  0.201500
2013-01-01 06:00:00  2.220833  2.220833  0.166917
2013-01-01 09:00:00  2.055000  2.055000  0.175667
2013-01-01 12:00:00  1.710000  1.710000  0.129583
2013-01-01 15:00:00  1.420000  1.420000  0.096333
2013-01-01 18:00:00  1.178583  1.178583  0.083083
2013-01-01 21:00:00  0.898250  0.898250  0.077167
2013-01-02 00:00:00  0.860000  0.860000  0.075000
```

[36]:
```
data['2013']
#或者写成 data.loc['2013']
#取 2013 年的数据，直接写个‘2013’就行了
```

[36]:
```
                     L06_347  LS06_347  LS06_348
Time
2013-01-01 00:00:00  1.688333  1.688333  0.207333
2013-01-01 03:00:00  2.693333  2.693333  0.201500
2013-01-01 06:00:00  2.220833  2.220833  0.166917
2013-01-01 09:00:00  2.055000  2.055000  0.175667
2013-01-01 12:00:00  1.710000  1.710000  0.129583
2013-01-01 15:00:00  1.420000  1.420000  0.096333
2013-01-01 18:00:00  1.178583  1.178583  0.083083
2013-01-01 21:00:00  0.898250  0.898250  0.077167
2013-01-02 00:00:00  0.860000  0.860000  0.075000
```

[37]:
```
data['2012-12':'2013-01']
#取 2012 年 12 月份到 2013 年 1 月
```

[37]:

Time	L06_347	LS06_347	LS06_348
2012-12-01 00:00:00	0.083500	0.083500	0.009583
2012-12-01 03:00:00	0.082167	0.082167	0.010000
2012-12-01 06:00:00	0.081167	0.081167	0.010000
2012-12-01 09:00:00	0.081000	0.081000	0.010333
2012-12-01 12:00:00	0.081750	0.081750	0.010667
…	…	…	…
2013-01-01 12:00:00	1.710000	1.710000	0.129583
2013-01-01 15:00:00	1.420000	1.420000	0.096333
2013-01-01 18:00:00	1.178583	1.178583	0.083083
2013-01-01 21:00:00	0.898250	0.898250	0.077167
2013-01-02 00:00:00	0.860000	0.860000	0.075000

[257 rows x 3 columns]

[38]:
```
data[data.index.month == 1]
#使用布尔值的数组来取数据。意为取出所有年中的一月份的数据
#data.index.month == 1 这个判定条件出来是一个布尔数组
```

[38]:

Time	L06_347	LS06_347	LS06_348
2009-01-01 00:00:00	0.137417	0.097500	0.016833
2009-01-01 03:00:00	0.131250	0.088833	0.016417
2009-01-01 06:00:00	0.113500	0.091250	0.016750
2009-01-01 09:00:00	0.135750	0.091500	0.016250
2009-01-01 12:00:00	0.140917	0.096167	0.017000
…	…	…	…
2013-01-01 12:00:00	1.710000	1.710000	0.129583
2013-01-01 15:00:00	1.420000	1.420000	0.096333
2013-01-01 18:00:00	1.178583	1.178583	0.083083
2013-01-01 21:00:00	0.898250	0.898250	0.077167
2013-01-02 00:00:00	0.860000	0.860000	0.075000

[1001 rows x 3 columns]

[39]:
```
data[(data.index.hour >8) & (data.index.hour <= 12)]
#取每一天的 8 点到 12 点之间的数据
```

[39]:

Time	L06_347	LS06_347	LS06_348
2009-01-01 09:00:00	0.135750	0.091500	0.016250
2009-01-01 12:00:00	0.140917	0.096167	0.017000
2009-01-02 09:00:00	0.141917	0.097083	0.016417
2009-01-02 12:00:00	0.147833	0.101917	0.016417
2009-01-03 09:00:00	0.124583	0.084417	0.015833
…	…	…	…
2012-12-30 12:00:00	1.465000	1.465000	0.086833
2012-12-31 09:00:00	0.682750	0.682750	0.066583
2012-12-31 12:00:00	0.651250	0.651250	0.063833
2013-01-01 09:00:00	2.055000	2.055000	0.175667
2013-01-01 12:00:00	1.710000	1.710000	0.129583

[2924 rows x 3 columns]

[40]:
```
data.between_time('8:00','12:00')
#跟上面结果一样的另一种写法
```

[40]:

Time	L06_347	LS06_347	LS06_348
2009-01-01 09:00:00	0.135750	0.091500	0.016250
2009-01-01 12:00:00	0.140917	0.096167	0.017000
2009-01-02 09:00:00	0.141917	0.097083	0.016417
2009-01-02 12:00:00	0.147833	0.101917	0.016417
2009-01-03 09:00:00	0.124583	0.084417	0.015833
…	…	…	…
2012-12-30 12:00:00	1.465000	1.465000	0.086833
2012-12-31 09:00:00	0.682750	0.682750	0.066583
2012-12-31 12:00:00	0.651250	0.651250	0.063833
2013-01-01 09:00:00	2.055000	2.055000	0.175667
2013-01-01 12:00:00	1.710000	1.710000	0.129583

```
[2924 rows x 3 columns]
```

[41]:
```
data.head()
#打印一下原表，方便与下面做对比
```

[41]:
```
                     L06_347  LS06_347  LS06_348
Time
2009-01-01 00:00:00  0.137417  0.097500  0.016833
2009-01-01 03:00:00  0.131250  0.088833  0.016417
2009-01-01 06:00:00  0.113500  0.091250  0.016750
2009-01-01 09:00:00  0.135750  0.091500  0.016250
2009-01-01 12:00:00  0.140917  0.096167  0.017000
```

[42]:
```
data.resample('D').mean().head()
#以天为单位进行重新采样，重新划分表
#原来是带着时间的，一天当中有很多时间段的，现在新的表里都没有了
#原来一天当中那么多时间段的以求平均的形式，统合成了一天一天的数据
```

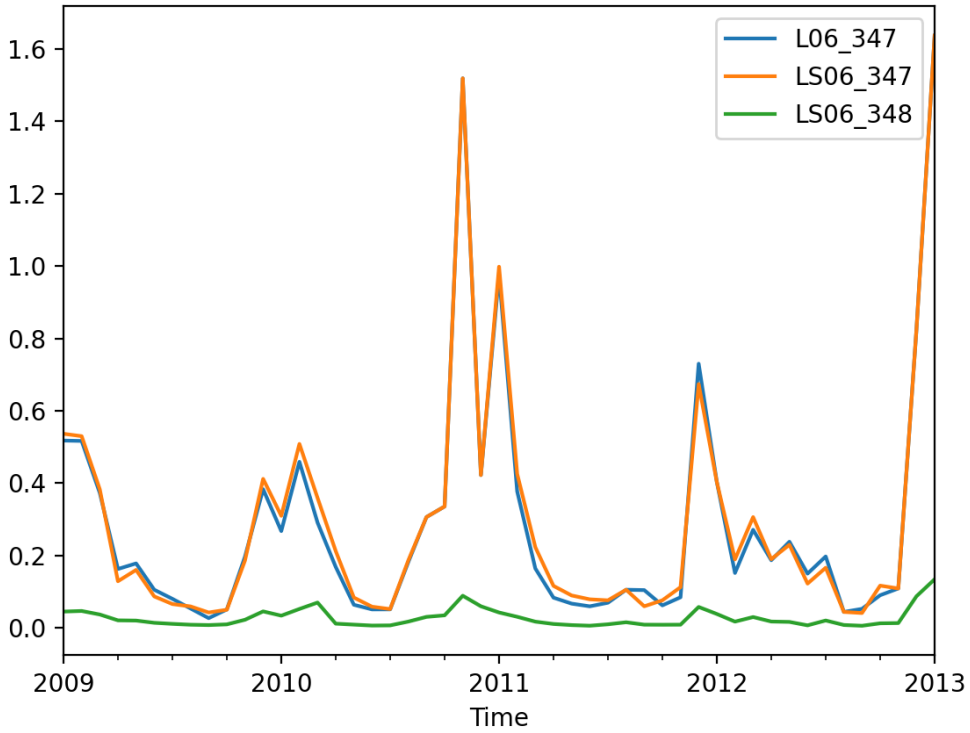
[42]:
```
             L06_347  LS06_347  LS06_348
Time
2009-01-01  0.125010  0.092281  0.016635
2009-01-02  0.124146  0.095781  0.016406
2009-01-03  0.113562  0.085542  0.016094
2009-01-04  0.140198  0.102708  0.017323
2009-01-05  0.128812  0.104490  0.018167
```

[43]:
```
data.resample('3D').max().head()
#执行每三天一次，取三天内最大值来集約する的重新采样
```

[43]:
```
             L06_347  LS06_347  LS06_348
Time
2009-01-01  0.147833  0.101917  0.017583
2009-01-04  0.161500  0.115167  0.021583
2009-01-07  0.139333  0.095750  0.018083
2009-01-10  0.193500  0.147000  0.018750
2009-01-13  1.158750  1.620000  0.104917
```

[45]:
```
%matplotlib notebook
#加载插件进行一个画图的展示
data.resample('M').mean().plot()
#画一下图，可视化，后面会详细写，这里只是展示一下可以这么用
#当然，上面其他的命令也可以这么用
```

