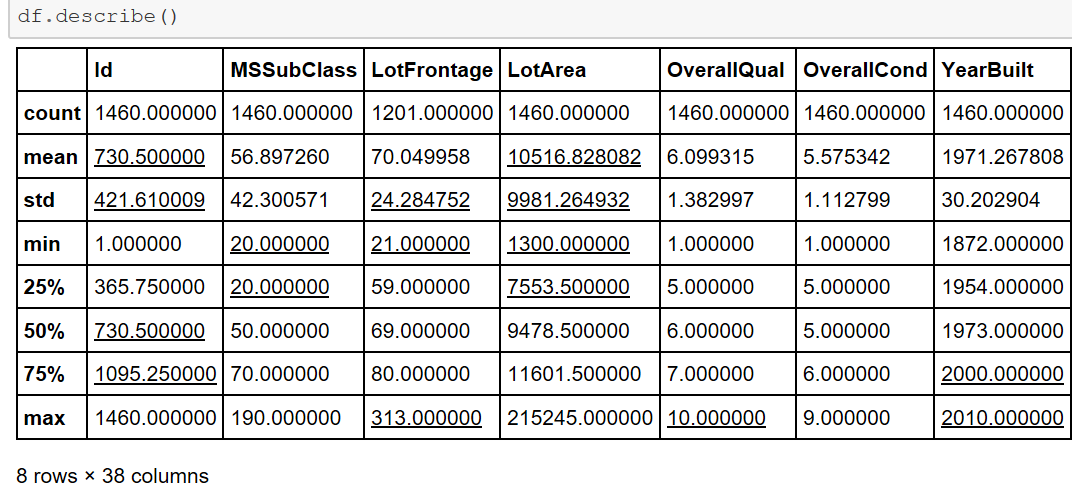
# 一、[数据预处理](https://www.cnblogs.com/gczr/p/6761613.html)

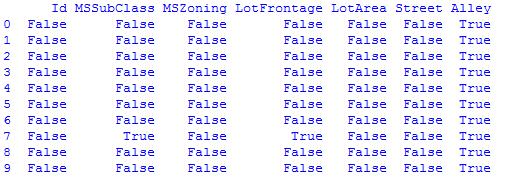
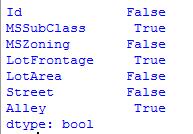
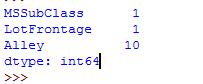
## 1、加载数据

　　houseprice=pd.read\_csv('../input/train.csv') #加载后放入dataframe里  
all\_data=pd.read\_csv('a.csv', header=0,parse\_dates=['time'],usecols=['time','LotArea','price']) #可以选择加载哪几列  
　 houseprice.head() #显示前5行数据  
 houseprice.info() #查看各字段的信息

houseprice.shape #查看数据集行列分布，几行几列  
　 houseprice.describe() #查看数据的大体情况



## 2、分析缺失数据

houseprice.isnull() #元素级别的判断，把对应的所有元素的位置都列出来，元素为空或者NA就显示True，否则就是False  
  
  
　　 houseprice.isnull().any()　 #列级别的判断，只要该列有为空或者NA的元素，就为True，否则False  
  
　　 missing=houseprice.columns[houseprice.isnull().any()].tolist() 　　　　#将为空或者NA的列找出来  
https://images2015.cnblogs.com/blog/1102791/201704/1102791-20170425140745662-955687954.jpg  
　　houseprice[missing].isnull().sum()　　　　#将列中为空或者NA的个数统计出来  


　　# 将某一列中缺失元素的值，用value值进行填充。处理缺失数据时，比如该列都是字符串，不是数值，可以将出现次数最多的字符串填充缺失值。

**def cat\_imputation(column, value):**

**houseprice.loc[houseprice[column].isnull(),column] = value**

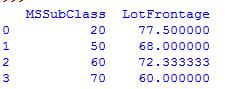
houseprice[['LotFrontage','Alley']][houseprice['Alley'].isnull()==True] #从LotFrontage 和Alley 列中进行选择行，选择Alley中数据为空的行。主要用来看两个列的关联程度，是不是大多同时为空。

　　houseprice['Fireplaces'][houseprice['FireplaceQu'].isnull()==True].describe() #对筛选出来的数据做一个描述，比如一共多少行，均值、方差、最小值、最大值等等。

## 3、统计分析

houseprice['MSSubClass'].value\_counts() #统计某一列中各个元素值出现的次数

print("Skewness: %f" % houseprice['MSSubClass'].skew()) #列出数据的偏斜度  
　　print("Kurtosis: %f" % houseprice['MSSubClass'].kurt())　　 #列出数据的峰度

houseprice['LotFrontage'].corr(houseprice['LotArea']) #计算两个列的相关度  
　　houseprice['SqrtLotArea']=np.sqrt(houseprice['LotArea']) #将列的数值求根，并赋予一个新列  
 houseprice[['MSSubClass', 'LotFrontage']].groupby(['MSSubClass'], as\_index=False).mean() #跟MSSubClass进行分组，并求分组后的平均值  


## 4、数据处理

**1)删除相关**  
　　del houseprice['SqrtLotArea'] #删除列

　houseprice['LotFrontage'].dropna() #去掉为空值或者NA的元素  
　　houseprice.drop(['Alley'],axis=1) #去掉Alley列，不管空值与否  
　　df.drop(df.columns[[0,1]],axis=1,inplace=True) #删除第1，2列，inplace=True表示直接就在内存中替换了，不用二次赋值生效。

houseprice.dropna(axis=0) #删除带有空值的行

houseprice.dropna(axis=1) #删除带有空值的列  
**2）缺失值填充处理**　　houseprice['LotFrontage']=houseprice['LotFrontage'].fillna(0) 　　#将该列中的空值或者NA填充为0  
　　all\_data.product\_type[all\_data.product\_type.isnull()]=all\_data.product\_type.dropna().mode().values #如果该列是字符串的，就将该列中出现次数最多的字符串赋予空值,mode()函数就是取出现次数最多的元素。  
　　houseprice['LotFrontage'].fillna(method='pad') 　　#使用前一个数值替代空值或者NA，就是NA前面最近的非空数值替换   
　　houseprice['LotFrontage'].fillna(method='bfill',limit=1)　#使用后一个数值替代空值或者NA，limit=1就是限制如果几个连续的空值，只能最近的一个空值可以被填充。  
　　houseprice['LotFrontage'].fillna(houseprice['LotFrontage'].mean()) 　　#使用平均值进行填充  
　　houseprice['LotFrontage'].interpolate() 　　　　# 使用插值来估计NaN 如果index是数字，可以设置参数method='value' ，如果是时间，可以设置method='time'  
　　houseprice= houseprice.fillna(houseprice.mean()) #将缺失值全部用该列的平均值代替，这个时候一般已经提前将字符串特征转换成了数值。

注：在kaggle中有人这样处理缺失数据，如果数据的缺失达到15%，且并没有发现该变量有多大作用，就删除该变量！  
  
 **3）字符串替换**

　　houseprice['MSZoning']=houseprice['MSZoning'].map({'RL':1,'RM':2,'RR':3,}).astype(int) #将MSZoning中的字符串变成对应的数字表示  
 **4）数据连接**  
　　merge\_data=pd.concat([new\_train,df\_test]) #讲训练数据与测试数据连接起来，以便一起进行数据清洗

all\_data = pd.concat((train.loc[:,'MSSubClass':'SaleCondition'], test.loc[:,'MSSubClass':'SaleCondition'])) #另一种合并方式，按列名字进行合并。  
　　　　res = pd.merge(df1, df2,on=['time']) #将df1,df2按照time字段进行合并，两个df中都含有time字段

**5）数据保存**merge\_data.to\_csv('merge\_data.csv'，index=False) #index=False，写入的时候不写入列的索引序号

**6）数据转换**houseprice["Alley"] = np.log1p(houseprice["Alley"]) #采用log(1+x)方式对原数据进行处理，改变原数据的偏斜度，使数据更加符合正态曲线分布。

　 numeric\_feats =houseprice.dtypes[houseprice.dtypes != "object"].index #把内容为数值的特征列找出来

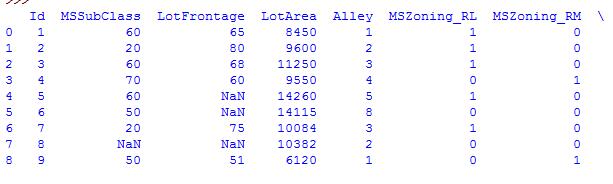
**#下面几行代码将偏斜度大于0.75的数值列做一个log转换，使之尽量符合正态分布，因为很多模型的假设数据是服从正态分布的**

　　skewed\_feats = train[numeric\_feats].apply(lambda x: skew(x.dropna())) #compute skewness

　　skewed\_feats = skewed\_feats[skewed\_feats >0.75]

　　skewed\_feats = skewed\_feats.index

　　all\_data[skewed\_feats] = np.log1p(all\_data[skewed\_feats])

　　houseprice= pd.get\_dummies(houseprice) #另外一种形式数据转换，将字符串特征列中的内容分别提出来作为新的特征出现，这样就省去了将字符串内容转化为数值特征内容的步骤了。  


**6）数据标准化**我们都知道大多数的梯度方法（几乎所有的机器学习算法都基于此）对于数据的缩放很敏感。因此，在运行算法之前，我们应该进行标准化，或所谓的规格化。标准化包括替换所有特征的名义值，让它们每一个的值在0和1之间。而对于规格化，它包括数据的预处理，使得每个特征的值有0和1的离差。Scikit-Learn库已经为其提供了相应的函数。

**from sklearn import preprocessing**

# normalize the data attributes

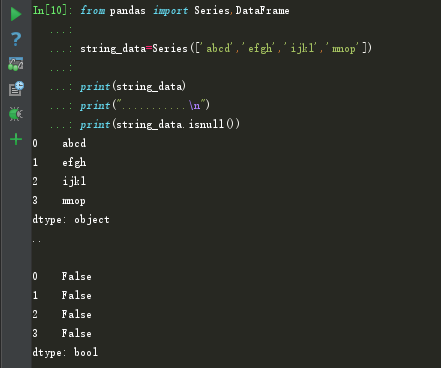
**normalized\_X = preprocessing.normalize(X)**

# standardize the data attributes

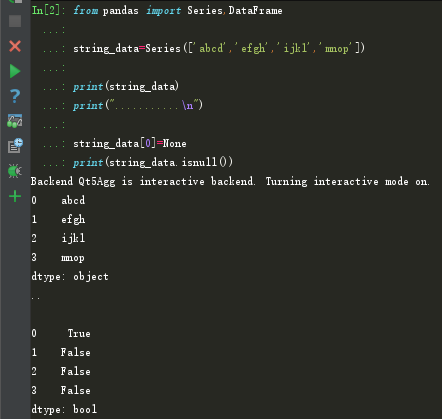
**standardized\_X = preprocessing.scale(X)**

## 5、数据缺失

数据缺失在大部分数据分析应用中都很常见，Pandas使用浮点值NaN表示浮点和非浮点数组中的缺失数据，他只是一个便于被检测出来的数据而已。



Python内置的None值也会被当作NA处理



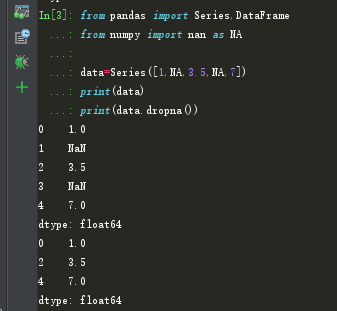
处理NA的方法有四种：dropna,fillna,isnull,notnull

*is(not)null*，这一对方法对对象做出元素级的应用，然后返回一个布尔型数组，一般可用于布尔型索引。

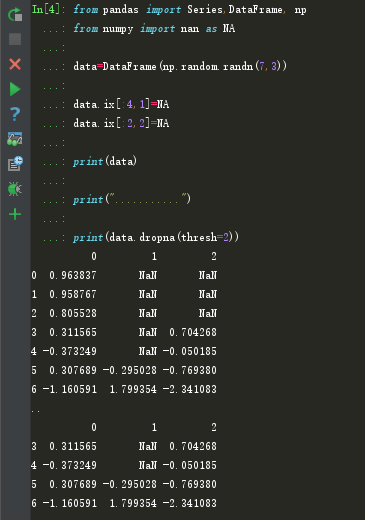
*dropna*，对于一个Series，dropna返回一个仅含非空数据和索引值的Series。

问题在于DataFrame的处理方式，因为一旦drop的话，至少要丢掉一行（列）。这里解决方法与前面类似，还是通过一个额外的参数：dropna(axis=0,how=’any’,thresh=None)，how参数可选的值为any或者all.all仅在切片元素全为NA时才抛弃该行(列)。thresh为整数类型，eg:thresh=3,那么一行当中至少有三个NA值时才将其保留。

*fillna*,fillna(value=None,method=None,axis=0)中的value除了基本类型外，还可以使用字典，这样可以实现对不同列填充不同的值。

