Pandas 实例

目录

- 使用**pandas**进行数据清洗
  - 数据表中的重复值
  - 数据表中的空值/缺失值
  - 数据间的空格
  - 数据中的异常和极端值
  - 更改数据格式
  - 数据分组
  - 数据分列
- 使用**Pandas**对数据进行筛选和排序
  - sort()

目录

目录

- 使用**Pandas**进行数据匹配
  - merge()介绍
  - inner模式匹配
  - left模式匹配
  - right模式匹配
  - outer模式匹配
  - NaN值匹配模式
- 使用**Pandas**创建数据透视表
  - pandas.pivot_table()
  - 创建简单的数据透视表

目录

目录

目录

- 产品数据分析
- 运营数据分析
- 风控数据分析

使用pandas进行数据清洗

数据清洗是一项复杂且繁琐的工作，同时也是整个数据分析过程中最为重要的环节。有人说一个分析项目80%的时间都是在清洗数据，这听起来有些匪夷所思，但在实际的工作中确实如此。数据清洗的目的有两个，第一是通过清洗让数据可用。第二是让数据变的更适合进行后续的分析工作。换句话说就是有”脏”数据要洗，干净的数据也要洗。

```
import numpy as np
import pandas as pd
loandata=pd.DataFrame(pd.read_csv('loandata.csv'))
#安装 xlrd 模块，把loandata.xlsx表导入工作目录
```

使用pandas进行数据清洗

```
In [2]: loandata
```

Out[2]:

	member_id	loan_amnt	grade	emp_length	annual_inc	issue_d	loan_status	open_acc	total_pymnt	total_rec_int
0	1296599	5000	B-B2	10+ years	24000.00	2016-01-15	Fully Paid	3	5863.155187	863.16
1	1311748	NaN	B-B5	1 year	NaN	2016-06-16	Current	15	3581.120000	1042.85
2	1313524	2400	C-C5	10+ years	12252.00	2016-06-14	Fully Paid	2	3005.666844	605.67
3	1277178	100000	C-C1	10+ years	49200.00	2016-01-15	Fully Paid	10	12231.890000	2214.92
4	1311748	NaN	B-B5	1 year	NaN	2016-06-16	Current	15	3581.120000	1042.85
5	1311441	5000	A-A4	3 years	NaN	2016-01-15	Fully Paid	9	5632.210000	632.21
6	1304742	NaN	C-C5	8 years	47004.00	2016-05-16	Fully Paid	7	10137.840010	3137.84
7	1288686	3000	E-E1	9 years	48000.00	2016-01-15	Fully Paid	4	3939.135294	939.14
8	1306957	NaN	F-F2	4 years	NaN	2016-04-12	charged Off	11	646.020000	294.94
9	1306721	5375	B-B5	< 1 year	15000.00	2016-11-12	Charged Off	2	1476.190000	533.42
10	1305201	6500	C-C3	5 years	72000.00	2016-06-13	fully paid	14	7678.017673	1178.02
11	1305008	12000	B-B5	10+ years	75000.00	2016-09-13	Fully Paid	12	13947.989160	1947.99
12	1306957	NaN	F-F2	4 years	NaN	2016-04-12	charged Off	11	646.020000	294.94
13	1304956	3000	B-B1	3 years	15000.00	2016-01-15	Fully Paid	11	3480.269999	480.27
14	1303503	10000	B-B2	3 years	100000.00	2016-10-13	Charged Off	14	7471.990000	1393.42
15	1304871	1000	D-D1	< 1 year	28000.00	2016-01-15	fully paid	11	1270.716942	270.72
16	1299699	10000	C-C4	4 years	NaN	2016-01-15	Fully Paid	14	12527.150000	2527.15
17	1304884	36	A-A1	10+ years	110000.00	2016-05-13	Fully paid	20	3785.271965	185.27
18	1304871	1000	D-D1	< 1 year	28000.00	2016-01-15	fully paid	11	1270.716942	270.72
19	1304855	9200	A-A1	6 years	77385.19	2016-07-12	Fully Paid	8	9460.000848	260.00

使用pandas进行数据清洗

数据清洗的目的有两个，第一是通过清洗让脏数据变的可用。这也是我们首先要解决的问题。无论是线下人工填写的手工表，还是线上通过工具收集到的数据，又或者是CRM系统中导出的数据。很多数据源都有一些这样或者那样的问题，例如：数据中的重复值，异常值，空值，以及多余的空格和大小写错误的问题。下面我们逐一进行处理。