```
<IPython.core.display.Javascript object>
```



[45]:
```
<AxesSubplot:xlabel='Time'>
```

5.6 表内的【排序, 去重, 映射, 分组】, 以及有关缺省值的处理

1. 排序操作 (对 Series 和 DataFrame 均可使用)
- **Series_name.sort_values(axis=0,ascending=True,inplace=False, kind='quicksort', na_position='last')**
- **DataFrame_name.sort_values(by = 'xxx',axis=0,ascending=True, inplace=False, kind='quicksort',na_position='last')**

各参数说明: 具体参考 (click)
by = 'xxx' :(DataFrame のみ) 对某【行】或者某【列】进行操作
axis = xx : 省略可，默认 0，对【行】内 (0 或者是'index') 还是【列】内 (1 或者是'columns') 进行操作。【列】方向に指定できるのは DataFrame だけ
ascending = True/False : 省略可，默认 False, 是否按升序进行排列
inplace = True/False : 省略可，默认 False, 是否影响原数据
kind = 'quicksort' : 省略可，指定排序用的算法，默认 quicksort
na_position = 'last' : 省略可，缺省值 (NaN) 放在开头还是末尾，默认末尾

2. 去重（若有多个完全一样的行，就去掉多余的只留一个）:
DataFrame_name.drop_duplicates(subset=None, keep='first', inplace=False, ignore_index=False)
各参数说明: 具体参考 (click)
subset = 'xxx' : 仅以某列的值的种类为基准
keep='first'/'last'/False : 省略可，默认 False，保留重复的第一个/最后一个值
inplace = True/False : 省略可，默认 False, 是否影响原数据
ignore_index = True/False : 省略可, 默认 False, 是否重命名 index 到纯数字的从 0 开始的数列

3. 映射（跟 5.1.3 的批量替换数据值的感觉很像，但是映射可以用来生成一个新的列）:
DataFrame_name/Series_name.map(字典结构/对应关系)
各参数说明: 具体参考 (click)
对应关系: 把所有的 NY 写成 NewYork→ **'NY':'NewYork'** 这种设置一一对应关系，可以写多个，用逗号隔开
字典结构: 内含多个对应关系的的一个集合的结构，形如下面例子中In [17]的【foodmapping】

4. 分组（在 array 内部分成一个个区间，英语 bins):
pd.cut(array_name, bins, right=True, labels=None, include_lowest=False, duplicates='raise')
- **各参数说明**: 具体参考 (click)
array_name : 想要划分区间的 array
bins = xxx : 指定如何分组, 传进来的可以是 array 的名字，也可以是 [1,5,10] 这样直接写一个出

来

labels = xxx: 省略可，默认 None, 是否给分组后的数据打标签，传进来的可以是 array 的名字，也可以是 ['S','M','L'] 这样直接写一个出来

include_lowest = False: 省略可，默认 False, Whether the first interval should be left-inclusive or not.

duplicates = 'raise'/'drop': 省略可，默认'raise',If bin edges are not unique, raise ValueError or drop non-uniques.

- **一些可以配合着用的命令:**

bins_result.codes, 这个是把分组的情况结果做成一个数组, 可以用于运算

pd.value_counts(bins_result), 看看各个区间里分别有几个值

5. 有关缺省值的处理 (NaN,Null):

- **data.isnull()**, 看一下数据里哪里是缺省值, 结果里写 True 的地方是缺省值
- **data[x].isnull()**, 看一下数据里第 x【行】的哪里是缺省值
- **data.iloc[x].isnull()**, 看一下数据里第 x【列】的哪里是缺省值
- **data.isnull().any()**, 检查各【列】里是否"包含"缺省值, 注意：检查的是是否包含，而不是缺省值在哪里
- **data.isnull().any(axis = 1)**, 检查各【行】里是否"包含"缺省值, 注意：检查的是是否包含，而不是缺省值在哪里
- **data.fillna(xxx)**, 用 xxx 去填充缺省值, 可以借助 Numpy 里的命令用服从分布的随机数来填充

[4]:
```
import numpy as np
data = pd.DataFrame({'group':['a','a','a','b','b','b','c','c','c']})
data['data'] = np.random.randint(0,15,9)
data
# 生成一个 DataFrame 用于演示
```

[4]:

	group	data
0	a	14
1	a	2
2	a	0
3	b	0
4	b	3
5	b	11
6	c	10
7	c	13
8	c	12

[6]:
```
data.sort_values(by = ['group','data'],ascending = [False,True],
                 inplace = True)
data
#在 group 中按降序排列，然后排完的的 group 的小组内根据 data 的数字按升序排列
#同时命令结果会写入原 data 中
#ascending 是在问是否升序，写 False 是降序
```

[6]:

	group	data
6	c	10
8	c	12
7	c	13
3	b	0
4	b	3
5	b	11
2	a	0
1	a	2
0	a	14

[19]:
```
data2 = pd.DataFrame({'k1':['one']*3+['two']*4,
                      'k2': np.random.randint(1,5,7)})
data2
#生成一个新的数据用于演示
```

[19]:

	k1	k2
0	one	1
1	one	4
2	one	1
3	two	4
4	two	4
5	two	3
6	two	4

[20]:
```
data2.sort_values(by = 'k2')
#按照 k2 进行排序
```

[20]:

	k1	k2
0	one	1
2	one	1

```
5  two   3
1  one   4
3  two   4
4  two   4
6  two   4
```

[21]:
```
data2.drop_duplicates()
#去重
```

[21]:
```
    k1  k2
0  one   1
1  one   4
3  two   4
5  two   3
```

[39]:
```
data2.drop_duplicates(subset = 'k1')
#仅以 k1 为基准去重
```

[39]:
```
    k1  k2
0  one   1
3  two   4
```

[3]:
```
data3 = pd.DataFrame({'food':['A1','A2','A3',
                              'B1','B2','B3',
                              'C1','C2','C3'],
                      'values': np.random.randint(1,8,9)})
data3
#生成一个新的数据用于演示
```

[3]:
```
  food  values
0   A1       2
1   A2       4
2   A3       6
3   B1       4
4   B2       6
5   B3       7
6   C1       1
7   C2       1
```

```
8   C3      4
```

[4]:
```
foodmapping = {
    'A1':'A',
    'A2':'A',
    'A3':'A',
    'B1':'B',
    'B2':'B',
    'B3':'B',
    'C1':'C',
    'C2':'C',
    'C3':'C',
}
#创建一个字典结构数据用于传进函数里做映射（mapping）
data3['map'] = data3['food'].map(foodmapping)
#做映射，A1～A3都视为A大类，B/C也做同样处理，这样映射分组，map列作为映射结果的接收处
data3
```

[4]:
```
  food  values map
0   A1       2   A
1   A2       4   A
2   A3       6   A
3   B1       4   B
4   B2       6   B
5   B3       7   B
6   C1       1   C
7   C2       1   C
8   C3       4   C
```

[5]:
```
data4 = pd.DataFrame({'value1': np.random.randint(1,100,10),
                      'value2': np.random.randint(1,100,10)})
data4['ration'] = data4['value1']/data4['value2']
data4
#新建的一列的数据元素也可以是通过运算得来
```

[5]:

	value1	value2	ration
0	90	24	3.750000
1	72	5	14.400000
2	92	37	2.486486
3	2	66	0.030303
4	98	20	4.900000
5	4	48	0.083333
6	19	62	0.306452
7	42	38	1.105263
8	23	87	0.264368
9	67	6	11.166667

[7]:
```
Series = pd.Series(np.random.randint(1,15,9))
Series
#准备一个 Series 用来做演示
```

[7]:
```
0     5
1     8
2    11
3     5
4     5
5     3
6     2
7    10
8    14
dtype: int64
```

[10]:
```
Series.replace(to_replace = 5 ,value = np.nan ,inplace = True )
#把 Series 里面所有的 5 都换成缺省值，缺省值可以用 np.nan 来表示是生成
Series
```

[10]:
```
0     NaN
1     8.0
2    11.0
3     NaN
4     NaN
5     3.0
6     2.0
```

```
7    10.0
8    14.0
dtype: float64
```

[14]:
```
ages = np.random.randint(10,80,15)
#随机生成一个年龄的数组
bins = [10,30,50,80]
#做一个分组的划分，10-30，30-50,50-80三个分组
bins_result = pd.cut(ages,bins)
#把想要做分组的数据和如何分组的规定传进pd.cut()里面
print(ages)
print(bins_result)
#就可以给ages里面每一个数据做出分组
#从另一个角度讲，也是把ages的每一个数据根据bins的规则映射进了各个空间里
```

```
[47 14 20 38 37 27 19 45 41 12 39 73 54 70 27]

[(30, 50], (10, 30], (10, 30], (30, 50], (30, 50], …, (30, 50], (50, 80], (50,
80], (50, 80], (10, 30]]
Length: 15
Categories (3, interval[int64, right]): [(10, 30] < (30, 50] < (50, 80]]
```

[16]:
```
bins_result.codes
#这个是把分组的情况结果做成一个数组，0代表在(10, 30]，1是(30, 50]，2是(50, 80]
```

[16]:
```
array([1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 2, 2, 2, 0], dtype=int8)
```

[17]:
```
pd.value_counts(bins_result)
#看看各个区间里有几个值
```

[17]:
```
(10, 30]    6
(30, 50]    6
(50, 80]    3
dtype: int64
```

[18]:
```
pd.cut(ages,[10,30,50,80])
#这么写也是可以的，直接传进一个经过定义的数组的名字，或者是直接写一个数组都行
```

[18]:
```
[(30, 50], (10, 30], (10, 30], (30, 50], (30, 50], …, (30, 50], (50, 80], (50, 80], (50, 80], (10, 30]]
Length: 15
Categories (3, interval[int64, right]): [(10, 30] < (30, 50] < (50, 80]]
```

[21]:
```
group_names = ['Youth','Mille','Old']
#创建一个文字序列
result = pd.cut(ages,bins,labels = group_names)
#把上面的序列传进来，作为分组过后的标签
print(result)
print(pd.value_counts(result))
#可以看到分组的情况已经不是数字的区间了，而是指定的标签
```