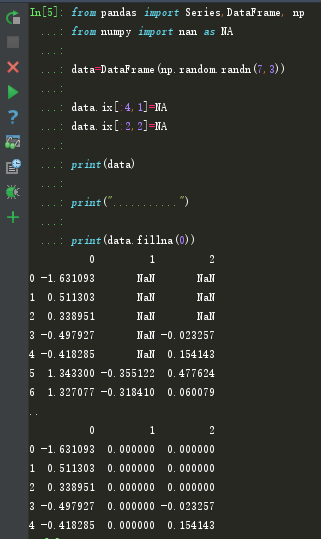


另一个过滤DataFrame行的问题涉及问题序列数据。假设只想留一部分观察数据，可以用thresh参数实现此目的：

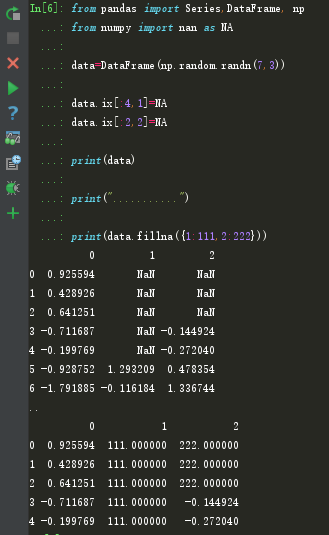


不想滤除缺失的数据，而是通过其他方式填补“空洞”，fillna是最主要的函数。

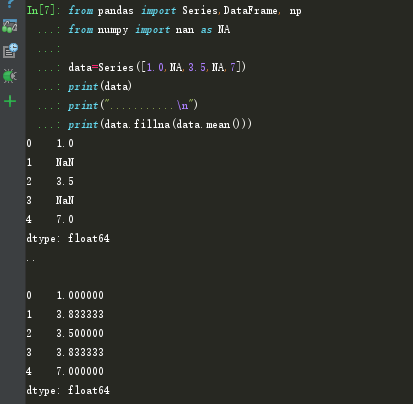
通过一个常数调用fillna就会将缺失值替换为那个常数值：



若是通过一个字典调用fillna，就可以实现对不同列填充不同的值。



可以利用fillna实现许多别的功能，比如可以传入Series的平均值或中位数：



数据缺失在大部分数据分析应用中都很常见，Pandas使用浮点值NaN表示浮点和非浮点数组中的缺失数据，他只是一个便于被检测出来的数据而已。

from pandas import Series,DataFrame

string\_data=Series(['abcd','efgh','ijkl','mnop'])

print(string\_data)

print("...........\n")

print(string\_data.isnull())

Python内置的None值也会被当作NA处理

from pandas import Series,DataFrame

string\_data=Series(['abcd','efgh','ijkl','mnop'])

print(string\_data)

print("...........\n")

string\_data[0]=None

print(string\_data.isnull())

处理NA的方法有四种：dropna,fillna,isnull,notnull

***is(not)null***，这一对方法对对象做出元素级的应用，然后返回一个布尔型数组，一般可用于布尔型索引。

***dropna***，对于一个Series，dropna返回一个仅含非空数据和索引值的Series。

问题在于DataFrame的处理方式，因为一旦drop的话，至少要丢掉一行（列）。这里解决方法与前面类似，还是通过一个额外的参数：dropna(axis=0,how=’any’,thresh=None)，how参数可选的值为any或者all.all仅在切片元素全为NA时才抛弃该行(列)。thresh为整数类型，eg:thresh=3,那么一行当中至少有三个NA值时才将其保留。

***fillna***,fillna(value=None,method=None,axis=0)中的value除了基本类型外，还可以使用字典，这样可以实现对不同列填充不同的值。

## 6、过滤数据

对于一个Series，dropna返回一个仅含非空数据和索引值的Series：

from pandas import Series,DataFrame

from numpy import nan as NA

data=Series([1,NA,3.5,NA,7])

print(data.dropna())

另一个过滤DataFrame行的问题涉及问题序列数据。假设只想留一部分观察数据，可以用thresh参数实现此目的：

from pandas import Series,DataFrame, np

from numpy import nan as NA

data=DataFrame(np.random.randn(7,3))

data.ix[:4,1]=NA

data.ix[:2,2]=NA

print(data)

print("...........")

print(data.dropna(thresh=2))

不想滤除缺失的数据，而是通过其他方式填补“空洞”，fillna是最主要的函数。   
通过一个常数调用fillna就会将缺失值替换为那个常数值：

from pandas import Series,DataFrame, np

from numpy import nan as NA

data=DataFrame(np.random.randn(7,3))

data.ix[:4,1]=NA

data.ix[:2,2]=NA

print(data)

print("...........")

print(data.fillna(0))

若是通过一个字典调用fillna，就可以实现对不同列填充不同的值。

from pandas import Series,DataFrame, np

from numpy import nan as NA

data=DataFrame(np.random.randn(7,3))

data.ix[:4,1]=NA

data.ix[:2,2]=NA

print(data)

print("...........")

print(data.fillna({1:111,2:222}))

可以利用fillna实现许多别的功能，比如可以传入Series的平均值或中位数：

from pandas import Series,DataFrame, np

from numpy import nan as NA

data=Series([1.0,NA,3.5,NA,7])

print(data)

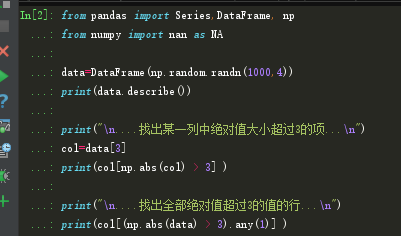
print("...........\n")

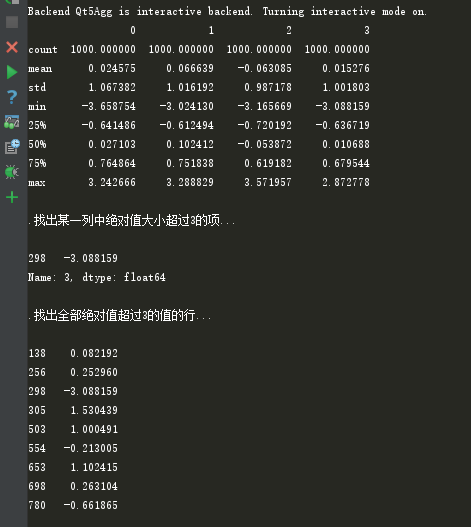
print(data.fillna(data.mean()))

## 7、检测和过滤异常值

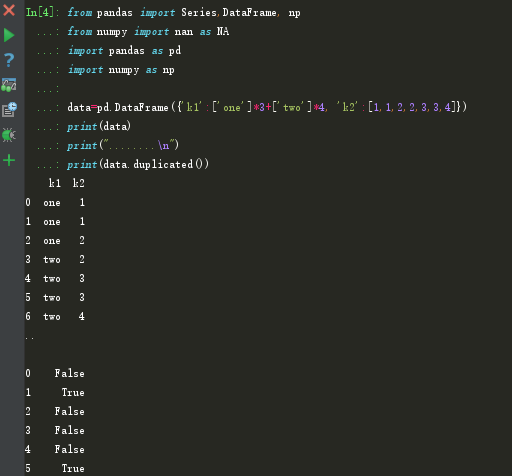
**检测和过滤异常值**

异常值(outlier)的过滤或变换运算在很大程度上就是数组运算。如下一个(1000,4)的标准正态分布数组：

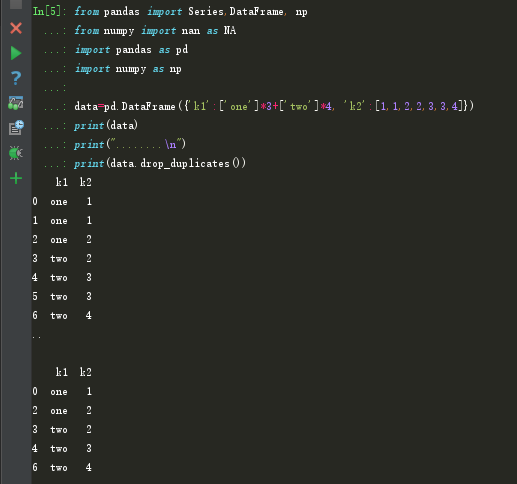




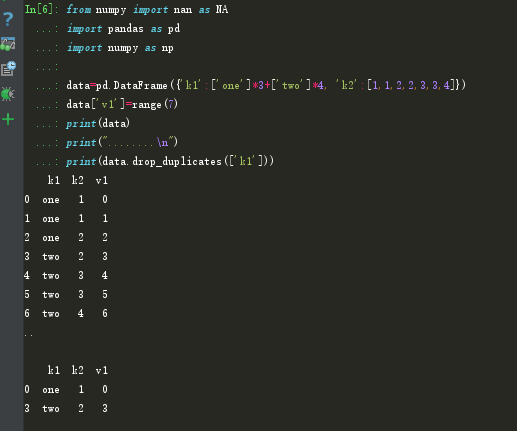
DataFrame的duplicated方法返回一个布尔型Series，表示各行是否是重复行。



与此相关的还有一个drop\_duplicated方法，它用于返回一个移除了重复行的DataFrame：



上面的两个方法会默认判断全部列，也可以指定部分列进行重复项判断，假设还有一列值，而只希望根据k1列过滤重复项。duplicates和drop\_duplicates默认保留第一个出现的值组合。传入take\_last=True则保留最后一个



异常值(outlier)的过滤或变换运算在很大程度上就是数组运算。如下一个(1000,4)的标准正态分布数组：

from pandas import Series,DataFrame, np

from numpy import nan as NA

data=DataFrame(np.random.randn(1000,4))

print(data.describe())

print("\n....找出某一列中绝对值大小超过3的项...\n")

col=data[3]

print(col[np.abs(col) >3] )

print("\n....找出全部绝对值超过3的值的行...\n")

print(col[(np.abs(data) > 3).any(1)] )

## 8、移除重复数据

DataFrame的duplicated方法返回一个布尔型Series，表示各行是否是重复行。

from pandas import Series,DataFrame, np

from numpy import nan as NA

import pandas as pd

import numpy as np

data=pd.DataFrame({'k1':['one']\*3+['two']\*4, 'k2':[1,1,2,2,3,3,4]})

print(data)

print("........\n")

print(data.duplicated())

与此相关的还有一个drop\_duplicated方法，它用于返回一个移除了重复行的DataFrame：

from pandas import Series,DataFrame, np

from numpy import nan as NA

import pandas as pd

import numpy as np

data=pd.DataFrame({'k1':['one']\*3+['two']\*4, 'k2':[1,1,2,2,3,3,4]})

print(data)

print("........\n")

print(data.drop\_duplicates())

上面的两个方法会默认判断全部列，也可以指定部分列进行重复项判断，假设还有一列值，而只希望根据k1列过滤重复项。

from pandas import Series,DataFrame, np

from numpy import nan as NA

import pandas as pd

import numpy as np

data=pd.DataFrame({'k1':['one']\*3+['two']\*4, 'k2':[1,1,2,2,3,3,4]})

data['v1']=range(7)

print(data)

print("........\n")

print(data.drop\_duplicates(['k1']))

duplicates和drop\_duplicates默认保留第一个出现的值组合。传入take\_last=True则保留最后一个：

from pandas import Series,DataFrame, np

from numpy import nan as NA

import pandas as pd

import numpy as np

data=pd.DataFrame({'k1':['one']\*3+['two']\*4, 'k2':[1,1,2,2,3,3,4]})

data['v1']=range(7)

print(data)

print("........\n")

print(data.drop\_duplicates(['k1','k2'],take\_last=True))

一般训练与测试的数据都提供csv格式，使用pandas库读取：

df\_train = pd.read\_csv('../train.csv')

此时读取的df\_train为DataFrame格式。   
同时pandas还可以读取各种不同格式的数据，如存储比较快的hdf格式、excel等

但有时数据不是简单的csv，它按照文本保存，如“ID||texttexttexttext”这样的一条数据需要将中间的“||”当作分隔符，读取方式如下：

train = pd.read\_csv('../input/training\_text', sep="\|\|", engine='python', header=None, skiprows=1, names=["ID","Text"])

## 9、数据粗略查看

在pandas读进来数据一个train后，train的格式为DataFrame，调用下面的几个方法就可以大致了解我们得到的数据是什么，有什么特征值，特征值的数据类型是什么，如果是数值那么最大最小值是什么等：

train.head(5) #显示前5行数据

train.tail(5) #显示后5行

train.columns#查看列名

train.info() #查看各字段的信息

train.shape#查看数据集行列分布，几行几列

train.describe() #查看数据的大体情况

如train.describe()：   
处理丢失的数据

处理这些数据以前不建议把train与test 连接起来，因为这样容易造成test里数据的丢失，个人认为较好的方式为：

full\_data = [train, test]

将两个数据集合成为list，然后清洗时对其for循环即可，如：

for dataset in full\_data:

dataset['FamilySize'] = dataset['SibSp'] + dataset['Parch'] + 1

下面只介绍对测试集train 的操作。

找到丢失的位置

输出每个列丢失值也即值为NaN的数据和，并从多到少排序：

total = train.isnull().sum().sort\_values(ascending=False)

print(total)

Cabin 687

Age 177

Embarked 2

Fare 0

Ticket 0

Parch 0

SibSp 0

Sex 0

Name 0

Pclass 0

Survived 0

PassengerId 0

也可以输出百分比：

percent =(train.isnull().sum()/train.isnull().count()).sort\_values(ascending=False)

missing\_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])

missing\_data.head(20)