使用pandas进行数据清洗

数据表中的重复值

第一个要处理的问题是数据表中的重复值，pandas中有两个函数是专门用来处理重复值的，第一个是duplicated函数。Duplicated函数用来查找并显示数据表中的重复值。下面是使用这个函数对数据表进行重复值查找后的结果。

```
loandata.duplicated()
```

使用pandas进行数据清洗

这里有两点需要说明：第一，数据表中两个条目间所有列的内容都相等时duplicated才会判断为重复值。(Duplicated也可以单独对某一列进行重复值判断)。第二，duplicated支持从前向后(first)，和从后向前(last)两种重复值查找模式。默认是从前向后进行重复值的查找和判断。换句话说就是将后出现的相同条件判断为重复值。在前面的表格中索引为4的1311748和索引为1的条目相同。默认情况下后面的条目在重复值判断中显示为True。

使用pandas进行数据清洗

Pandas中的drop_duplicates函数用来删除数据表中的重复值，判断标准和逻辑与duplicated函数一样。使用drop_duplicates函数后，python将返回一个只包含唯一值的数据表。下面是使用drop_duplicates函数后的结果。与原始数据相比减少了3行，仔细观察可以发现，drop_duplicates默认也是使用了first模式删除了索引为4的重复值，以及后面的另外两个重复值。

```
loandata.drop_duplicates()
```

使用pandas进行数据清洗

```
In [5]: loandata.drop_duplicates()
```

Out[5]:

	member_id	loan_amnt	grade	emp_length	annual_inc	issue_d	loan_status	open_acc	total_pymnt	total_rec_int
0	1296599	5000	B-B2	10+ years	24000.00	2016-01-15	Fully Paid	3	5863.155187	863.16
1	1311748	NaN	B-B5	1 year	NaN	2016-06-16	Current	15	3581.120000	1042.85
2	1313524	2400	C-C5	10+ years	12252.00	2016-06-14	Fully Paid	2	3005.666844	605.67
3	1277178	100000	C-C1	10+ years	49200.00	2016-01-15	Fully Paid	10	12231.890000	2214.92
5	1311441	5000	A-A4	3 years	NaN	2016-01-15	Fully Paid	9	5632.210000	632.21
6	1304742	NaN	C-C5	8 years	47004.00	2016-05-16	Fully Paid	7	10137.840010	3137.84
7	1288686	3000	E-E1	9 years	48000.00	2016-01-15	Fully Paid	4	3939.135294	939.14
8	1306957	NaN	F-F2	4 years	NaN	2016-04-12	charged Off	11	646.020000	294.94
9	1306721	5375	B-B5	< 1 year	15000.00	2016-11-12	Charged Off	2	1476.190000	533.42
10	1305201	6500	C-C3	5 years	72000.00	2016-06-13	fully paid	14	7678.017673	1178.02
11	1305008	12000	B-B5	10+ years	75000.00	2016-09-13	Fully Paid	12	13947.989160	1947.99
13	1304956	3000	B-B1	3 years	15000.00	2016-01-15	Fully Paid	11	3480.269999	480.27
14	1303503	10000	B-B2	3 years	100000.00	2016-10-13	Charged Off	14	7471.990000	1393.42
15	1304871	1000	D-D1	< 1 year	28000.00	2016-01-15	fully paid	11	1270.716942	270.72
16	1299699	10000	C-C4	4 years	NaN	2016-01-15	Fully Paid	14	12527.150000	2527.15
17	1304884	36	A-A1	10+ years	110000.00	2016-05-13	Fully paid	20	3785.271965	185.27
19	1304855	9200	A-A1	6 years	77385.19	2016-07-12	Fully Paid	8	9460.000848	260.00

使用pandas进行数据清洗

- 数据表中的空值/缺失值

第二个要处理的问题是数据表中的空值，在python中空值被显示为NaN。在处理空值之前先来检查下数据表中的空值数量。对于一个小的数据表，可以人工查找，但对于较为庞大的数据表，需要寻找一个更为方便快捷的方法。对关键字段进行空值查找。分别选择了对loan_amnt字段和annual_inc字段查找空值。

```
loandata.isnull()   #loandata.notnull()
loandata['loan_amnt'].isnull().value_counts()
loandata['annual_inc'].isnull().value_counts()
#通过isnull函数和value_counts函数分别获得了loan_amnt列和annual_inc列中的空值数据量。
```