```
['Mille', 'Youth', 'Youth', 'Mille', 'Mille', …, 'Mille', 'Old', 'Old', 'Old', 'Youth']
Length: 15
Categories (3, object): ['Youth' < 'Mille' < 'Old']

Youth    6
Mille    6
Old      3
dtype: int64
```

[22]:
```
data = pd.DataFrame([range(3),[0,np.nan,0],[0,0,np.nan],range(3)])
#生成一个四行三列的数据
data
```

[22]:

	0	1	2
0	0	1.0	2.0
1	0	NaN	0.0
2	0	0.0	NaN
3	0	1.0	2.0

[23]:
```
data.isnull()
#看一下数据里哪里是缺省值
#写 True 的是缺省值
```

[23]:
```
       0      1      2
0  False  False  False
1  False   True  False
2  False  False   True
3  False  False  False
```

[25]:
```
data[0].isnull()
#抽取出第一列看一看哪里是缺省值
```

[25]:
```
0    False
1    False
2    False
3    False
Name: 0, dtype: bool
```

[26]:
```
data.iloc[2].isnull()
#看一下第 2 行哪里是缺省值，从第 0 行开始
```

[26]:
```
0    False
1    False
2     True
Name: 2, dtype: bool
```

[27]:
```
data.isnull().any()
#检查各【列】里是否“包含”缺省值
#注意：检查的是是否包含，而不是缺省值在哪里
```

[27]:
```
0    False
1     True
2     True
dtype: bool
```

[28]:
```
data.isnull().any(axis = 1)
#检查各【行】里是否“包含”缺省值
#注意：检查的是是否包含，而不是缺省值在哪里
```

[28]:
```
0    False
1     True
2     True
```

```
3    False
dtype: bool
```

[29]:
```
data.fillna(3.14)
#用 3.14 去填充缺省值
```

[29]:
```
   0     1     2
0  0  1.00  2.00
1  0  3.14  0.00
2  0  0.00  3.14
3  0  1.00  2.00
```

[31]:
```
data.fillna(np.random.normal())
#用服从标准正态分布的随机数去填充缺省值
```

[31]:
```
   0         1         2
0  0  1.000000  2.000000
1  0  0.082548  0.000000
2  0  0.000000  0.082548
3  0  1.000000  2.000000
```

[32]:
```
data[data.isnull().any(axis=1)]
#把“检查哪些行里有缺省值”这个结果的 bool 数组传进 data 里
#这样就可以定位并筛选出有缺省值的【行】了
```

[32]:
```
   0    1    2
1  0  NaN  0.0
2  0  0.0  NaN
```

5.7 字符串操作

以下的命令是 pyhon 的，所以是针对所有 String 结构的，不仅仅是 Series

1. `Series.str`：把 Series 里的 obejct 的字母元素转化为 String 字符串结构

2. `Series.str.lower()`：把 Series 里的把所有的 String 转化为小写

3. `Series.str.upper()`：把 Series 里的把所有的 String 转化为大写

4. `Series.str.len()`：计算 Series 中 String 各项的长度

5. `Series.str.strip()`：去除 String 各项里的空格

6. `Series.str.lstrip()`：只去除 String 中字符左边的空格

7. `Series.str.rstrip()`：只去除 String 中字符左边的空格

8. `df.columns = df.columns.str.replace()`：修改列名，逻辑是先将列名转化为字符串，然后使用 replace() 命令，这样就达到了 rename() 的效果

9.

colorboxlightgraySeries.str.split('x',expand = True,n = 1): 将一个字段切分成多个，括号内写的内容是以什么为标志进行切割

- **`expand = True/False`**：省略可，默认 False，是否将切开的字符都分成单独的一列一列的
- **`n = x`**：省略可，默认 0，配合 expand 使用，将字符切 x 次（也就是切成 x+1 列），写 0 为切到能切的最大值

10. `Series.str.contains('xxxx')`：看每一项里是否包含 xxxx, 这里的逻辑是不是仅仅有 xxxx，而是只要包含了，就算 True。逻辑是.any( ) 不是.all( )

11. `Series.str.get_dummies(sep = '|')`：以 | 来切一下看看有几种可能性，都分别有什么字母，处于什么位置，当然 sep 参数可以指定任意字符，比如,//-* 等等，不是只有 | ，这个理解起来有点抽象，一定要看下面的例子去理解

[3]:
```
s = pd.Series(['A','b','B','beer','AGE',np.nan])
s
#生成一个 Series 结构用于演示
```

[3]:
```
0       A
1       b
2       B
3    beer
4     AGE
5     NaN
dtype: object
```

[4]:
```
s.str.lower()
#.str,把 Series 里的 obejct 转化为 String 字符串结构
#.lower 把所有的 String 转化为小写
```

[4]:
```
0       a
1       b
2       b
3    beer
4     age
5     NaN
dtype: object
```

[7]:
```
s.str.upper()
#.upper 把所有的 String 转化为大写
```

[7]:
```
0       A
1       B
2       B
3    BEER
4     AGE
5     NaN
dtype: object
```

[8]:
```
s.str.len()
#计算 Series 中 String 各项的长度
#.len 计算 String 长度
```

[8]:
```
0    1.0
1    1.0
2    1.0
3    4.0
4    3.0
5    NaN
dtype: float64
```

[10]:
```
index = pd.Index([' tang',' yu   ','di'])
index
#注意里面有空格
```

[10]:
```
Index([' tang', ' y u   ', 'di'], dtype='object')
```

[11]:
```
index.str.strip()
#去除空格
```

[11]:
```
Index(['tang', 'y u', 'di'], dtype='object')
```

[12]:
```
index.str.lstrip()
#只去除字符左边的空格
```

[12]:
```
Index(['tang', 'y u   ', 'di'], dtype='object')
```

[13]:
```
index.str.rstrip()
#只去除字符右边的空格
```

[13]:
```
Index([' tang', ' y u', 'di'], dtype='object')
```

[22]:
```
np.random.seed(10)
df = pd.DataFrame(np.random.randn(3,2),columns = ['A a','B b'],
                  index = range(3))
df
#生成一个 DataFrame 用于演示
#注意列名的 A a，有空格感觉怪怪的
```

[22]:

	A a	B b
0	1.331587	0.715279
1	-1.545400	-0.008384
2	0.621336	-0.720086

[21]:
```
df.columns = df.columns.str.replace(' ','_')
df
#虽然没有用到.rename()，但是这样也可以修改列名
```

[21]:

	A_a	B_b
0	1.331587	0.715279
1	-1.545400	-0.008384
2	0.621336	-0.720086

[23]:
```
df.rename(columns = {'A a':'A_a','B b':'B_b'})
#跟上面效果一样的 rename 写法
```

[23]:
```
        A_a       B_b
0  1.331587  0.715279
1 -1.545400 -0.008384
2  0.621336 -0.720086
```

[24]:
```
s = pd.Series(['a_b_C','c_d_e','f_g_h'])
s
#生成一个新的 Series 用于演示
```

[24]:
```
0    a_b_C
1    c_d_e
2    f_g_h
dtype: object
```

[25]:
```
s.str.split('_')
#将一个字段切分成多个，括号内写的内容是以什么为标志进行切割
#但是切割完了的数据是类似于 list 结构，虽然数据被切割了但是还是同属‘同一条’数据
```

[25]:
```
0    [a, b, C]
1    [c, d, e]
2    [f, g, h]
dtype: object
```

[26]:
```
s.str.split('_',expand = True)
#切成三列
```

[26]:
```
   0  1  2
0  a  b  C
1  c  d  e
2  f  g  h
```

[27]:
```
s.str.split('_',expand = True,n = 1)
#切成两列，逻辑是从左开始切一次，n=1
```

[27]:
```
   0    1
0  a  b_C
1  c  d_e
2  f  g_h
```

[28]:
```
s.str.split('_',expand = True,n = 0)
#切到最大值，逻辑是 n=0 时默认切到最后
```

[28]:

	0	1	2
0	a	b	C
1	c	d	e
2	f	g	h

[29]:
```
s.str.split('_',expand = True,n = 2)
#切 2 次
```

[29]:

	0	1	2
0	a	b	C
1	c	d	e
2	f	g	h

[30]:
```
s = pd.Series(['A','Ass','Afgew','Ager','Asda','Rdafas'])
s
#生成一个新的 Series 用于演示
```

[30]:
```
0         A
1       Ass
2     Afgew
3      Ager
4      Asda
5    Rdafas
dtype: object
```

[32]:
```
s.str.contains('Ass')
#看看每一项里是否包含 Ass
```

[32]:
```
0    False
1     True
2    False
3    False
4    False
5    False
dtype: bool
```

[33]:
```
s.str.contains('A')
#看看每一项里是否包含 A
#这里的逻辑是虽然前 5 项不是仅仅有 A，但是只要包含了，就算 True
#逻辑是.any() 不是.all()
```

[33]:
```
0     True
1     True
2     True
3     True
4     True
5    False
dtype: bool
```

[34]:
```
s = pd.Series(['a','a|b','a|c'])
s
#生成一个新的 Series 用于演示
```

[34]:
```
0      a
1    a|b
2    a|c
dtype: object
```

[35]:
```
s.str.get_dummies(sep = '|')
#以 | 来切一下看看有几种可能性，都分别有什么字母，处于什么位置
#比如第 0 行的 100，意思是有 a，a 处于第 1 个位置，没有 bc
#第 1 行的 110，有 a，b 分别处于最开始第一个和第二个位置，没有 c
```

[35]:

	a	b	c
0	1	0	0
1	1	1	0
2	1	0	1

6 索引进阶

1. **Series.isin(array)**：看一下 Series 中的各元素是否在 array 的集合范围中，然后把结果输出成一个 bool 的 array
2. **DataFrame.select_dtypes(include = 'int64')**：抽出指定的数据值的类型的列，详见
3. **DataFrame/Series.where()**：看看所有数据中哪些符合条件，然后仅‘正常’表示符合条件的数据值，详见
4. **DataFrame/Series.query(查找条件)**：抽出符合条件的行，クエリ,详见

[6]:
```
s = pd.Series(np.arange(5),index = np.arange(5)[::-1],dtype = 'int64')
s
```

[6]:
```
4    0
3    1
2    2
1    3
0    4
dtype: int64
```

[7]:
```
s.isin([1,3,4])
#看一下S的各项元素在不在指定数列当中
#这个输出结果相当于一个索引
```

[7]:
```
4    False
3     True
2    False
1     True
0     True
dtype: bool
```

[8]:
```
s[s.isin([1,3,4])]
#把这个输出的结果再带回s中就可以把在符合条件的元素给取出来
```

[8]:
```
3    1
1    3
0    4
dtype: int64
```

[19]:
```
s[s>2]
#把 s>2 的结果带进 s 里达到一个 where 的效果（Numpy 第 18 页）
```

[19]:
```
1    3
0    4
dtype: int64
```

[12]:
```
s2 = pd.Series(np.arange(6),index = pd.MultiIndex.
↪from_product([[0,1],['a','b','c']]))
s2
```

[12]:
```
0  a    0
   b    1
   c    2
1  a    3
   b    4
   c    5
dtype: int64
```

[16]:
```
s2.index.isin([(1,'a'),(0,'b')])
#之前的例子是扫描的数据的值，不是 index
#这里加了 s2.index 了就是扫描的 index 了
```

[16]:
```
array([False,  True, False,  True, False, False])
```

[17]:
```
s2.iloc[s2.index.isin([(1,'a'),(0,'b')])]
```

[17]:
```
0  b    1
1  a    3
dtype: int64
```

[21]:
```
dates = pd.date_range('20220416',periods = 8)
df = pd.DataFrame(np.random.randn(8,4),index = dates,columns =␣
↪['A','B','C','D'])
#np.random.randn(8,4) 服从标准正态的随机数，八行四列
df
```

[21]:
```
                   A         B         C         D
2022-04-16  0.227153  0.696273 -0.163297  0.166266
```