由此可以看到‘Cabin’的缺失数量最多，‘Embarked’最少。

缺失值得处理

对缺失数据的处理我们有很多方法，如《机器学习实战》里提到：

使用可用特征的均值来填补缺失值；

使用特殊值来填补缺失值，如-1；

忽略有缺失值的样本；

使用相似样本的均值添补缺失值；

使用另外的机器学习算法预测缺失值。

填补

#使用出现次数最多的值填补

train['Embarked'] = train['Embarked'].fillna('S')

train.product\_type[train.product\_type.isnull()]=train.product\_type.dropna().mode().values

这里的特殊值是在这个特征下出现最多的一个值，特殊值的选取需要根据情况来判断，中位数也是特殊值：

#使用中位数填补

train['Fare'] = train['Fare'].fillna(train['Fare'].median())

使用平均数填补

train['Age'] = train['Age'].fillna(train['Age'].mean())

train['LotFrontage'].fillna(train['LotFrontage'].mean())

忽略

根据情况可以选择忽略这一特征的列和忽略出现缺失的那几行。   
通常后者出现在缺失的行数比较少的情况下。

#去掉一列

train = train.drop(['Cabin'], axis = 1)

#去掉这个特征为空的行

#当然后面可以加上inplace=True表示直接就在内存中替换了不用再赋值个train\_new，但是本人多次删除掉几个行，发现有问题时又需要重新建立已经分析好的train，很浪费时间，个人认为还是重新开辟一个比较好

train\_new = train.drop(train[train['Embarked'].isnull()].index)

#返回已经去掉重复行的数据集

train.drop\_duplicates()

处理偏离值

查找偏离值

将数据可视化（python中的pyplot），观察异常值，引用HousePrice中的一个例子：

#bivariate analysis saleprice/grlivarea

var = 'GrLivArea'

data = pd.concat([df\_train['SalePrice'], df\_train[var]], axis=1)

data.plot.scatter(x=var, y='SalePrice', ylim=(0,800000));

可以发现，有四个点在非常偏离的位置，先讨论上面两个偏移位。   
在下方无数点形成非常好的线性关系，那这偏移很大的两点是异常值吗？不是，虽然偏离位置很远，但是与下面无数点形成的线性关系还是拟合的。   
再看看右下两个值，显然它们偏离得没有道理，删！

删除偏离值

train.sort\_values(by = 'GrLivArea', ascending = False)[:2]

train= train.drop(train[train['Id'] == 1299].index)

train= train.drop(train[train['Id'] == 524].index)

保留偏离值

当然并不是所有的偏离值都需要删除，具体需要在分析之后选择处理方式。这里将偏离值保留下来并不是原封不动保留，而需要做标准化或归一化处理，具体的处理方式可查看最后一节数据转换、标准化、归一化。

## 10、数据统计

现在我们已经拿到了一份比较完整的数据，让我们开始对它做一个简单的统计。   
统计包括数据的偏斜度、峰度、不同特征值的相关系数、协方差、均值等。

#统计某一列中各个元素值出现的次数

train['MSSubClass'].value\_counts()

#

#列出数据的偏斜度

train['MSSubClass'].skew()

#列出数据的峰度

train['MSSubClass'].kurt()

#计算两个列的相关度

train['LotFrontage'].corr(train['LotArea'])

#观察两个列的值的二维图

x = 'GrLivArea'；y = 'SalePrice'

data = pd.concat([train[y], train[x]], axis=1)

data.plot.scatter(x=x, y=y, ylim=(0,800000));#这里800000为y的最大值

#计算所有特征值每两个之间的相关系数，并作图表示。

corrmat = train.corr()#得到相关系数

f,ax = plt.subplots(figsize = (12,9))

sns.heatmap(corrmat, vmax = .8, square = True)#热点图

#取出相关性最大的前十个，做出热点图表示

k = 10#number of variables for heatmap

cols = corrmat.nlargest(k, 'SalePrice')['SalePrice'].index

cm = np.corrcoef(train[cols].values.T)

sns.set(font\_scale=1.25)

hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot\_kws={'size': 10}, yticklabels=cols.values, xticklabels=cols.values)

plt.show()

热点图展示： 

## 11、特征值的合并、连接

在合并连接之前，我们需要了解pandas.groupby这个分组方法，因为很多时候我们是在从几个特征值里挖掘一些值来当作新的特征值，这样子我们这个分组的方法就显得尤为重要了，如按照同一个用户进行分组来计算这个用户的行为次数当作新的特征值等等

分组的常用代码块

#按照用户分组---------------------一个特征值

train.groupby('userid',as\_index=False)

#按照用户与目的地分组---------------两个特征值

train.groupby(['userid','end\_loc'],as\_index=False)

#用户、起点、目的地-----------------三个特征值

train.groupby(['userid','start\_loc','end\_loc'],as\_index=False)

#跟MSSubClass进行分组，并求分组后的平均值

train[['MSSubClass', 'LotFrontage']].groupby(['MSSubClass'], as\_index=False).mean()

#选取特定的属性的某个值然后进行分类

train[train['date']=='2017-1-2'].groupby(['userid'],as\_index=False)

按照分组的结果，得到需要的信息然后合并

#获得分组后，统计分组中'end\_loc'的数量返回为一列由‘userid’和‘user\_count’组成的新的DataFrame

user\_count = train.groupby('userid',as\_index=False)['end\_loc'].agg({'user\_count':'count'})

#将获得的新的DataFrame合并到train，更多的merge参数请查阅文档

train= pd.merge(train,user\_count,on=['userid'],how='left')

user\_eloc\_count = train.groupby(['userid','end\_loc'],as\_index=False)['userid'].agg({'user\_eloc\_count':'count'})

train= pd.merge(train,user\_eloc\_count,on=['userid','end\_loc'],how='left')

数据的连接

#讲训练数据与测试数据连接起来，以便一起进行数据清洗。

#这里需要注意的是，如果没有后面的ignore\_index=True，那么index的值在连接后的这个新数据中是不连续的，如果要按照index删除一行数据，可能会发现多删一条。

merge\_data=pd.concat([train,test]，ignore\_index=True)

#另一种合并方式，按列名字进行合并。

all\_data = pd.concat((train.loc[:,'MSSubClass':'SaleCondition'], test.loc[:,'MSSubClass':'SaleCondition']))

## 12、数据转换、标准化、归一化

数值转换

#浮点型数值转换为整型

train['Age']=train['Age'].astype(int)

#字符串的替换--映射

train['MSZoning']=train['MSZoning'].map({'RL':1,'RM':2,'RR':3,}).astype(int)

train['Embarked'] = train['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(int)

#一般建议将map拿出来

title\_mapping = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}

train['Title'] = train['Title'].map(title\_mapping)

train['Title'] = train['Title'].fillna(0)

#将字符串特征列中的内容分别提出来作为新的特征出现，表现为0、1。

train= pd.get\_dummies(houseprice)

#将连续型特征值分块，每一块用数字标识

train.loc[ train['Fare'] <= 7.91, 'Fare'] = 0

train.loc[(train['Fare'] >7.91) & (train['Fare'] <= 14.454), 'Fare'] = 1

train.loc[(train['Fare'] >14.454) & (train['Fare'] <= 31), 'Fare'] = 2

train.loc[ train['Fare'] >31, 'Fare'] = 3

train['Fare'] = train['Fare'].astype(int)

下面这个数值转换是将数值进行log计算，使分布的数值显常态

train['SalePrice'] = np.log(train['SalePrice'])

1

而有时这样的log不可行，就需要使用log（x+1）来 处理，至于原因请[点击链接](http://blog.csdn.net/liyuanbhu/article/details/8544644)

train["SalePrice"] = np.log1p(train["SalePrice"])

1

#将偏斜度大于0.75的数值列log转换，使之尽量符合正态分布。

skewed\_feats = train[numeric\_feats].apply(lambda x: skew(x.dropna())) #compute skewness

skewed\_feats = skewed\_feats[skewed\_feats >0.75]

skewed\_feats = skewed\_feats.index

all\_data[skewed\_feats] = np.log1p(all\_data[skewed\_feats])

数据标准化和归一化（Standardization、Normalization）

标准化归一化概念不再赘述，实际使用时最主要的还是要了解什么时候需要标准化，什么时候用归一化，还需要清楚当前数据适合什么标准化方式等等。   
在[sklearn.preprocessing](http://scikit-learn.org/stable/modules/preprocessing.html#normalization) 介绍的标准化方式有：   
1. preprocessing.scale()、preprocessing.StandardScaler()，使数据集呈现标准正态分布，即mean = 0，且标准差std = 1。   
2. MinMaxScaler 、MaxAbsScaler，前者使数据集分布在[0,1]，后者分布在[-1,1]。这种方式通常在(1) 特征的标准差较小 (2) 可以使稀疏数据集中的0值继续为0，这两种情况下使用。   
3. preprocessing.QuantileTransformer()，将数据映射到[0,1]之间均匀分布，会破坏原数据之间的相关特性。   
4. 归一化方式：preprocessing.normalize()，将样本缩放成单位向量，(1)需要使用二次方程，比如点积或者其他核方法计算样本对之间的相似性(2)常用于文本分类和内容聚类的向量空间模型的基础。

# 二、Pandas统计分析

pandas 是基于NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。pandas提供了大量能使我们快速便捷地处理数据的函数和方法。

　　>>> from pandas import Series, DataFrame

　　>>> import pandas as pd

### pandas

**表pandas常用函数**

|  |  |
| --- | --- |
| **函数** | **说明** |
| pd.isnull(series)  pd.notnull(series) | 判断是否为空(NaN)  判断是否不为空(not NaN) |
|  |  |
|  |  |

### B.Series

　　　　Series可以运用ndarray或字典的几乎所有索引操作和函数，融合了字典和ndarray的优点。

**表Series常用属性**

|  |  |
| --- | --- |
| **属性** | **说明** |
| values | 获取数组 |
| index | 获取索引 |
| name | values的name |
| index.name | 索引的name |

**表Series常用函数**

|  |  |
| --- | --- |
| **函数** | **说明** |
| Series([x,y,...])Series({'a':x,'b':y,...}, index=param1) | 生成一个Series |
| Series.copy() | 复制一个Series |
| Series.reindex([x,y,...], fill\_value=NaN)  Series.reindex([x,y,...], method=NaN)  Series.reindex(columns=[x,y,...]) | 重返回一个适应新索引的新对象，将缺失值填充为fill\_value  返回适应新索引的新对象，填充方式为method  对列进行重新索引 |
| Series.drop(index) | 丢弃指定项 |
| Series.map(f) | 应用元素级函数 |
|  |  |
| **排序函数** | **说明** |
| Series.sort\_index(ascending=True) | 根据索引返回已排序的新对象 |
| Series.order(ascending=True) | 根据值返回已排序的对象，NaN值在末尾 |
| Series.rank(method='average', ascending=True, axis=0) | 为各组分配一个平均排名 |
| df.argmax()  df.argmin() | 返回含有最大值的索引位置  返回含有最小值的索引位置 |

　　　　reindex的method选项：

　　　　　　ffill, bfill　　　　　向前填充/向后填充

　　　　　　pad, backfill　　　向前搬运，向后搬运

　　　　rank的method选项

　　　　　　'average'　　　　在相等分组中，为各个值分配平均排名

　　　　　　'max','min'　　　使用整个分组中的最小排名

　　　　　　'first'　　　　　　按值在原始数据中出现的顺序排名

### C.DataFrame

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

　　　　DataFrame可以通过类似字典的方式或者.columnname的方式将列获取为一个Series。行也可以通过位置或名称的方式进行获取。

　　　　为不存在的列赋值会创建新列。

　　　　>>> del frame['xxx']　　# 删除列

**表DataFrame常用属性**

|  |  |
| --- | --- |
| **属性** | **说明** |
| values | DataFrame的值 |
| index | 行索引 |
| index.name | 行索引的名字 |
| columns | 列索引 |
| columns.name | 列索引的名字 |
| ix | 返回行的DataFrame |
| ix[[x,y,...], [x,y,...]] | 对行重新索引，然后对列重新索引 |
| T | frame行列转置 |
|  |  |