```
2022-04-17 -0.053414  0.883283  1.384194  0.778599
2022-04-18 -0.021729 -1.017212  0.014065 -0.772067
2022-04-19  0.214300  0.861759 -1.077567  0.428103
2022-04-20 -0.426067  1.792496 -0.298694 -1.404630
2022-04-21  0.410137  1.182143 -0.773018  1.042808
2022-04-22  1.646927 -0.037920  0.083796  0.620174
2022-04-23 -0.031129  0.461979 -0.396730 -1.948634
```

[28]:
```
df['E'] = (np.random.randint(0,100,8))
df.select_dtypes(include = 'int64')
#挑选符合条件的列，指定数据值的类型
```

[28]:
```
             E
2022-04-16  66
2022-04-17  92
2022-04-18  14
2022-04-19  96
2022-04-20  21
2022-04-21  89
2022-04-22  57
2022-04-23  91
```

[29]:
```
df.where(df<0)
#大于 0 的都被替换成了 NaN，符合条件的小于 0 的都原封不动
```

[29]:
```
                   A         B         C         D   E
2022-04-16       NaN       NaN -0.163297       NaN NaN
2022-04-17 -0.053414       NaN       NaN       NaN NaN
2022-04-18 -0.021729 -1.017212       NaN -0.772067 NaN
2022-04-19       NaN       NaN -1.077567       NaN NaN
2022-04-20 -0.426067       NaN -0.298694 -1.404630 NaN
2022-04-21       NaN       NaN -0.773018       NaN NaN
2022-04-22       NaN -0.037920       NaN       NaN NaN
2022-04-23 -0.031129       NaN -0.396730 -1.948634 NaN
```

[30]:
```
df.where(df<0,other = 'inconformity')
#要是觉得 NaN 太扎眼，就可以自定义不符合条件的怎么办
```

	A	B	C	D	E
2022-04-16	inconformity	inconformity	-0.163297	inconformity	inconformity
2022-04-17	-0.053414	inconformity	inconformity	inconformity	inconformity
2022-04-18	-0.021729	-1.017212	inconformity	-0.772067	inconformity
2022-04-19	inconformity	inconformity	-1.077567	inconformity	inconformity
2022-04-20	-0.426067	inconformity	-0.298694	-1.40463	inconformity
2022-04-21	inconformity	inconformity	-0.773018	inconformity	inconformity
2022-04-22	inconformity	-0.03792	inconformity	inconformity	inconformity
2022-04-23	-0.031129	inconformity	-0.39673	-1.948634	inconformity

[31]:
```
df.where(df<0,other = 00000)
#可以自定义成数字
```

[31]:
```
                   A         B         C         D  E
2022-04-16  0.000000  0.000000 -0.163297  0.000000  0
2022-04-17 -0.053414  0.000000  0.000000  0.000000  0
2022-04-18 -0.021729 -1.017212  0.000000 -0.772067  0
2022-04-19  0.000000  0.000000 -1.077567  0.000000  0
2022-04-20 -0.426067  0.000000 -0.298694 -1.404630  0
2022-04-21  0.000000  0.000000 -0.773018  0.000000  0
2022-04-22  0.000000 -0.037920  0.000000  0.000000  0
2022-04-23 -0.031129  0.000000 -0.396730 -1.948634  0
```

[38]:
```
df.where(df<0,other = np.random.rand())
#可以自定义函数
```

[38]:
```
                   A         B         C         D         E
2022-04-16  0.441954  0.441954 -0.163297  0.441954  0.441954
2022-04-17 -0.053414  0.441954  0.441954  0.441954  0.441954
2022-04-18 -0.021729 -1.017212  0.441954 -0.772067  0.441954
2022-04-19  0.441954  0.441954 -1.077567  0.441954  0.441954
2022-04-20 -0.426067  0.441954 -0.298694 -1.404630  0.441954
2022-04-21  0.441954  0.441954 -0.773018  0.441954  0.441954
2022-04-22  0.441954 -0.037920  0.441954  0.441954  0.441954
```

```
2022-04-23 -0.031129  0.441954 -0.396730 -1.948634  0.441954
```

[40]:
```
df = pd.DataFrame(np.random.rand(10,3),columns = list('abc'))
df
```

[40]:

	a	b	c
0	0.025962	0.574788	0.483700
1	0.691312	0.616487	0.137699
2	0.693775	0.536818	0.918854
3	0.206047	0.511861	0.601002
4	0.312647	0.140194	0.781439
5	0.723642	0.387837	0.991192
6	0.199025	0.819137	0.081670
7	0.727984	0.062002	0.884683
8	0.593585	0.577090	0.899433
9	0.855752	0.979678	0.823118

[41]:
```
df.query('(a<b)')
#抽出所有 a 的数值大于 b 的行
```

[41]:

	a	b	c
0	0.025962	0.574788	0.483700
3	0.206047	0.511861	0.601002
6	0.199025	0.819137	0.081670
9	0.855752	0.979678	0.823118

[43]:
```
df.query('(a<b) & (b>c)')
#追加条件
```

[43]:

	a	b	c
0	0.025962	0.574788	0.483700
6	0.199025	0.819137	0.081670
9	0.855752	0.979678	0.823118

[44]:
```
df.query('(a > 0.8) | (c < 0.1)')
#追加条件 | 为 OR 运算
```

[44]:

	a	b	c
6	0.199025	0.819137	0.081670
9	0.855752	0.979678	0.823118

7 Pandas 绘图

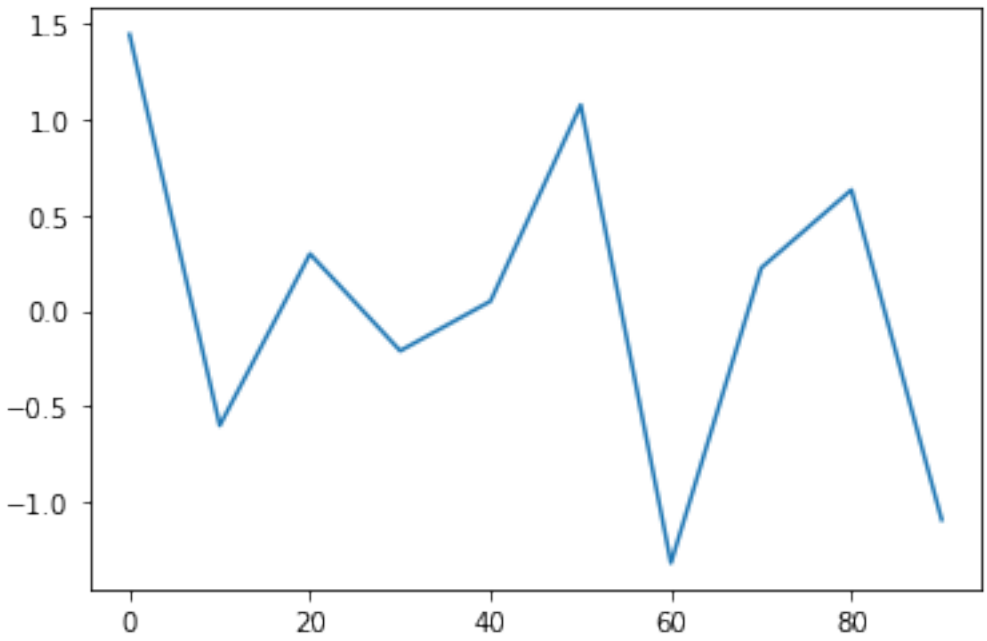
0. **%matplotlib inline**，为了在 jupyter notebook 中使用绘图工具 matplotlib，所以写的是%，不然就会跳出弹窗了，就不能把图留在笔记内了。当然，实际使用的时候写**import matplotlib.pyplot as plt**
1. 折线图:**Series/DataFrame.plot()**
2. 柱状图:**Series/DataFrame.plot(kind = 'bar')**
3. 横着的柱状图：**在上面的'bar'改成'barh'就行了**
4. 直方图：**Series/DataFrame.plot(kind = 'hist',bins = xx)**，为分成 xx 个区间，详见
5. 散点图 1：**pd.plotting.scatter_matrix(DataFrame,color = 'k',alpha = 0.3)**，把所有属性（列）都做一个两两的匹配然后画散点图出来（会有超级多）
6. 散点图 2：**DataFrame.plot.scatter('column1','column2')**，把名为 column1 column2 的两个列拿出来做个散点图，详见

使用大全详见:click this

[45]:
```
%matplotlib inline
#为了在 jupyter notebook 中使用绘图工具 matplotlib，所以写的是%，不然就会跳出弹窗了，
就不能把图留在笔记内了
#当然，实际使用的时候写 import matplotlib.pyplot as plt
```

[46]:
```
s = pd.Series(np.random.randn(10),index = np.arange(0,100,10))
s.plot()
#对 s 画折线图
```

[46]:
```
<AxesSubplot:>
```



[53]:
```
np.random.seed(10)
df = pd.DataFrame(np.random.randn(10,4),
                  index = np.arange(0,100,10),
                  columns = ['A','B','C','D'])
df
```

[53]:

	A	B	C	D
0	1.331587	0.715279	-1.545400	-0.008384
10	0.621336	-0.720086	0.265512	0.108549
20	0.004291	-0.174600	0.433026	1.203037
30	-0.965066	1.028274	0.228630	0.445138
40	-1.136602	0.135137	1.484537	-1.079805
50	-1.977728	-1.743372	0.266070	2.384967
60	1.123691	1.672622	0.099149	1.397996
70	-0.271248	0.613204	-0.267317	-0.549309
80	0.132708	-0.476142	1.308473	0.195013
90	0.400210	-0.337632	1.256472	-0.731970

[54]:
```
#给数值那里的函数加一个参数
np.random.seed(10)
df1 = pd.DataFrame(np.random.randn(10,4).cumsum(0),
                   index = np.arange(0,100,10),
                   columns = ['A','B','C','D'])
df1
```

[54]:

	A	B	C	D
0	1.331587	0.715279	-1.545400	-0.008384
10	1.952922	-0.004807	-1.279889	0.100165
20	1.957214	-0.179407	-0.846863	1.303202
30	0.992148	0.848867	-0.618232	1.748340
40	-0.144454	0.984004	0.866305	0.668535
50	-2.122182	-0.759368	1.132375	3.053502
60	-0.998491	0.913254	1.231524	4.451498
70	-1.269739	1.526458	0.964207	3.902189
80	-1.137031	1.050316	2.272680	4.097203
90	-0.736821	0.712684	3.529152	3.365233

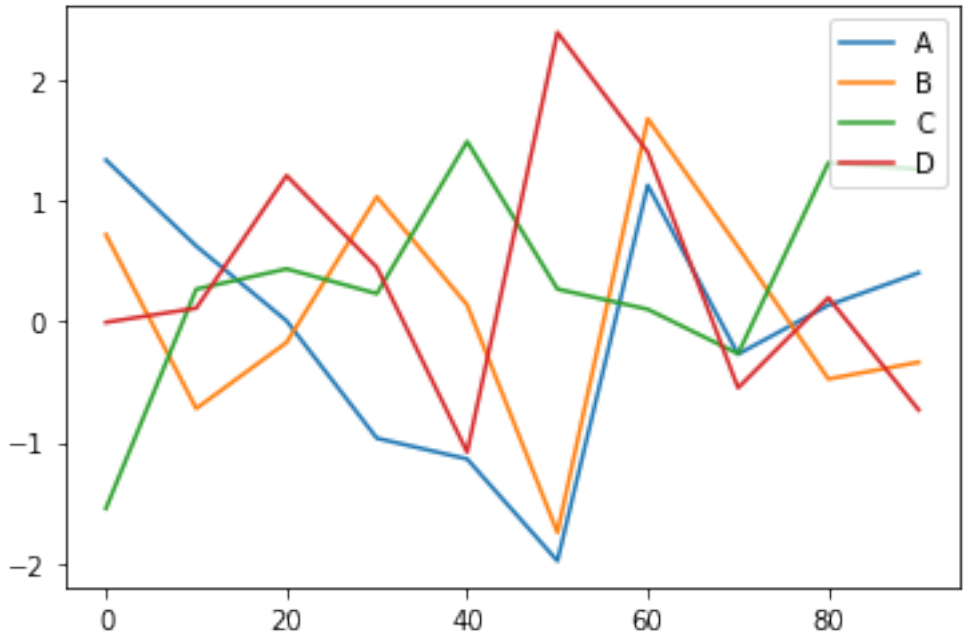
np.cumsum() 累积和。

求出到各个 index 位置的累积之和，然后替换原位置

详见：click this

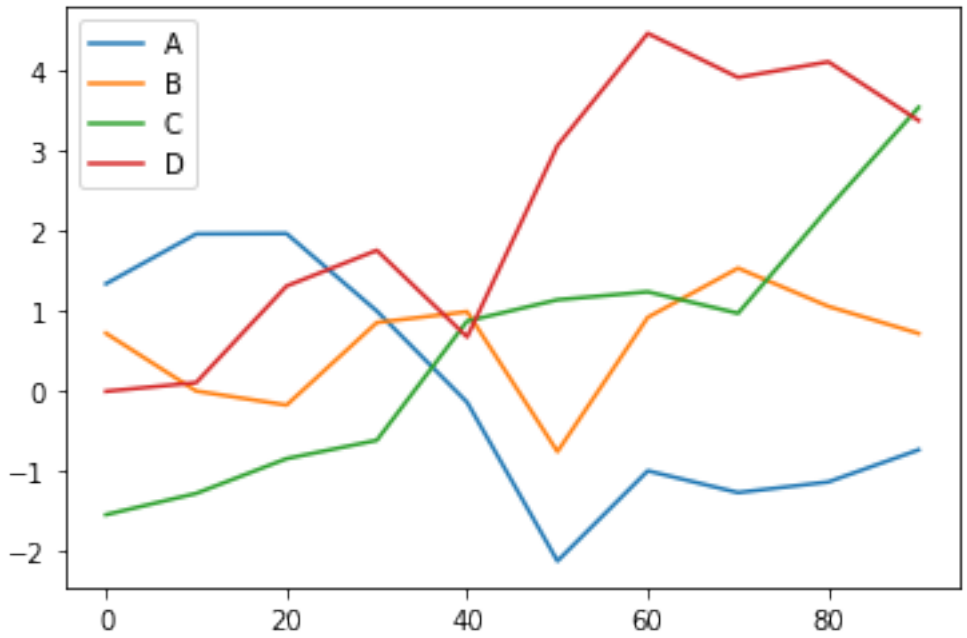
[55]:
```
df.plot()
```

[55]:
```
<AxesSubplot:>
```



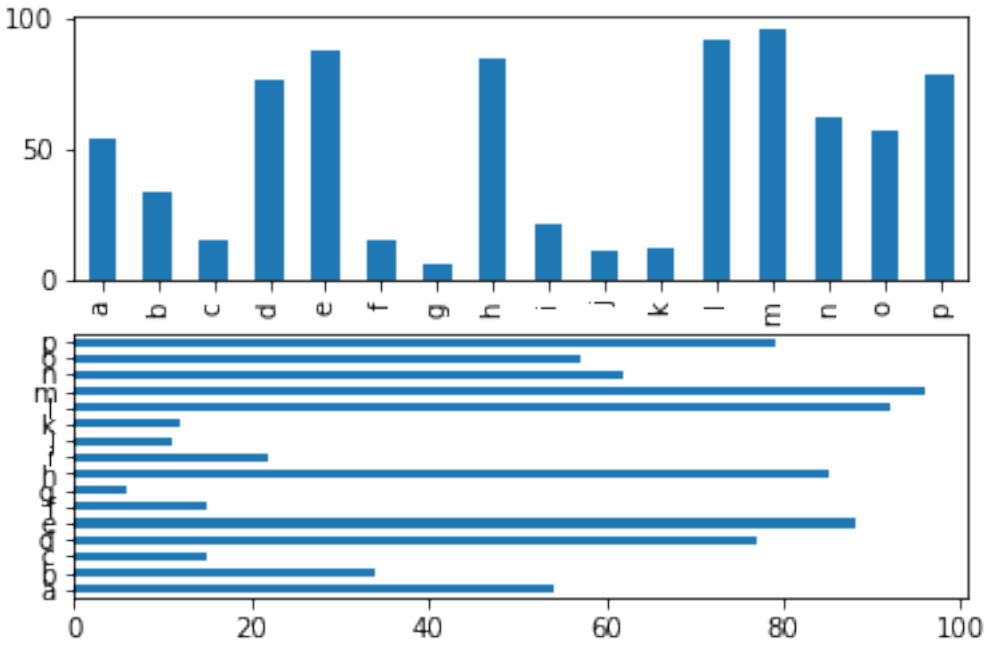
[56]:
```
df1.plot()
```

[56]: <AxesSubplot:>



[57]:
```
import matplotlib.pyplot as plt
#为了画出 subplot，也就是自图
#这里只是为了做个演示，具体的放到 matplotlib 那里面详细讲
fig,axes = plt.subplots(2,1)
data = pd.Series(np.random.randint(0,100,16),
                 index = list('abcdefghijklmnop'))
data.plot(ax = axes[0],kind = 'bar')
data.plot(ax = axes[1],kind = 'barh')
```

[57]: <AxesSubplot:>



[64]:
```
df = pd.DataFrame(np.random.rand(6,4),
                  index = ['one','two','three','four','five','six'],
                  columns = pd. Index (['A','B','C','D'],name = 'Genus'))
df
```

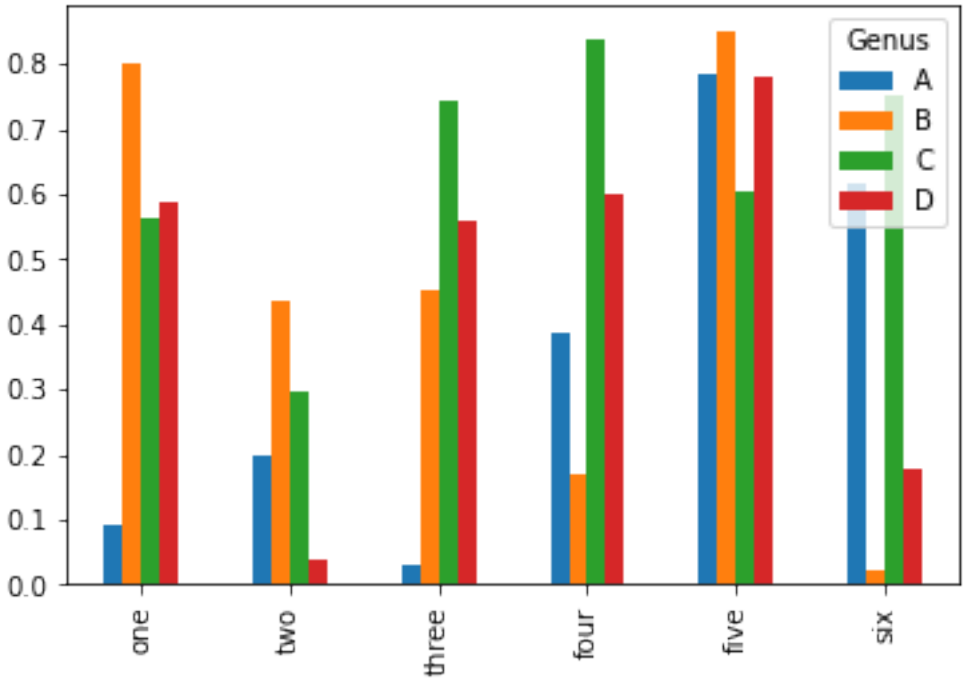
[64]:

Genus	A	B	C	D
one	0.090107	0.800069	0.565126	0.589348