**表Dataframe常用函数**

|  |  |
| --- | --- |
| **函数** | **说明** |
| DataFrame(dict, columns=dict.index, index=[dict.columnnum])  DataFrame(二维ndarray)  DataFrame(由数组、列表或元组组成的字典)  DataFrame(NumPy的结构化/记录数组)  DataFrame(由Series组成的字典)  DataFrame(由字典组成的字典)  DataFrame(字典或Series的列表)  DataFrame(由列表或元组组成的列表)  DataFrame(DataFrame)  DataFrame(NumPy的MaskedArray) | 构建DataFrame  数据矩阵，还可以传入行标和列标  每个序列会变成DataFrame的一列。所有序列的长度必须相同  类似于“由数组组成的字典”  每个Series会成为一列。如果没有显式制定索引，则各Series的索引会被合并成结果的行索引  各内层字典会成为一列。键会被合并成结果的行索引。  各项将会成为DataFrame的一行。索引的并集会成为DataFrame的列标。  类似于二维ndarray  沿用DataFrame  类似于二维ndarray，但掩码结果会变成NA/缺失值 |
| df.reindex([x,y,...], fill\_value=NaN, limit)  df.reindex([x,y,...], method=NaN)  df.reindex([x,y,...], columns=[x,y,...],copy=True) | 返回一个适应新索引的新对象，将缺失值填充为fill\_value，最大填充量为limit  返回适应新索引的新对象，填充方式为method  同时对行和列进行重新索引，默认复制新对象。 |
| df.drop(index, axis=0) | 丢弃指定轴上的指定项。 |
|  |  |
| **排序函数** | **说明** |
| df.sort\_index(axis=0, ascending=True)  df.sort\_index(by=[a,b,...]) | 根据索引排序 |
|  |  |
| **汇总统计函数** | **说明** |
| df.count() | 非NaN的数量 |
| df.describe() | 一次性产生多个汇总统计 |
| df.min()  df.min() | 最小值  最大值 |
| df.idxmax(axis=0, skipna=True)  df.idxmin(axis=0, skipna=True) | 返回含有最大值的index的Series  返回含有最小值的index的Series |
| df.quantile(axis=0) | 计算样本的分位数 |
| df.sum(axis=0, skipna=True, level=NaN)  df.mean(axis=0, skipna=True, level=NaN)  df.median(axis=0, skipna=True, level=NaN)  df.mad(axis=0, skipna=True, level=NaN)  df.var(axis=0, skipna=True, level=NaN)  df.std(axis=0, skipna=True, level=NaN)  df.skew(axis=0, skipna=True, level=NaN)  df.kurt(axis=0, skipna=True, level=NaN)  df.cumsum(axis=0, skipna=True, level=NaN)  df.cummin(axis=0, skipna=True, level=NaN)  df.cummax(axis=0, skipna=True, level=NaN)  df.cumprod(axis=0, skipna=True, level=NaN)  df.diff(axis=0)  df.pct\_change(axis=0) | 返回一个含有求和小计的Series  返回一个含有平均值的Series  返回一个含有算术中位数的Series  返回一个根据平均值计算平均绝对离差的Series  返回一个方差的Series  返回一个标准差的Series  返回样本值的偏度（三阶距）  返回样本值的峰度（四阶距）  返回样本的累计和  返回样本的累计最大值  返回样本的累计最小值  返回样本的累计积  返回样本的一阶差分  返回样本的百分比数变化 |
|  |  |
|  |  |
| **计算函数** | **说明** |
| df.add(df2, fill\_value=NaN, axist=1)  df.sub(df2, fill\_value=NaN, axist=1)  df.div(df2, fill\_value=NaN, axist=1)  df.mul(df2, fill\_value=NaN, axist=1) | 元素级相加，对齐时找不到元素默认用fill\_value  元素级相减，对齐时找不到元素默认用fill\_value  元素级相除，对齐时找不到元素默认用fill\_value  元素级相乘，对齐时找不到元素默认用fill\_value |
| df.apply(f, axis=0) | 将f函数应用到由各行各列所形成的一维数组上 |
| df.applymap(f) | 将f函数应用到各个元素上 |
| df.cumsum(axis=0, skipna=True) | 累加，返回累加后的dataframe |

**表Dataframe常用索引方式**

|  |  |
| --- | --- |
| **索引方式** | **说明** |
| df[val] | 选取DataFrame的单个列或一组列 |
| df.ix[val] | 选取Dataframe的单个行或一组行 |
| df.ix[:,val] | 选取单个列或列子集 |
| df.ix[val1,val2] | 将一个或多个轴匹配到新索引 |
| reindex方法 | 将一个或多个轴匹配到新索引 |
| xs方法 | 根据标签选取单行或者单列，返回一个Series |
| icol、irow方法 | 根据整数位置选取单列或单行，并返回一个Series |
| get\_value、set\_value | 根据行标签和列标签选取单个值 |

运算：

　一般情况下，Dataframe和Series之间的算术运算会将Series的索引匹配到的Dataframe的列，沿着列一直向下传播。若索引找不到，则会重新索引产生并集。

### D.Index

　pandas的索引对象负责管理轴标签和其他元数据（比如轴名称等）。构建Series或DataFrame时，所用到的任何数组或其他序列的标签都会被转换成一个Index。**Index对象不可修改，从而在多个数据结构之间安全共享。**

**表 主要的Index属性**

|  |  |
| --- | --- |
| **主要的Index对象** | **说明** |
| Index | 最广泛的Index对象，将轴标签表示为一个由Python对象组成的NumPy数组 |
| Int64Index | 针对整数的特殊Index |
| MultiIndex | “层次化”索引对象，表示单个轴上的多层索引。可以看做由元组组成的数组 |
| DatetimeIndex | 存储纳秒级时间戳（用NumPy的Datetime64类型表示） |
| PeriodIndex | 针对Period数据（时间间隔）的特殊Index |

**表常用Index函数**

|  |  |
| --- | --- |
| **函数** | **说明** |
| Index([x,y,...]) | 创建索引 |
| append(Index) | 连接另一个Index对象，产生一个新的Index |
| diff(Index) | 计算差集，产生一个新的Index |
| intersection(Index) | 计算交集 |
| union(Index) | 计算并集 |
| isin(Index) | 检查是否存在与参数索引中，返回bool型数组 |
| delete(i) | 删除索引i处元素，得到新的Index |
| drop(str) | 删除传入的值，得到新Index |
| insert(i,str) | 将元素插入到索引i处，得到新Index |
| is\_monotonic() | 当各元素大于前一个元素时，返回true |
| is\_unique() | 当Index没有重复值时，返回true |
| unique() | 计算Index中唯一值的数组 |

pandas 是基于 Numpy 构建的含有更高级数据结构和工具的数据分析包

类似于 Numpy 的核心是 ndarray，pandas 也是围绕着 Series 和 DataFrame 两个核心数据结构展开的 。Series 和 DataFrame 分别对应于一维的序列和二维的表结构。pandas 约定俗成的导入方法如下：

lang:python

from pandas import Series,DataFrame

import pandas as pd

Series 可以看做一个“定长的有序字典”。任意一维数据都可以用来构造 Series 对象：

lang:python

>>> s = Series([1,2,3.0,'abc'])

>>> s

0 1

1 2

2 3

3 abc

dtype: object

Series 对象包含两个主要的属性：index 和 values，分别为上例中左右两列。因为传给构造器的是一个列表，所以 index 的值是从 0 起递增的整数，如果传入的是一个类字典的键值对结构，就会生成 index-value 对应的 Series；或者在初始化的时候以关键字参数显式指定一个 index 对象：

lang:python

>>> s = Series(data=[1,3,5,7],index = ['a','b','x','y'])

>>> s

a 1

b 3

x 5

y 7

dtype: int64

>>> s.index

Index(['a', 'b', 'x', 'y'], dtype='object')

>>> s.values

array([1, 3, 5, 7], dtype=int64)

Series 对象的元素会严格依照给出的 index 构建，这意味着：如果 data 参数是有键值对的，那么只有 index 中含有的键会被使用；以及如果 data 中缺少响应的键，即使给出 NaN 值，这个键也会被添加。

注意 Series 的 index 和 values 的元素之间虽然存在对应关系，但这与字典的映射不同。index 和 values 实际仍为互相独立的 ndarray 数组，因此 Series 对象的性能完全 ok。

Series 这种使用键值对的数据结构最大的好处在于，Series 间进行算术运算时，index 会自动对齐。

另外，Series 对象和它的 index 都含有一个 name 属性：

lang:python

>>> s.name = 'a\_series'

>>> s.index.name = 'the\_index'

>>> s

the\_index

a 1

b 3

x 5

y 7

Name: a\_series, dtype: int64

DataFrame 是一个“表格型”的数据结构，它含有一组有序的列（类似于 index），每列可以是不同的值类型（不像 ndarray 只能有一个 dtype）。基本上可以把 DataFrame 看成是共享同一个 index 的 Series 的集合。

DataFrame 的构造方法与 Series 类似，只不过可以同时接受多条一维数据源，每一条都会成为单独的一列：

lang:python

>>> data = {'state':['Ohino','Ohino','Ohino','Nevada','Nevada'],

'year':[2000,2001,2002,2001,2002],

'pop':[1.5,1.7,3.6,2.4,2.9]}

>>> df = DataFrame(data)

>>> df

pop state year

01.5 Ohino 2000

11.7 Ohino 2001

23.6 Ohino 2002

32.4 Nevada 2001

42.9 Nevada 2002

[5 rows x 3 columns]

虽然参数 data 看起来是个字典，但字典的键并非充当 DataFrame 的 index 的角色，而是 Series 的 “name” 属性。这里生成的 index 仍是 "01234"。

**较**完整的 DataFrame 构造器参数为：DataFrame(data=None,index=None,coloumns=None)，columns 即 “name”：

lang:python

>>> df = DataFrame(data,index=['one','two','three','four','five'],

columns=['year','state','pop','debt'])

>>> df

year state pop debt

one 2000 Ohino 1.5NaN

two 2001 Ohino 1.7NaN

three 2002 Ohino 3.6NaN

four 2001 Nevada 2.4NaN

five 2002 Nevada 2.9NaN

[5 rows x 4 columns]

同样缺失值由 NaN 补上。看一下 index、columns 和 索引的类型：

lang:python

>>> df.index

Index(['one', 'two', 'three', 'four', 'five'], dtype='object')

>>> df.columns

Index(['year', 'state', 'pop', 'debt'], dtype='object')

>>> type(df['debt'])

<class 'pandas.core.series.Series'>

DataFrame 面向行和面向列的操作基本是平衡的，任意抽出一列都是 Series。

对象属性

###查找索引 查找某个值在数组中的索引，类似于 Python 内建的 list.index(value) 方法。可以通过布尔索引来实现。比如我们想在一个 Series 中寻找到 ‘c’：

lang:python

>>> ser = Series(list('abcdefg'))

>>> ser[ser='c']

2 c

dtype: object

Series 中还有一对 ser.idxmax() 和 ser.idxmin() 方法，可以返回数组中最大（小）值的索引值，或者 .argmin() 和 .argmax() 返回索引位置。当然这两类方法也是可以通过上面这种 ser[ser=ser.max()] 来替代实现。修改索引，数组的 index 属性时是不可变的，因此所谓修改索引，其实操作的是一个使用了新索引的新数组，并继承旧数据。

obj.set\_index(keys, drop=True, append=False, inplace=False, verify\_integrity=False) 方法接受一个新索引（key）并返回一个新数组。这个 key 的值可以是序列类型，也可以是调用者的一个列名，即将某一列设为新数组的索引。

lang:python

>>> indexed\_df = df.set\_index(['A', 'B'])

>>> indexed\_df2 = df.set\_index(['A', [0, 1, 2, 0, 1, 2]])

>>> indexed\_df3 = df.set\_index('column1')

###重新索引

Series 对象的重新索引通过其 `.reindex(index=None,\*\*kwargs)` 方法实现。`\*\*kwargs` 中常用的参数有俩：`method=None,fill\_value=np.NaN`：

lang:python

ser = Series([4.5,7.2,-5.3,3.6],index=['d','b','a','c'])

>>> a = ['a','b','c','d','e']

>>> ser.reindex(a)

a -5.3

b 7.2

c 3.6

d 4.5

e NaN

dtype: float64

>>>ser.reindex(a,fill\_value=0)

a -5.3

b 7.2

c 3.6

d 4.5

e 0.0

dtype: float64

>>> ser.reindex(a,method='ffill')

a -5.3

b 7.2

c 3.6

d 4.5

e 4.5

dtype: float64

>>> ser.reindex(a,fill\_value=0,method='ffill')

a -5.3

b 7.2

c 3.6

d 4.5

e 4.5

dtype: float64

.reindex() 方法会返回一个新对象，其 index 严格遵循给出的参数，method:{'backfill', 'bfill', 'pad', 'ffill', None}参数用于指定插值（填充）方式，当没有给出时，自动用 fill\_value 填充，默认为 NaN（ffill = pad，bfill = back fill，分别指插值时向前还是向后取值）

DataFrame 对象的重新索引方法为：.reindex(index=None,columns=None,\*\*kwargs)。仅比 Series 多了一个可选的 columns 参数，用于给列索引。用法与上例类似，只不过插值方法 method 参数只能应用于**行**，即轴 0。

lang:python

>>> state = ['Texas','Utha','California']

>>> df.reindex(columns=state,method='ffill')

Texas Utha California

a 1NaN2

c 4NaN5

d 7NaN8

[3 rows x 3 columns]

>>> df.reindex(index=['a','b','c','d'],columns=state,method='ffill')

Texas Utha California

a 1NaN2

b 1NaN2

c 4NaN5

d 7NaN8

[4 rows x 3 columns]

不过 fill\_value 依然对有效。聪明的小伙伴可能已经想到了，可不可以通过 df.T.reindex(index,method='\*\*').T 这样的方式来实现在列上的插值呢，答案是可行的。另外要注意，使用 reindex(index,method='\*\*') 的时候，index 必须是**单调**的，否则就会引发一个 ValueError: Must be monotonic for forward fill，比如上例中的最后一次调用，如果使用 index=['a','b','d','c'] 的话就不行。

###删除指定轴上的项 即删除 Series 的元素或 DataFrame 的某一行（列）的意思，通过对象的 .drop(labels, axis=0) 方法：

lang:python

>>> ser

d 4.5

b 7.2

a -5.3

c 3.6

dtype: float64

>>> df

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

[3 rows x 3 columns]

>>> ser.drop('c')

d 4.5

b 7.2

a -5.3

dtype: float64

>>> df.drop('a')

Ohio Texas California

c 3 4 5

d 6 7 8

[2 rows x 3 columns]

>>> df.drop(['Ohio','Texas'],axis=1)

California

a 2

c 5

d 8

[3 rows x 1 columns]

.drop() 返回的是一个新对象，元对象不会被改变。

###索引和切片

就像 Numpy，pandas 也支持通过 obj[::] 的方式进行索引和切片，以及通过布尔型数组进行过滤。

不过须要注意，因为 pandas 对象的 index 不限于整数，所以当使用**非整数**作为切片索引时，它是**末端包含**的。

lang:python

>>>foo

a 4.5

b 7.2

c-5.3

d 3.6

dtype: float64

>>>bar

0 4.5

1 7.2

2 -5.3

3 3.6

dtype: float64

>>>foo[:2]

a 4.5

b 7.2

dtype: float64

>>>bar[:2]

0 4.5

1 7.2

dtype: float64

>>>foo[:'c']

a 4.5

b 7.2

c-5.3

dtype: float64

这里 foo 和 bar 只有 index 不同——bar 的 index 是整数序列。可见当使用整数索引切片时，结果与 Python 列表或 Numpy 的默认状况相同；换成 'c' 这样的字符串索引时，结果就包含了这个边界元素。

另外一个特别之处在于 DataFrame 对象的索引方式，因为他有两个轴向（双重索引）。

可以这么理解：DataFrame 对象的标准切片语法为：.ix[::,::]。ix 对象可以接受两套切片，分别为行（axis=0）和列（axis=1）的方向：

lang:python

>>> df

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

[3 rows x 3 columns]

>>> df.ix[:2,:2]

Ohio Texas

a 0 1

c 3 4

[2 rows x 2 columns]

>>> df.ix['a','Ohio']

0

而不使用 ix ，直接切的情况就特殊了：

* 索引时，选取的是列
* 切片时，选取的是行

这看起来有点不合逻辑，但作者解释说 “这种语法设定来源于实践”，我们信他。

lang:python

>>> df['Ohio']

a 0

c 3

d 6

Name: Ohio, dtype: int32

>>> df[:'c']

Ohio Texas California

a 0 1 2

c 3 4 5

[2 rows x 3 columns]

>>> df[:2]

Ohio Texas California

a 0 1 2

c 3 4 5

[2 rows x 3 columns]

还有一种特殊情况是：假如有这样一个索引 index([2,4,5]) ，当我们使用 ser[2] 索引的时候，到底会被解释为第一个索引还是第三个索引呢？

答案是第一个索引，即当你的数组 index 是整数类型的时候，你使用整数索引，都会被自动解释为基于标签的索引，而不是基于位置的索引。要想消除这种歧义，可以使用

* .loc[label] 这是严格基于标签的索引
* .iloc[inte] 这是严格基于整数位置的索引

.ix[] 更像是这两种严格方式的智能整合版。

使用布尔型数组的情况，注意行与列的不同切法（列切法的 : 不能省）：

lang:python

>>>df['Texas']>=4

a False

c True

d True

Name: Texas, dtype: bool

>>>df[df['Texas']>=4]

Ohio Texas California

c 3 4 5

d 6 7 8

[2 rows x 3 columns]

>>> df.ix[:,df.ix['c']>=4]

Texas California

a 1 2

c 4 5

d 7 8

[3 rows x 2 columns]

###算术运算和数据对齐

pandas 最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时，结果的索引取索引对的“并集”。自动的数据对齐在不重叠的索引处引入空值，默认为 NaN。

lang:python

>>> foo = Series({'a':1,'b':2})

>>> foo

a 1

b 2

dtype: int64

>>> bar = Series({'b':3,'d':4})

>>> bar

b 3

d 4

dtype: int64

>>> foo + bar

a NaN

b 5

d NaN

dtype: float64

DataFrame 的对齐操作会同时发生在行和列上。

当不希望在运算结果中出现 NA 值时，可以使用前面 reindex 中提到过的fill\_value参数，不过为了传递这个参数，就需要使用对象的方法，而不是操作符：df1.add(df2,fill\_value=0)。其他算术方法还有：sub(), div(), mul()。

Series 和 DataFrame 之间的算术运算涉及广播，暂时先不讲。

###函数应用和映射

Numpy 的 ufuncs（元素级数组方法）也可用于操作 pandas 对象。

当希望将函数应用到 DataFrame 对象的某一行或列时，可以使用 .apply(func, axis=0, args=(), \*\*kwds) 方法。

lang:python

f = lambda x:x.max()-x.min()

>>> df

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

[3 rows x 3 columns]

>>> df.apply(f)

Ohio 6

Texas 6

California 6

dtype: int64

>>> df.apply(f,axis=1)

a 2

c 2

d 2

dtype: int64

###排序和排名

Series 的sort\_index(ascending=True)`方法可以对 index 进行排序操作，ascending 参数用于控制升序或降序，默认为升序。

若要按值对 Series 进行排序，当使用.order(na\_last=True, ascending=True, kind='mergesort')方法，任何缺失值默认都会被放到 Series 的末尾。

在 DataFrame 上，.sort\_index(axis=0, by=None, ascending=True)方法多了一个轴向的选择参数与一个 by 参数，by 参数的作用是针对某一（些）**列**进行排序（不能对行使用 by 参数）：

lang:python

>>> df.sort\_index(by='Ohio')

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

[3 rows x 3 columns]

>>> df.sort\_index(by=['California','Texas'])

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

[3 rows x 3 columns]

>>> df.sort\_index(axis=1)

California Ohio Texas

a 2 0 1

c 5 3 4

d 8 6 7

[3 rows x 3 columns]

排名（Series.rank(method='average', ascending=True)）的作用与排序的不同之处在于，他会把对象的 values 替换成名次（从 1 到 n）。这时唯一的问题在于如何处理平级项，方法里的 method 参数就是起这个作用的，他有四个值可选：average, min, max, first。

lang:python

>>> ser=Series([3,2,0,3],index=list('abcd'))

>>> ser

a 3

b 2

c 0

d 3

dtype: int64

>>> ser.rank()

a 3.5

b 2.0

c 1.0

d 3.5

dtype: float64

>>> ser.rank(method='min')

a 3

b 2

c 1

d 3

dtype: float64

>>> ser.rank(method='max')

a 4

b 2

c 1

d 4

dtype: float64

>>> ser.rank(method='first')

a 3

b 2

c 1

d 4

dtype: float64

注意在 ser[0]=ser[3] 这对平级项上，不同 method 参数表现出的不同名次。

DataFrame的.rank(axis=0, method='average', ascending=True)方法多了个 axis 参数，可选择按行或列分别进行排名，暂时好像没有针对全部元素的排名方法。

###统计方法

pandas 对象有一些统计方法。它们大部分都属于约简和汇总统计，用于从 Series 中提取单个值，或从 DataFrame 的行或列中提取一个 Series。

比如DataFrame.mean(axis=0,skipna=True)方法，当数据集中存在 NA 值时，这些值会被简单跳过，除非整个切片（行或列）全是 NA，如果不想这样，则可以通过 skipna=False 来禁用此功能：

lang:python

>>> df

one two

a 1.40 NaN

b 7.10 -4.5

c NaN NaN

d 0.75 -1.3

[4 rows x 2 columns]

>>> df.mean()

one 3.083333

two -2.900000

dtype: float64

>>> df.mean(axis=1)

a 1.400

b 1.300

c NaN

d -0.275

dtype: float64

>>> df.mean(axis=1,skipna=False)

a NaN

b 1.300

c NaN

d -0.275

dtype: float64

其他常用的统计方法有：

count：非 NA 值的数量

describe：针对 Series 或 DF 的列计算汇总统计

min , max：最小值和最大值

argmin , argmax：最小值和最大值的索引位置（整数）

idxmin , idxmax：最小值和最大值的索引值

quantile：样本分位数（0 到 1）

sum：求和

mean：均值

median：中位数

mad：根据均值计算平均绝对离差

var：方差

std：标准差

skew：样本值的偏度（三阶矩）

kurt：样本值的峰度（四阶矩）

cumsum：样本值的累计和

cummin , cummax：样本值的累计最大值和累计最小值

cumprod：样本值的累计积

diff：计算一阶差分（对时间序列很有用）

pct\_change：计算百分数变化

###协方差与相关系数

Series 有两个方法可以计算协方差与相关系数，方法的主要参数都是另一个 Series。DataFrame 的这两个方法会对“列”进行两两运算，并返回一个 len(columns) 大小的方阵：

* .corr(other, method='pearson', min\_periods=1) 相关系数，默认皮尔森
* .cov(other, min\_periods=None) 协方差

min\_periods 参数为样本量的下限，低于此值的不进行运算。 <br /> ###列与 Index 间的转换 DataFrame 的 .set\_index(keys, drop=True, append=False, verify\_integrity=False) 方法会将其一个或多个列转换为行索引，并返回一个新对象。默认 drop=True表示转换后会删除那些已经变成行索引的列。另一个 .reset\_index() 方法的作用正相反，会把已经层次化的索引转换回列里面。

lang:python

>>> df = DataFrame(np.arange(8).reshape(4,2),columns=['a','b'])

>>> df

a b

0 0 1

1 2 3

2 4 5

3 6 7

[4 rows x 2 columns]

>>> df2 = df.set\_index('a')

>>> df2

b

a

0 1

2 3

4 5

6 7

[4 rows x 1 columns]

>>> df2.reset\_index()

a b

0 0 1

1 2 3

2 4 5

3 6 7

[4 rows x 2 columns]

#处理缺失数据

pandas 中 NA 的主要表现为 np.nan，另外 Python 内建的 None 也会被当做 NA 处理。

处理 NA 的方法有四种：dropna , fillna , isnull , notnull 。

###is(not)null

这一对方法对对象做元素级应用，然后返回一个布尔型数组，一般可用于布尔型索引。

###dropna

对于一个 Series，dropna 返回一个仅含非空数据和索引值的 Series。

问题在于对 DataFrame 的处理方式，因为一旦 drop 的话，至少要丢掉一行（列）。这里的解决方式与前面类似，还是通过一个额外的参数：dropna(axis=0, how='any', thresh=None) ，how 参数可选的值为 any 或者 all。all 仅在切片元素全为 NA 时才抛弃该行(列)。另外一个有趣的参数是 thresh，该参数的类型为整数，它的作用是，比如 thresh=3，会在一行中至少有 3 个**非 NA** 值时将其保留。

###fillna

fillna(value=None, method=None, axis=0)中的 value 参数除了基本类型外，还可以使用字典，这样可以实现对不同的列填充不同的值。method 的用法与前面.reindex()方法相同，这里不再赘述。

#inplace 参数

前面有个点一直没讲，结果整篇示例写下来发现还挺重要的。就是 Series 和 DataFrame 对象的方法中，凡是会对数组作出修改并返回一个新数组的，往往都有一个 replace=False 的可选参数。如果手动设定为 True，那么原数组就可以被替换。

#层次化索引

层次化索引（hierarchical indexing）是 pandas 的一项重要功能，它允许你在一个轴上拥有多个索引级别。换句话说，一个使用了层次化的索引的二维数组，可以存储和处理三维以上的数据。

lang:python

>>> hdf = DataFrame(np.arange(8).reshape(4,2),index=[['sh','sh','sz','sz'],['600000','600001','000001','000002']],columns=['open','close'])

>>> hdf

open close

sh 60000001

60000123

sz 00000145

00000267

[4 rows x 2 columns]

>>> hdf.index

MultiIndex(levels=[['sh', 'sz'], ['000001', '000002', '600000', '600001']],

labels=[[0, 0, 1, 1], [2, 3, 0, 1]])

上例中原本sh和sz已经是第三维的索引了，但使用层次化索引后，可以将整个数据集控制在二维表结构中。这对于数据重塑和基于分组的操作（如生成透视表）比较重要。

索引或层次化索引对象（Index 与 MultiIndex）都有一个names属性，可以用来给索引层次命名，以便索引和增加直观性。对 names 属性的操作可以直接通过obj.index.names=[]的形式来实现。

在这里总结一下；

1、对某一行进行求和： Row\_sum = df.iloc[i,0：].sum()

2、对某一列进行求和：column\_sum = df.iloc[:,j].sum()

3、对每一列进行求和：

        for i in df.columns:

            print(df[i].sum())

习惯上，我们会按下面格式引入所需要的包：



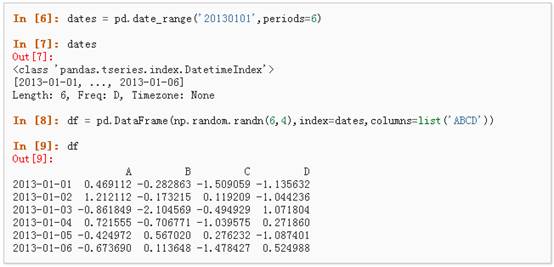
## 1、创建对象

可以通过 [Data Structure Intro Setion](http://pandas.pydata.org/pandas-docs/stable/dsintro.html#dsintro) 来查看有关该节内容的详细信息。