```
two     0.198101  0.436118  0.295904  0.037558
three   0.030685  0.453105  0.744864  0.557295
four    0.385114  0.168073  0.838261  0.599052
five    0.782715  0.848509  0.603163  0.781061
six     0.615737  0.021165  0.750465  0.176042
```

[65]:
```
df.plot(kind = 'bar')
```

[65]:
```
<AxesSubplot:>
```



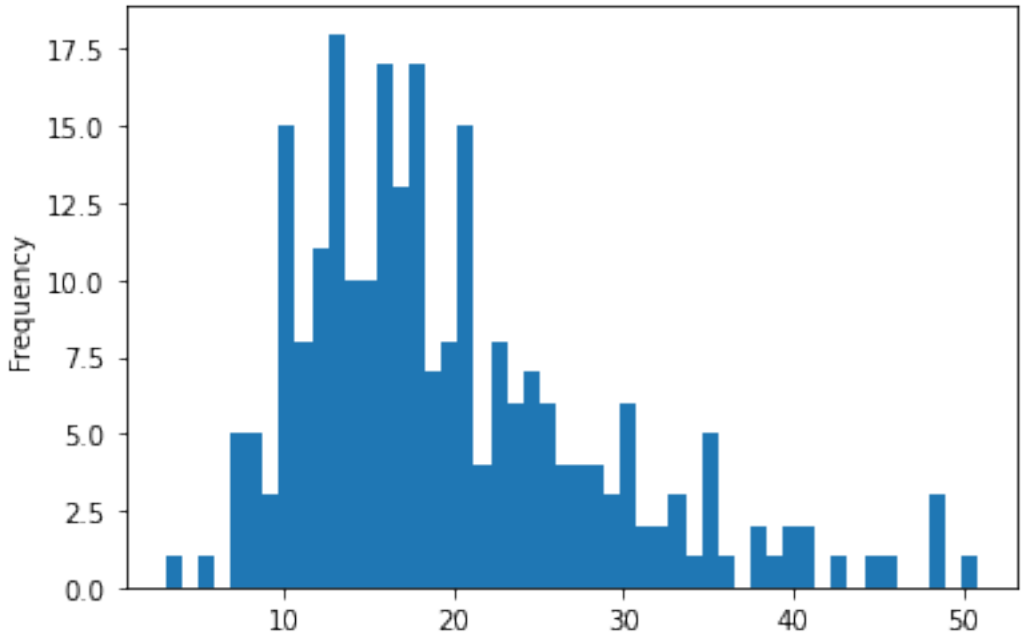
[69]:
```
tips = pd.read_csv('data/tips.csv')
tips.head()
```

[69]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

[80]:
```
tips.total_bill.plot(kind = 'hist',bins = 50)
#画直方图，数一下各个区间都有多少个数值
#bins 指定的是 50 个区间，算法会给自动划分好
```

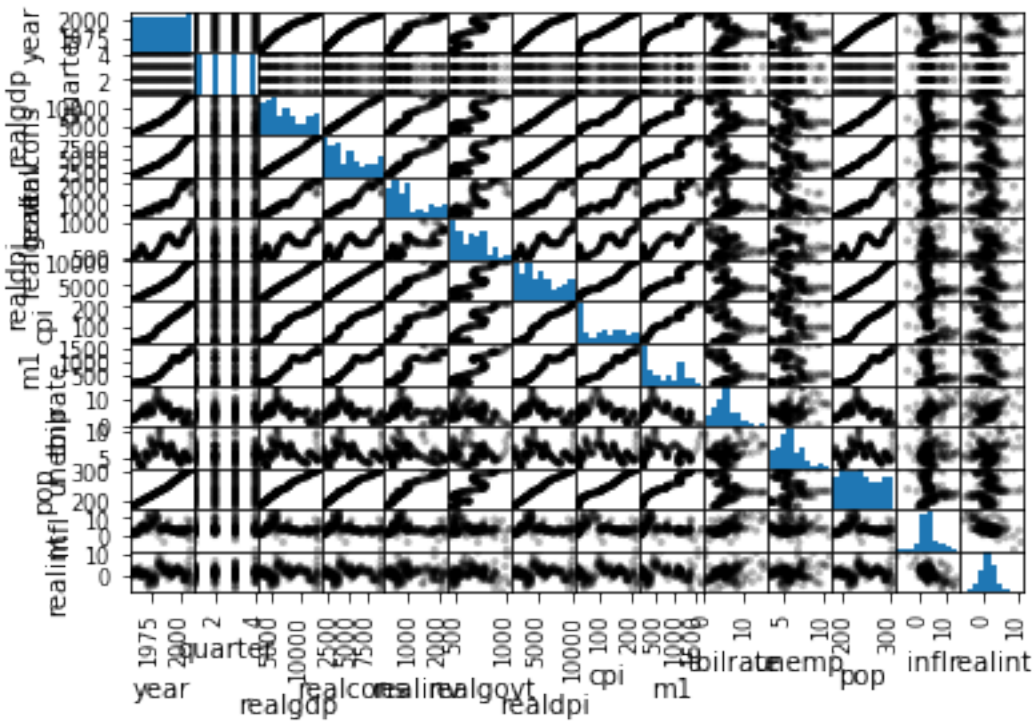
[80]: <AxesSubplot:ylabel='Frequency'>



[81]:
```
macro = pd.read_csv('data/macrodata.csv')
macro.head()
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	tbilrate	unemp	pop	infl	realint
0	1959	1	2710.349	1707.4	286.898	470.045	1886.9	28.98	139.7	2.82	5.8	177.146	0.00	0.00
1	1959	2	2778.801	1733.7	310.859	481.301	1919.7	29.15	141.7	3.08	5.1	177.830	2.34	0.74
2	1959	3	2775.488	1751.8	289.226	491.260	1916.4	29.35	140.5	3.82	5.3	178.657	2.74	1.09
3	1959	4	2785.204	1753.7	299.356	484.052	1931.3	29.37	140.0	4.33	5.6	179.386	0.27	4.06
4	1960	1	2847.699	1770.5	331.722	462.199	1955.5	29.54	139.6	3.50	5.2	180.007	2.31	1.19

[82]:
```
pd.plotting.scatter_matrix(macro,color = 'k',alpha = 0.3)
#把所有属性（列）都做一个两两的匹配然后画散点图
#注意：对角线上的是直方图，因为自身与自身画散点图是浪费，因为是 100% 的线性关系
```