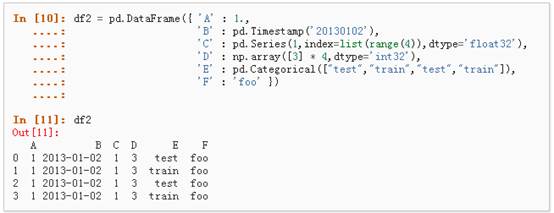
1、可以通过传递一个list对象来创建一个Series，pandas会默认创建整型索引：



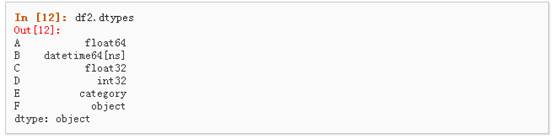
2、通过传递一个numpy array，时间索引以及列标签来创建一个DataFrame：



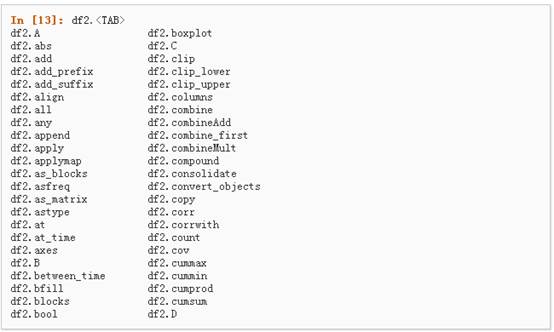
3、通过传递一个能够被转换成类似序列结构的字典对象来创建一个DataFrame：



4、查看不同列的数据类型：



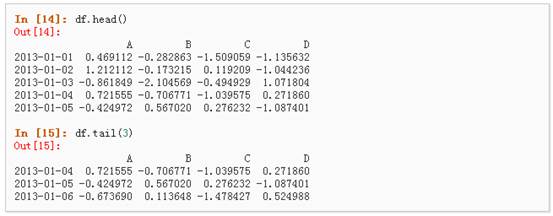
5、如果你使用的是IPython，使用Tab自动补全功能会自动识别所有的属性以及自定义的列，下图中是所有能够被自动识别的属性的一个子集：



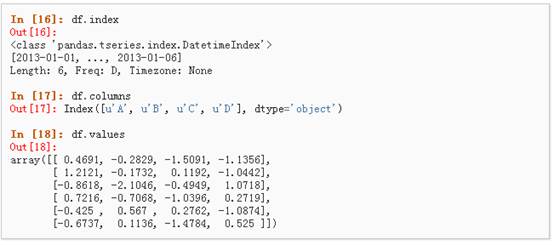
## 2、查看数据

详情请参阅：[Basics Section](http://pandas.pydata.org/pandas-docs/stable/basics.html#basics)

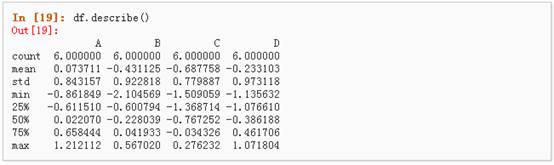
1、  查看frame中头部和尾部的行：



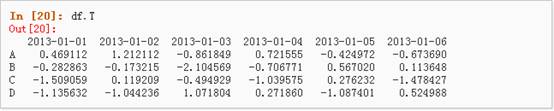
2、  显示索引、列和底层的numpy数据：



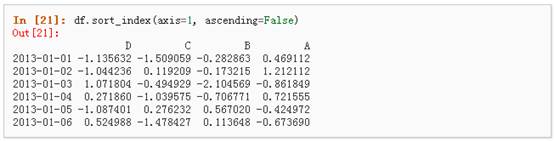
3、  describe()函数对于数据的快速统计汇总：



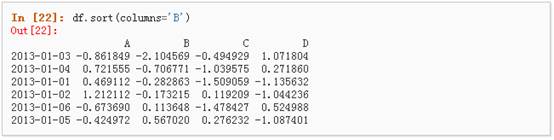
4、  对数据的转置：



5、  按轴进行排序



6、  按值进行排序

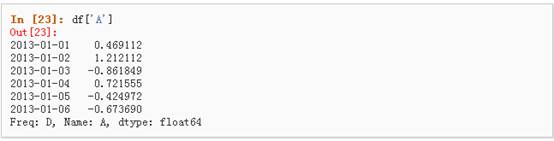


## 3、选择

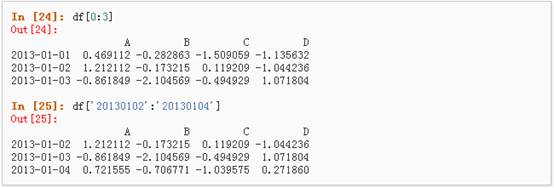
虽然标准的Python/Numpy的选择和设置表达式都能够直接派上用场，但是作为工程使用的代码，我们推荐使用经过优化的pandas数据访问方式：.at, .iat, .loc, .iloc 和 .ix详情请参阅[Indexing and Selecing Data](http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing) 和 [MultiIndex / Advanced Indexing](http://pandas.pydata.org/pandas-docs/stable/advanced.html#advanced)。

  获取

1、 选择一个单独的列，这将会返回一个Series，等同于df.A：



2、 通过[]进行选择，这将会对行进行切片

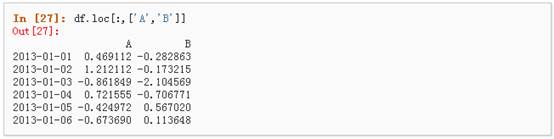


  通过标签选择

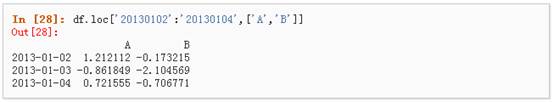
1、 使用标签来获取一个交叉的区域



2、 通过标签来在多个轴上进行选择



3、 标签切片



4、 对于返回的对象进行维度缩减



5、 获取一个标量

https://images0.cnblogs.com/blog/407700/201412/091345152121400.jpg

6、 快速访问一个标量（与上一个方法等价）

https://images0.cnblogs.com/blog/407700/201412/091345156038112.jpg

  通过位置选择

1、 通过传递数值进行位置选择（选择的是行）



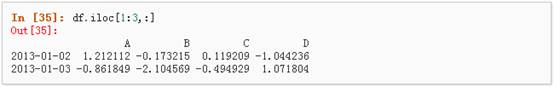
2、 通过数值进行切片，与numpy/python中的情况类似



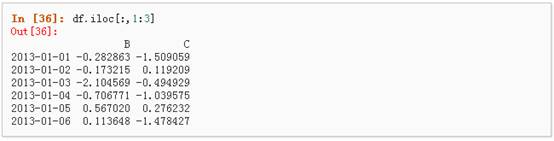
3、 通过指定一个位置的列表，与numpy/python中的情况类似



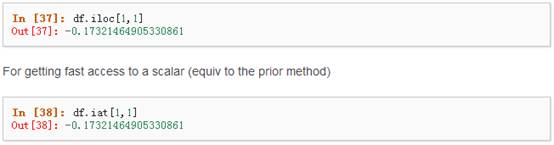
4、 对行进行切片



5、 对列进行切片

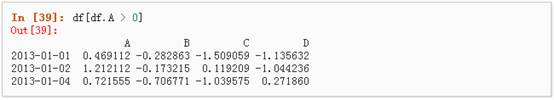


6、 获取特定的值

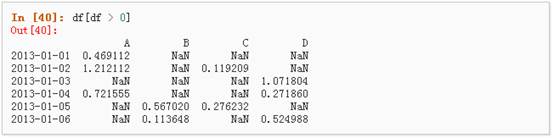


  布尔索引

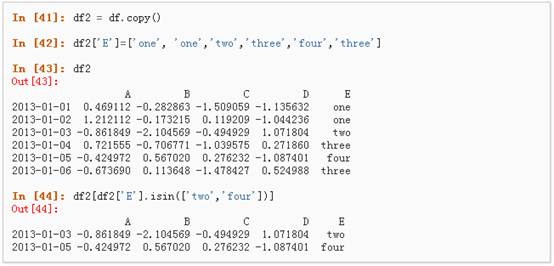
1、 使用一个单独列的值来选择数据：



2、 使用where操作来选择数据：

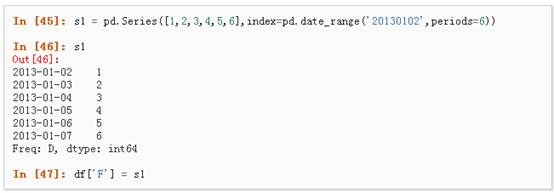


3、 使用isin()方法来过滤：



  设置

1、 设置一个新的列：



2、 通过标签设置新的值：

https://images0.cnblogs.com/blog/407700/201412/091345207902196.jpg

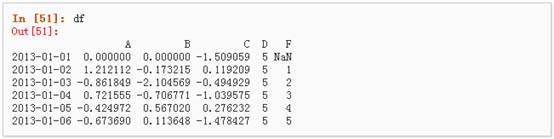
3、 通过位置设置新的值：

https://images0.cnblogs.com/blog/407700/201412/091345215409566.jpg

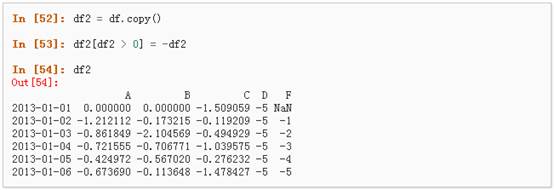
4、 通过一个numpy数组设置一组新值：

https://images0.cnblogs.com/blog/407700/201412/091345224314895.jpg

上述操作结果如下：



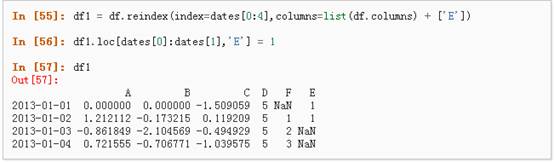
5、 通过where操作来设置新的值：



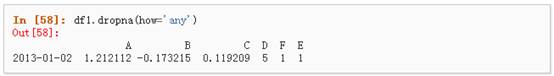
## 4、缺失值处理

在pandas中，使用np.nan来代替缺失值，这些值将默认不会包含在计算中，详情请参阅：[Missing Data Section](http://pandas.pydata.org/pandas-docs/stable/missing_data.html#missing-data)。

1、  reindex()方法可以对指定轴上的索引进行改变/增加/删除操作，这将返回原始数据的一个拷贝：、



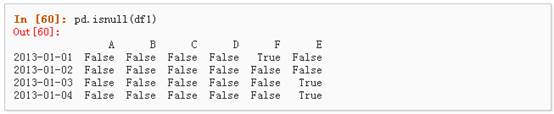
2、  去掉包含缺失值的行：



3、  对缺失值进行填充：



4、  对数据进行布尔填充：

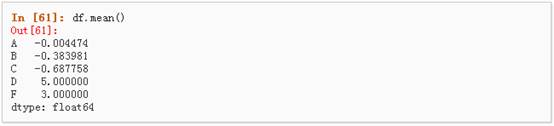


## 5、相关操作

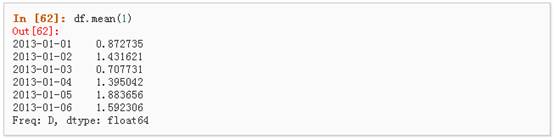
详情请参与 [Basic Section On Binary Ops](http://pandas.pydata.org/pandas-docs/stable/basics.html#basics-binop)

  统计（相关操作通常情况下不包括缺失值）

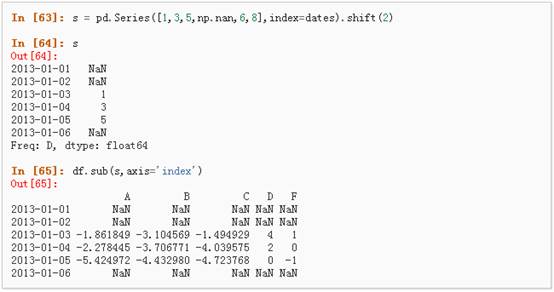
1、  执行描述性统计：



2、  在其他轴上进行相同的操作：

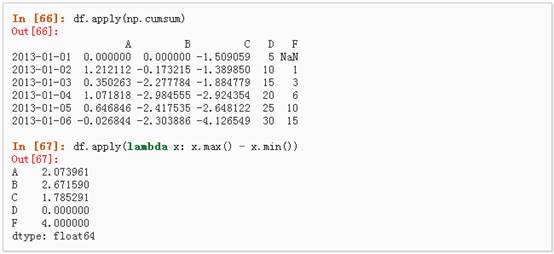


3、  对于拥有不同维度，需要对齐的对象进行操作。Pandas会自动的沿着指定的维度进行广播：



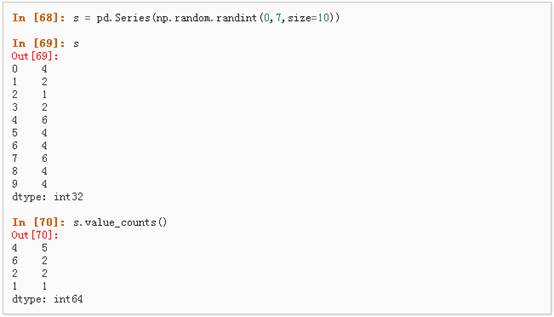
  Apply

1、  对数据应用函数：



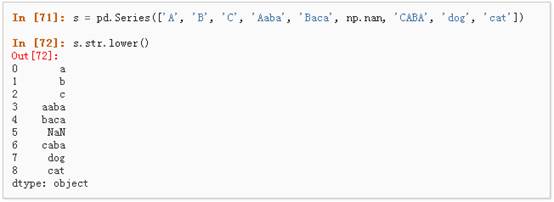
  直方图

具体请参照：[*Histogramming and Discretization*](http://pandas.pydata.org/pandas-docs/stable/basics.html#basics-discretization)



  字符串方法

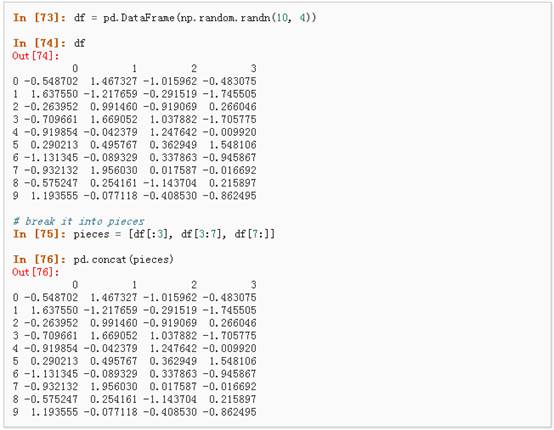
Series对象在其str属性中配备了一组字符串处理方法，可以很容易的应用到数组中的每个元素，如下段代码所示。更多详情请参考：[*Vectorized String Methods*](http://pandas.pydata.org/pandas-docs/stable/text.html#text-string-methods).



## 6、合并

Pandas提供了大量的方法能够轻松的对Series，DataFrame和Panel对象进行各种符合各种逻辑关系的合并操作。具体请参阅：[*Merging section*](http://pandas.pydata.org/pandas-docs/stable/merging.html#merging)

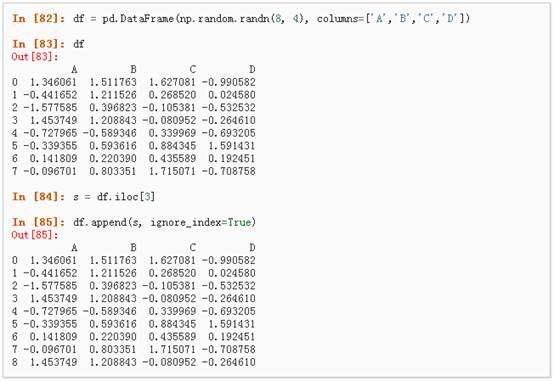
  Concat



  Join 类似于SQL类型的合并，具体请参阅：[*Database style joining*](http://pandas.pydata.org/pandas-docs/stable/merging.html#merging-join)



  Append 将一行连接到一个DataFrame上，具体请参阅[*Appending*](http://pandas.pydata.org/pandas-docs/stable/merging.html#merging-concatenation)：



## 7、分组

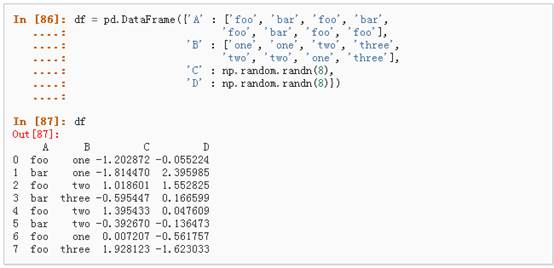
对于”group by”操作，我们通常是指以下一个或多个操作步骤：

  （Splitting）按照一些规则将数据分为不同的组；

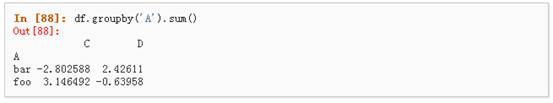
  （Applying）对于每组数据分别执行一个函数；

  （Combining）将结果组合到一个数据结构中；

详情请参阅：[*Grouping section*](http://pandas.pydata.org/pandas-docs/stable/groupby.html#groupby)



1、  分组并对每个分组执行sum函数：



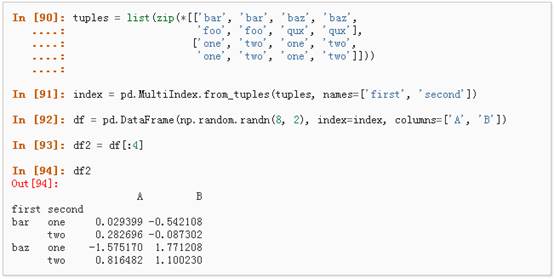
2、  通过多个列进行分组形成一个层次索引，然后执行函数：

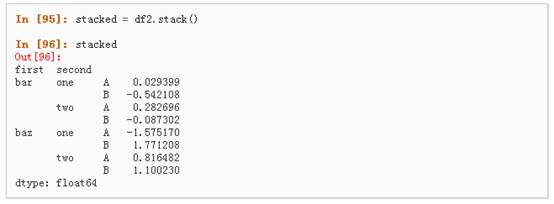


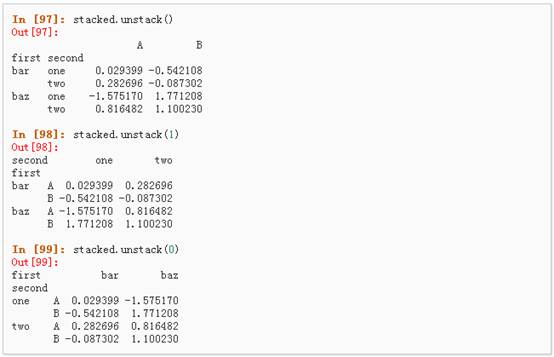
## 8、Reshaping

详情请参阅 [*Hierarchical Indexing*](http://pandas.pydata.org/pandas-docs/stable/advanced.html#advanced-hierarchical) 和 [*Reshaping*](http://pandas.pydata.org/pandas-docs/stable/reshaping.html#reshaping-stacking)。

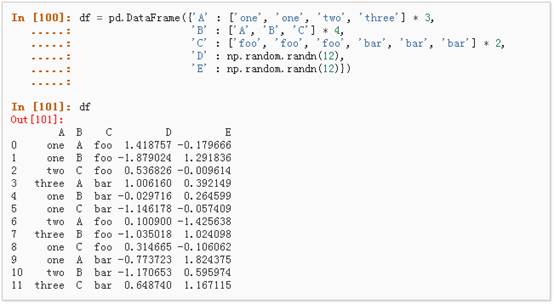
  Stack



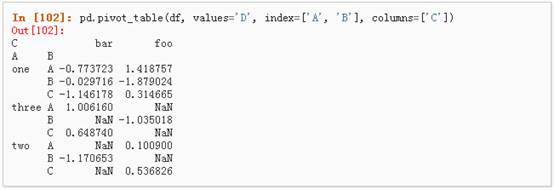




  数据透视表，详情请参阅：[*Pivot Tables*](http://pandas.pydata.org/pandas-docs/stable/reshaping.html#reshaping-pivot).

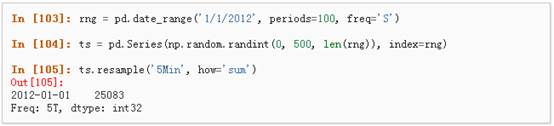


可以从这个数据中轻松的生成数据透视表：

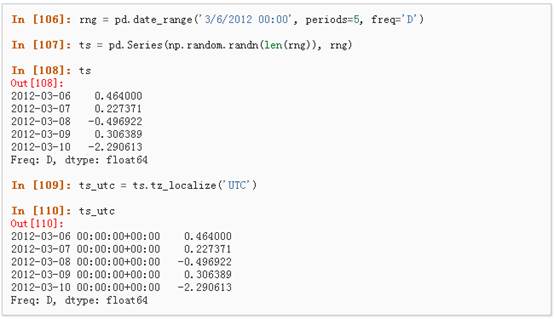


## 9、时间序列

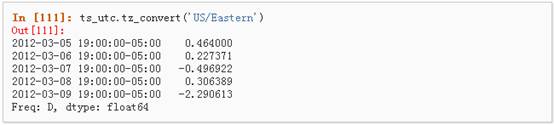
Pandas在对频率转换进行重新采样时拥有简单、强大且高效的功能（如将按秒采样的数据转换为按5分钟为单位进行采样的数据）。这种操作在金融领域非常常见。具体参考：[*Time Series section*](http://pandas.pydata.org/pandas-docs/stable/timeseries.html#timeseries)。



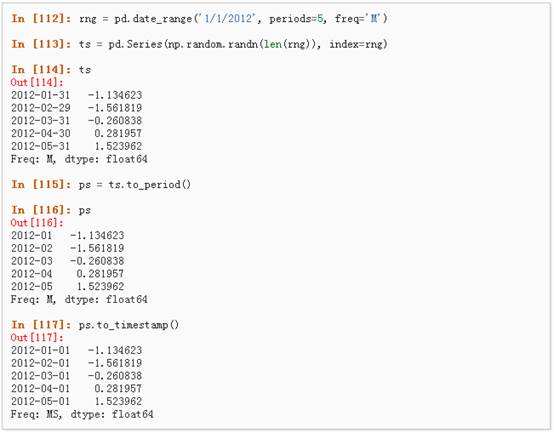
1、  时区表示：



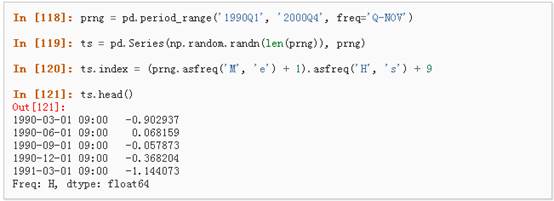
2、  时区转换：



3、  时间跨度转换：



4、  时期和时间戳之间的转换使得可以使用一些方便的算术函数。



## 10、Categorical

从0.15版本开始，pandas可以在DataFrame中支持Categorical类型的数据，详细 介绍参看：[*categorical introduction*](http://pandas.pydata.org/pandas-docs/stable/categorical.html#categorical)和[*API documentation*](http://pandas.pydata.org/pandas-docs/stable/api.html#api-categorical)。

https://images0.cnblogs.com/blog/407700/201412/091345345093572.jpg

1、  将原始的grade转换为Categorical数据类型：



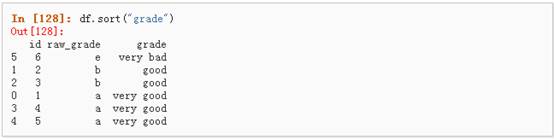
2、  将Categorical类型数据重命名为更有意义的名称：

https://images0.cnblogs.com/blog/407700/201412/091345356033669.jpg

3、  对类别进行重新排序，增加缺失的类别：



4、  排序是按照Categorical的顺序进行的而不是按照字典顺序进行：

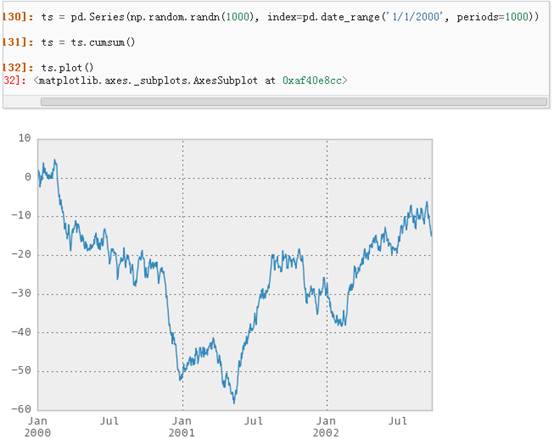


5、  对Categorical列进行排序时存在空的类别：

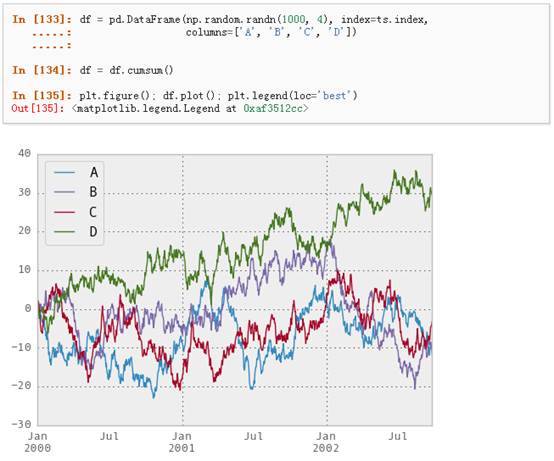


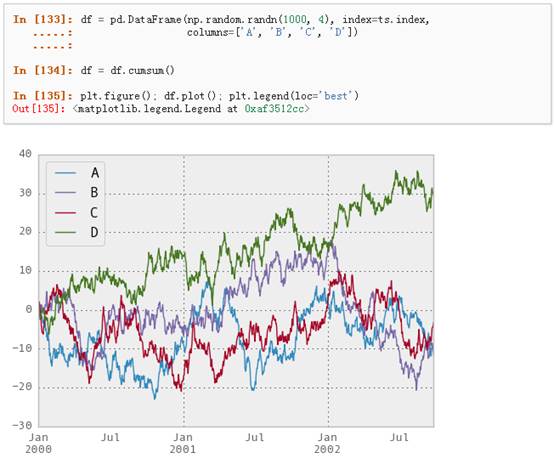
## 11、画图

具体文档参看：[*Plotting*](http://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization) docs