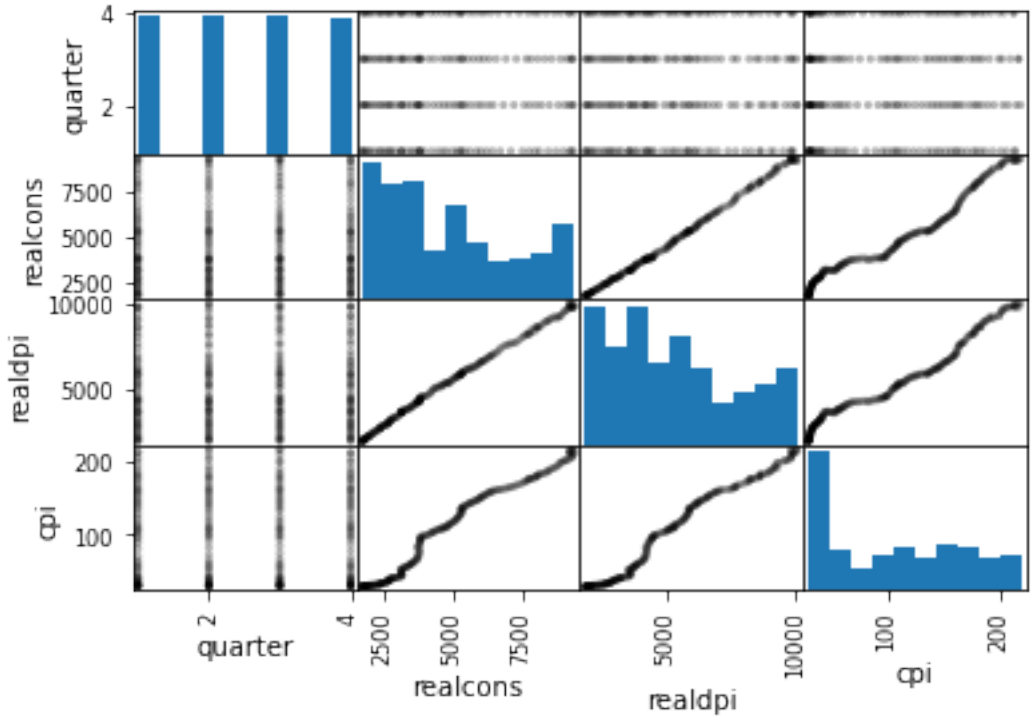
[84]:
```
#上面那个维度太多了不好查看，下面我们只抽出其中几列来看看
data = macro[['quarter','realcons','realdpi','cpi']]
data
```

[84]:

	quarter	realcons	realdpi	cpi
0	1	1707.4	1886.9	28.980
1	2	1733.7	1919.7	29.150
2	3	1751.8	1916.4	29.350
3	4	1753.7	1931.3	29.370
4	1	1770.5	1955.5	29.540
..	…	…	…	…
198	3	9267.7	9838.3	216.889
199	4	9195.3	9920.4	212.174
200	1	9209.2	9926.4	212.671
201	2	9189.0	10077.5	214.469
202	3	9256.0	10040.6	216.385

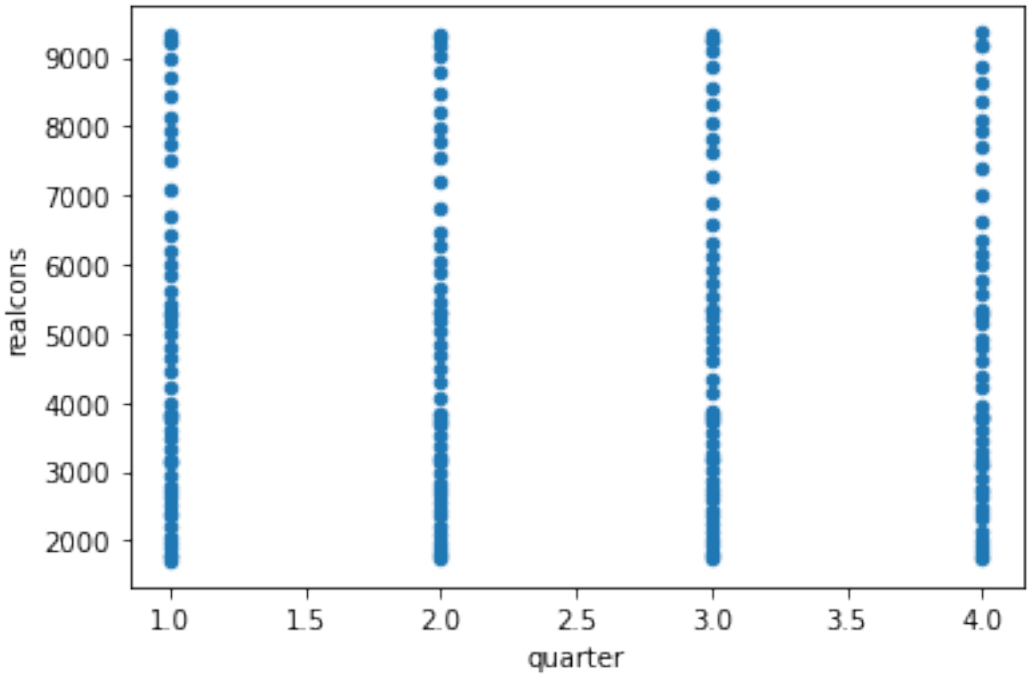
```
[203 rows x 4 columns]
```

[85]:
```
pd.plotting.scatter_matrix(data,color = 'k',alpha = 0.3)
#让 data 中的四个列两两配对画散点图
#注意：对角线上的是直方图，因为自身与自身画散点图是浪费，因为是 100% 的线性关系
```



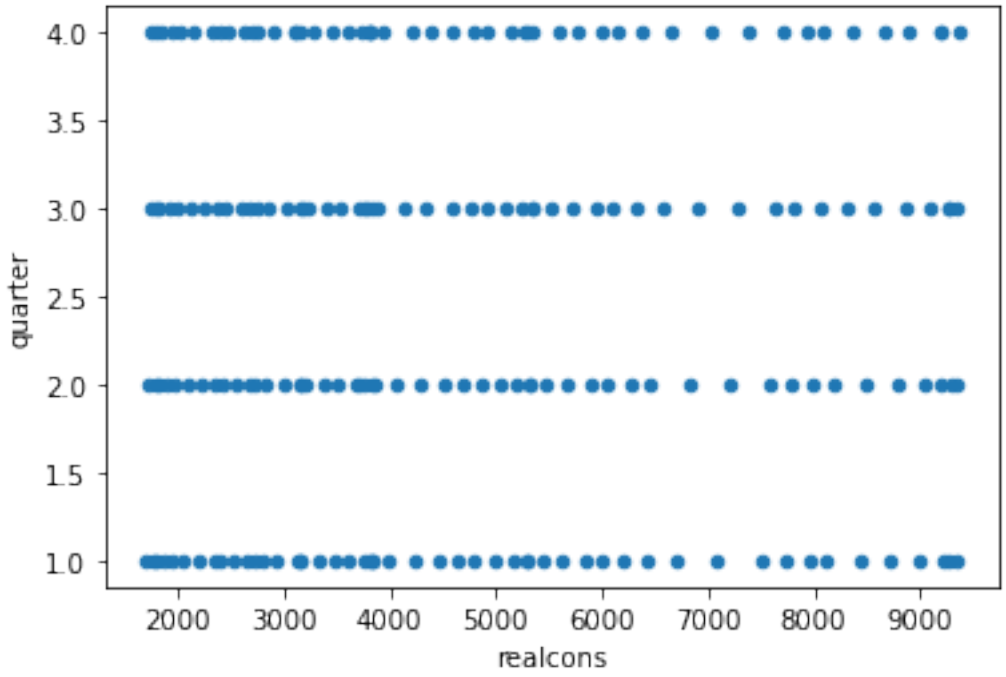
[87]:
```
data.plot.scatter('quarter','realcons')
#横坐标是'quarter'，纵坐标是'realcons
#就是上一条输出的第二行第一个图
```

[87]:
```
<AxesSubplot:xlabel='quarter', ylabel='realcons'>
```



[88]:
```
data.plot.scatter('realcons','quarter')
#给横纵坐标颠倒一下位置，就是之前那个大输出的第一行第二个图了
```

[88]: <AxesSubplot:xlabel='realcons', ylabel='quarter'>


4.0
3.5
3.0
2.5
2.0
1.5
1.0
quarter
2000
3000
4000
5000
6000
7000
8000
9000
realcons

8 大数据处理技巧

1. 看一下几行几列: **DataFrame.shape**，列太多了会很麻烦

2. 看一下占了多少内存: **DataFrame.info(memory_usage='deep')**，参数解释详见：click this

3. 转换数据类型： **pd.to_numeric()**，参数解释详见：click this

4. 用来给套函数： **Series/DataFrame.apply()**，详见

- **通过转换数据类型来减少内存的使用**
  `int64 → int32/int16/int8` 都行，只要转换后的数据类型不会被因为位数不够被撑爆就行
  `float64 → float32/float16` 同上
  `datetime64 → int32` 比如：2022-04-16 转换成 20220416 这样能压缩一半下去
  `object → category` 其实这里是最重要的，因为 object 是最占内存的

[13]:
```
gl = pd.read_csv('data/game_logs.csv')
gl.head()
```

	date	number_of_game	day_of_week	v_name	v_league	...	h_player_9_id	h_player_9_name	h_player_9_def_pos	additional_info	acquisition_info
0	18710504	0	Thu	CL1	na	...	kellb105	Bill Kelly	9.0	NaN	Y
1	18710505	0	Fri	BS1	na	...	berth101	Henry Berthrong	8.0	HTBF	Y
2	18710506	0	Sat	CL1	na	...	stirg101	Gat Stires	9.0	NaN	Y
3	18710508	0	Mon	CL1	na	...	zettg101	George Zettlein	1.0	NaN	Y
4	18710509	0	Tue	BS1	na	...	cravb101	Bill Craver	6.0	HTBF	Y