对于DataFrame来说，plot是一种将所有列及其标签进行绘制的简便方法：

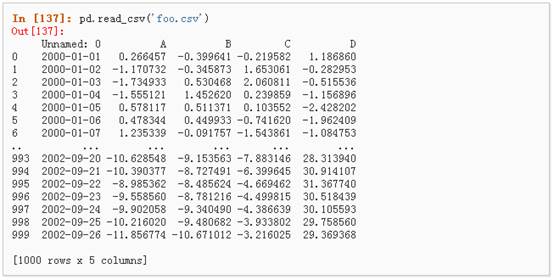




## 12、导入和保存数据

  CSV，参考：[*Writing to a csv file*](http://pandas.pydata.org/pandas-docs/stable/io.html#io-store-in-csv)

1、  从csv文件中读取：



2、  写入csv文件：

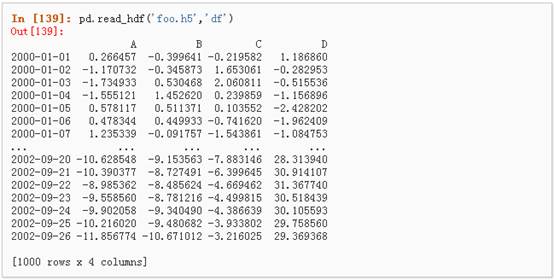
https://images0.cnblogs.com/blog/407700/201412/091345399933395.jpg

HDF5，参考：[*HDFStores*](http://pandas.pydata.org/pandas-docs/stable/io.html#io-hdf5)

1、  写入HDF5存储：

https://images0.cnblogs.com/blog/407700/201412/091345410091453.jpg

2、  从HDF5存储中读取：

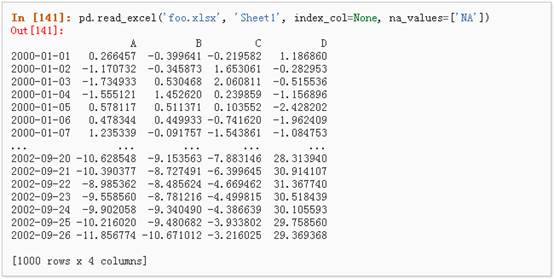


  Excel，参考：[*MS Excel*](http://pandas.pydata.org/pandas-docs/stable/io.html#io-excel)

1、  写入excel文件：

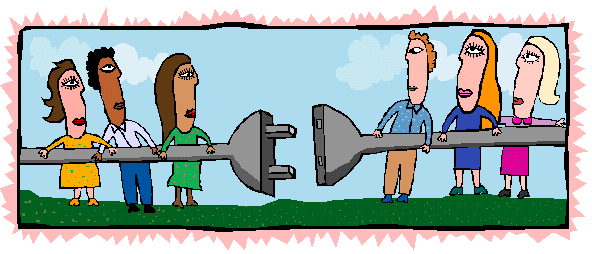
https://images0.cnblogs.com/blog/407700/201412/091345417757350.jpg

2、  从excel文件中读取：

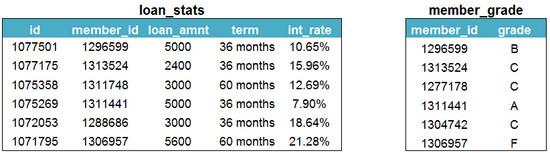


## 13、使用pandas进行数据匹配

Pandas中的merge函数类似于Excel中的Vlookup，可以实现对两个数据表进行匹配和拼接的功能。与Excel不同之处在于merge函数有4种匹配拼接模式，分别为inner，left，right和outer模式。 其中inner为默认的匹配模式。本篇文章我们将介绍merge函数的使用方法和4种拼接模式的区别。



下面是我们准备进行拼接的两个数据表，左边是贷款状态表loan\_stats，右边为用户等级表member\_grade。我们将分别用merge函数的4种匹配模式对这两个表进行拼接。



准备工作

开始使用merge函数进行数据拼接之前先导入所需的功能库，然后将分别读取两个数据表，并命名为loanstats表和member\_grade表。

**import** numpy **as** np

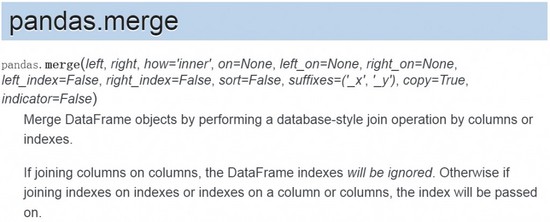
**import** pandas **as** pd

loanstats=pd.DataFrame(pd.read\_excel('loanStats.xlsx'))

member\_grade=pd.DataFrame(pd.read\_excel('member\_grade.xlsx'))

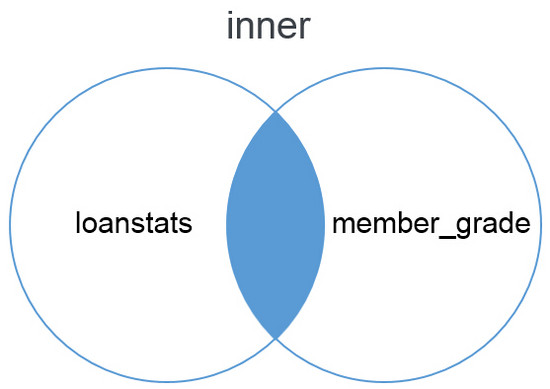
函数功能介绍

merge函数的使用方法很简单，以下是官方的函数功能介绍和使用说明。merge函数中第一个出现的数据表是拼接后的left部分，第二个出现的数据表是拼接后的right部分。第三个是数据匹配模 式，默认是inner模式。第四个参数on表示数据匹配所依据的字段名称，如果这个字段名称同时出现在两个数据表中，那么可以省略on参数的设置，merge默认会按照两个数据表中共有的字段名称进行匹配和拼接。如果两个数据表中的匹配字段名称不一致，则需要分别在left\_on和right\_on参数中指明两个表匹配字段的名称。如果两个数据表中没有匹配字段，需要使用索引列进行匹配和拼接，可以对left\_index和right\_index参数设置为True。merge还有一些排序和其他的参数，可在需要使用时进行设置。



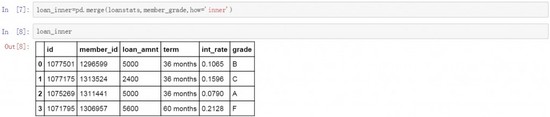
Inner模式匹配

inner模式是merge的默认匹配模式，我们通过下面的文氏图来说明inner的匹配方法。Inner模式提供在loanstats和member\_grade表中共有字段的匹配结果。也就是对两个的表交集部分进行匹配和拼接。单独只出现在一个表中的字段值不会参与匹配和拼接。



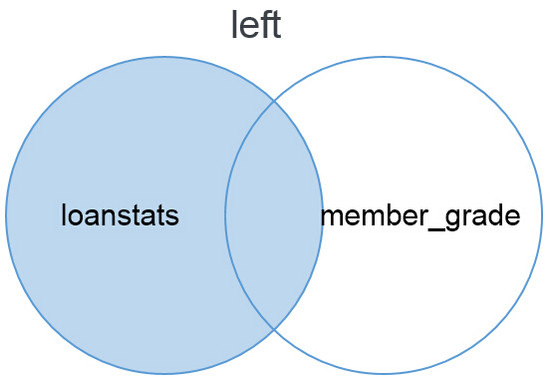
以下是使用merge函数进行拼接的代码，因为inner是默认的拼接模式，因此也可以省略how=’inner’部分。其中第一个出现的loanstats出现在拼接后的左侧，member\_grade出现在拼接后的右侧。拼接后的数据表中只包含两个表的交集，因此不存在未匹配到的NaN情况。

loan\_inner=pd.merge(loanstats,member\_grade,how='inner')



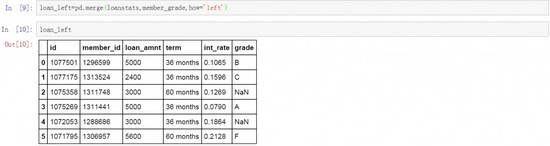
left模式匹配

left模式是左匹配，以左边的数据表loanstats为基础匹配右边的数据表member\_grade中的内容。匹配不到的内容以NaN值显示。在Excel中就好像将Vlookup公式写在了左边的表中。下面的文氏图说明了left模式的匹配方法。Left模式匹配的结果显示了所有左边数据表的内容，以及和右边数据表共有的内容。



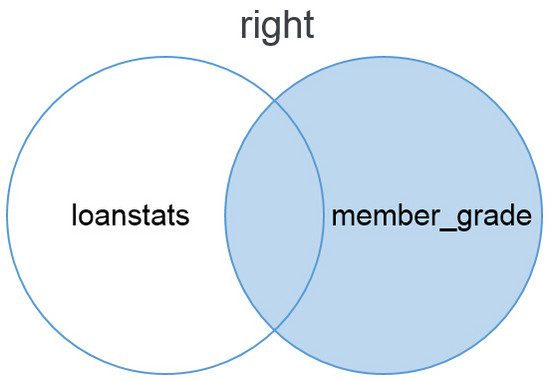
以下为使用left模式匹配并拼接后的结果，loanstats在merge函数中第一个出现，因此为左表，member\_grade第二个出现，为右表。匹配模式为left模式。从结果中可以看出left匹配模式保留了一张完整的loanstats表，以此为基础对member\_grade表中的内容进行匹配。loanstats表中有两个member\_id值在member\_grade中无法找到，因此grades字段显示为NaN值。

loan\_left=pd.merge(loanstats,member\_grade,how='left')



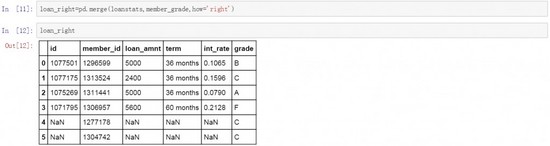
right模式匹配

第三种模式是right匹配，right与left模式正好相反，right模式是右匹配，以右边的数据表member\_grade为基础匹配左边的数据表loanstats。匹配不到的内容以NaN值显示。下面通过文氏图说明right模式的匹配方法。Right模式匹配的结果显示了所有右边数据表的内容，以及和左边数据表共有的内容。



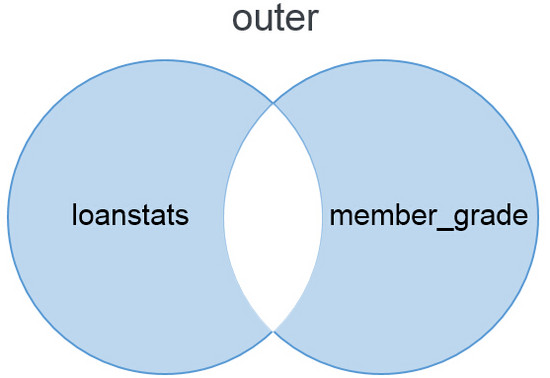
以下为使用right模式匹配拼接的结果，从结果表中可以看出right匹配模式保留了完整的member\_grade表，以此为基础对loanstats表进行匹配，在member\_grade数据表中有两个条目在loanstats数据表中无法找到，因此显示为了NaN值。

loan\_right=pd.merge(loanstats,member\_grade,how='right')



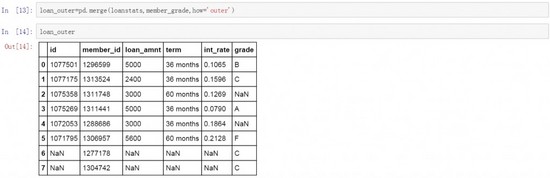
outer模式匹配

最后一种模式是outer匹配，outer模式是两个表的汇总，将loanstats和member\_grade两个要匹配的两个表汇总在一起，生成一张汇总的唯一值数据表以及匹配结果。



下面是使用outer模式匹配拼接的结果，其中member\_id列包含了loanstats和member\_grade中的唯一值，grade列显示了对member\_grade表匹配的结果，其他列则显示了对loanstats表匹配的结果 ，无法匹配的内容以NaN值显示。

loan\_outer=pd.merge(loanstats,member\_grade,how='outer')



NaN值匹配问题

在进行数据匹配和拼接的过程中经常会遇到NaN值。这种情况下merge函数会如何处理呢？merge会将两个数据表中的NaN值进行交叉匹配拼接，换句话说就是将loanstats表member\_id列中的NaN值

分别与member\_grade表中member\_id列中的每一个NaN值进行匹配，然后再拼接在一张表中。下面是包含NaN值的两张数据表进行拼接的结果，当我们使用left模式进行匹配时，loanstats作为基础

表，其中member\_id列的NaN值分别与member\_grade表中member\_id列的每一个NaN值进行匹配。并将匹配结果显示在了结果表中。

loan\_left=pd.merge(loanstats,member\_grade,how='left')