[5 rows x 161 columns]

[4]:
```
gl.shape
#看一下几行几列，因为列太多了，就会很麻烦
```

[4]:
```
(171907, 161)
```

[5]:
```
gl.info(memory_usage='deep')
#看一下占了多少内存，都有多少类型的数据
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 171907 entries, 0 to 171906
Columns: 161 entries, date to acquisition_info
dtypes: float64(77), int64(6), object(78)
```

```
memory usage: 859.4 MB
```

[8]:
```
for dtype in ['float64','int64','object']:
    selected = gl.select_dtypes(include = [dtype])
    mean_ussage_b = selected.memory_usage(deep = True).mean()
    mean_ussage_mb = mean_ussage_b/1024**2  #1024**2 1024的平方，b到kb到mb
    print('平均内存占用',dtype,np.round(mean_ussage_mb,decimals = 3))  #打印四舍五入到小数点后3位
```

```
平均内存占用 float64 1.295
平均内存占用 int64 1.124
平均内存占用 object 9.501
```

[12]:
```
int_types = ['uint8','int8','int16','int32','int64']
for it in int_types:
    print(np.iinfo(it))
#看一下各个数据类型能表达的数字在什么范围
```

```
Machine parameters for uint8
---------------------------------------------------------------
min = 0
max = 255
---------------------------------------------------------------

Machine parameters for int8
---------------------------------------------------------------
min = -128
max = 127
---------------------------------------------------------------

Machine parameters for int16
---------------------------------------------------------------
min = -32768
max = 32767
---------------------------------------------------------------

Machine parameters for int32
---------------------------------------------------------------
```

```
min = -2147483648
max = 2147483647
---------------------------------------------------------------

Machine parameters for int64
---------------------------------------------------------------
min = -9223372036854775808
max = 9223372036854775807
---------------------------------------------------------------
```

[21]:
```
#定义一个函数用于计算数据占用内存大小
def mem_usage(pandas_obj):
    if isinstance(pandas_obj,pd.DataFrame):
        usage_b = pandas_obj.memory_usage(deep=True).sum()
    else:
        usage_b = pandas_obj.memory_usage(deep=True)
    usage_mb = usage_b/1024**2
    return '{:03.2f}MB'.format(usage_mb)

gl_int = gl.select_dtypes(include = ['int64'])
converted_int = gl_int.apply(pd.to_numeric,downcast = 'unsigned')
print('转换前:',mem_usage(gl_int))
print('转换后:',mem_usage(converted_int))

#意思是想说，如果不耽误事的话，那么把数据类型向小了转换可以节省内存。
#不耽误事的意思是：只要转换后的数据的类型没有被撑爆
```

```
转换前: 7.87MB
转换后: 1.48MB
```

[22]:
```
gl_float = gl.select_dtypes(include = ['float64'])
converted_float = gl_float.apply(pd.to_numeric,downcast = 'float')
print('转换前:',mem_usage(gl_int))
print('转换后:',mem_usage(converted_int))
```

```
转换前: 7.87MB
转换后: 1.48MB
```

[26]:
```
copied_gl = gl.copy()

copied_gl[converted_int.columns] = converted_int
copied_gl[converted_float.columns] = converted_float

print('转换前:',mem_usage(gl))
print('转换后:',mem_usage(copied_gl))
#比对一下前后的大小
#converted_int/converted_float 是经过 select_dtypes 函数抽出来的符合类型条件的列的合集
#所以做一下.columns 一下后就可以取出符合条件的列名/索引。
#这样再带进去 copied_gl 这个 dataframe 里就能做到仅取出符合要求的列的合集

#但是我们会注意到其实没少多少
#因为 object 才是吃内存的主力!!!
```

转换前: 859.43MB
转换后: 802.54MB

[39]:
```
gl_object = gl.select_dtypes(include = ['object']).copy()
gl_object.describe()
#可以看到各个 object 都有是十几万条，但是下面表的第二行可以看出来，非重复（unique）的内容少，都是些重复的内容
#但是就算是重复的内容，他们都是会单独开一片内存去占用，并不会变成指针那样的去 share 内存区域
```

	day_of_week	v_name	v_league	h_name	h_league	...	h_player_8_name	h_player_9_id	h_player_9_name	additional_info	acquisition_info
count	171907	171907	171907	171907	171907	...	140838	140838	140838	1456	140841
unique	7	148	7	148	7	...	4710	5193	5142	332	1
top	Sat	CHN	NL	CHN	NL	...	Al Lopez	spahw101	Warren Spahn	HTBF	Y
freq	28891	8870	88866	9024	88867	...	676	339	339	1112	140841

[4 rows x 78 columns]

[ ]:
```
#所以我们就可以把这种重复性高的 object 数据转换成 category 结构，然后就可以使用指针（映射）了。
#这样就可以去节省内存了
```

[34]:
```
data = gl_object.day_of_week
data


#day_of_week不是函数，是上面表的第一列。
#同个这个操作可以取出第一列
#这个可以发现，因为这一列储存的是星期几，所以只有七种不同的数据
#但是十几万条数据都会给自己分配一个内存，就很浪费
```

[34]:
```
0         Thu
1         Fri
2         Sat
3         Mon
4         Tue
          …
171902    Sun
171903    Sun
171904    Sun
171905    Sun
171906    Sun
Name: day_of_week, Length: 171907, dtype: object
```

[35]:
```
data_cate = data.astype('category')
#用 Numpy 里学的命令来转换 dtype
data_cate
```

[35]:
```
0         Thu
1         Fri
2         Sat
3         Mon
4         Tue
          …
171902    Sun
171903    Sun
171904    Sun
171905    Sun
171906    Sun
Name: day_of_week, Length: 171907, dtype: category
```

```
Categories (7, object): ['Fri', 'Mon', 'Sat', 'Sun', 'Thu', 'Tue', 'Wed']
```

[37]:
```
data_cate.head(15).cat.codes
#data_cate.head(15).cat 是看 category 结构的内容 array 是什么
#是比如上面运行结果最后一行的 Categories (7, object): ['Fri', 'Mon', 'Sat',␣
↪'Sun', 'Thu', 'Tue', 'Wed']
#加上.codes 就能看到具体怎么编码的
#发现目录的编码就是数字，用于指定是 category 结构的内容的 array 里面的第几个
#这样就能很省内存了
#也达到了指针的效果
```

[37]:
```
0     4
1     0
2     2
3     1
4     5
5     4
6     2
7     2
8     1
9     5
10    5
11    6
12    4
13    0
14    2
dtype: int8
```

[38]:
```
print('转换前:',mem_usage(data))
print('转换后:',mem_usage(data_cate))
#就发现变化是真的大
#所以只要 object 的列里面的重复的值多的话，就能这么去转换来省内存了。
```

```
转换前: 9.84MB
转换后: 0.16MB
```

[51]:
```
def mem_usage(pandas_obj):#之前的用于计算数据占用内存大小的函数
    if isinstance(pandas_obj,pd.DataFrame):
        usage_b = pandas_obj.memory_usage(deep=True).sum()
    else:
        usage_b = pandas_obj.memory_usage(deep=True)
    usage_mb = usage_b/1024**2
    return usage_mb
converted_object = pd.DataFrame()
 #创建一个新的空的 DataFrame 结构。这样直接给传进去就好了
for col  gl_object.columns: #写一个小程序来批量转换 obeject
    num_unique_values = len(gl_object[col].unique())
   #.unique() 用于取出不重复的条目
    num_total_values = len(gl_object[col])
    if num_unique_values / num_total_values < 0.5:
    #只注目于小于独特率（不重复）小于 0.5 的，不过像是星期那种，可以再给他减小这个
        converted_object.loc[:,col] = gl_object[col].astype('category')
    else:
        converted_object.loc[:,col] = gl_object[col]
#转换 int:
gl_int = gl.select_dtypes(include = ['int64'])
converted_int = gl_int.apply(pd.to_numeric,downcast = 'unsigned')
#转换 float:
gl_float = gl.select_dtypes(include = ['float64'])
converted_float = gl_float.apply(pd.to_numeric,downcast = 'float')
#把三个转换的结果代入进 converted_gl:
converted_gl = gl.copy()
converted_gl[converted_int.columns] = converted_int
converted_gl[converted_float.columns] = converted_float
converted_gl[converted_object.columns] = converted_object
#看一下效果
print('转换前:{:03.2f}MB'.format(mem_usage(gl)))
print('转换后:{:03.2f}MB'.format(mem_usage(converted_gl)))
print('压缩率:{:.2%}'.format(1 - (mem_usage(converted_gl) / mem_usage(gl))))
```

```
转换前:859.43MB
转换后:101.86MB
压缩率:88.15%
```

[60]:
```
gl_date = gl.copy().date
print(gl_date.head())
print('{:03.2f}MB'.format(mem_usage(gl_date.astype('int32'))))
#本来是 int64 的，给压缩成 int32 的
#注意这里的日期格式，不是标准的日期模式
```

```
0    18710504
1    18710505
2    18710506
3    18710508
4    18710509
Name: date, dtype: int64

0.66MB
```

[58]:
```
print(pd.to_datetime(gl_date,format = '%Y%m%d').head())
print('{:03.2f}MB'.format(mem_usage(pd.to_datetime(gl_date,format = '%Y%m%d'))))
#但是我们转换成标准的日期格式后会发现反而要的内存多了

#所以!!! 上一条的那个原本的格式是日期的最优解!!!
```

```
0   1871-05-04
1   1871-05-05
2   1871-05-06
3   1871-05-08
4   1871-05-09
Name: date, dtype: datetime64[ns]

1.31MB
```