使用pandas进行数据清洗

对于空值有两种处理的方法，第一种是使用fillna函数对空值进行填充，可以选择填充0值或者其他任意值。第二种方法是使用dropna函数直接将包含空值的数据删除。

```
loandata.fillna(0)
loandata.dropna()
```

这里选择对空值数据进行填充，首先处理loan_amnt列中的空值。通过totalpymnt字段和total_rec_int字段值相减计算出loan_amnt列中的近似值。因为这里除了利息以外还可能包括一些逾期费，手续费和罚息等，所以只能获得一个实际贷款金额近似值。。

使用pandas进行数据清洗

由于贷款金额通常是一个整数，因此我们在代码的最后对格式进行了转换

```
loandata['loan_amnt']=loandata['loan_amnt'].fillna(loandata['total_pymnt']-loandata['total_rec_int']).astype(np.int64)

loandata['loan_amnt']
```

对于annual_inc列，在原始数据表中没有可用的辅助列进行计算，选择用现有数据的均值进行填充。可以看到贷款用户的收入均值为50060美金。使用这个值对annual_inc的空值进行填充。

```
loandata['annual_inc']=loandata['annual_inc'].fillna(loandata['annual_inc'].mean())

loandata['annual_inc']
```

使用pandas进行数据清洗

- 数据间的空格

第三个要处理的是数据中的空格。空格会影响数据的统计和计算。从下面的结果中就可看出空格对于常规的数据统计造成的影响。

查看数据中的空格

再对loan_status列进行频率统计时，由于空格的问题，相同的贷款状态被重复计算。造成统计结果不可用。因此，需要解决字段中存在的空格问题。

```
loandata['loan_status'].value_counts()
```

使用pandas进行数据清洗

去除数据中的空格

Python中去除空格的方法有三种，第一种是去除数据两边的空格，第二种是单独去除左边的空格，第三种是单独去除右边的空格。

```
loandata['loan_status']=loandata['loan_status'].astype(np.str)
#格式更改
loandata['loan_status'] = loandata['loan_status'].map(str.strip)

loandata['loan_status'] = loandata['loan_status'].map(lstr.strip)
loandata['loan_status'] = loandata['loan_status'].map(rstr.strip)

loandata['loan_status'].value_counts()
```

使用pandas进行数据清洗

大小写转换

大小写转换的方法也有三种可以选择，分别为全部转换为大写，全部转换为小写，和转换为首字母大写。

```
loandata['loan_status']=loandata['loan_status'].map(str.upper)

loandata['loan_status']=loandata['loan_status'].map(str.lower)

loandata['loan_status']=loandata['loan_status'].map(str.title)
#loandata['loan_status']=loandata['loan_status'].astype(np.str)
```

再次进行频率统计。从下面的结果清晰的显示了贷款的三种状态出现的频率。

```
loandata['loan_status'].value_counts()
```

使用pandas进行数据清洗

第四个还需要对数据表中**关键字段的内容进行检查**，确保关键字段中内容的统一。主要包括数据是否全部为字符，或数字。下面我们对emp_length列进行检验，此列内容由数字和字符组成，如果只包括字符，说明可能存在问题。下面的代码中我们检查该列是否全部为字符。答案全部为False。

```
loandata['emp_length'].apply(lambda x: x.isalpha())
```

除此之外，还能检验该列的内容是否全部为字母或数字。或者是否全部为数字。

```
loandata['emp_length'].apply(lambda x: x. isalnum ())
loandata['emp_length'].apply(lambda x: x. isdigit ())
```

使用pandas进行数据清洗

- 数据中的异常和极端值

第五个要处理的问题是数据中的异常值和极端值，发现异常值和极端值的方法是对数据进行描述性统计。使用describe函数可以生成描述统计结果。其中我们主要关注最大值(max)和最小值(min)情况。

检查异常和极端值

下面是对数据表进行描述统计的结果，其中loan_amnt的最大值和最小值分别为100000美金和36美金，这不符合业务逻辑，因此可以判断为异常值。

```
loandata.describe().astype(np.int64).T
```

使用pandas进行数据清洗

异常数据替换

对于异常值数据这里选择使用replace函数对loan_amnt的异常值进行替换，这里替换值选择为loan_amnt的均值。下面是具体的代码和替换结果。

```
loandata.replace([100000,36],loandata['loan_amnt'].mean())
```

数据清洗的第二个目的是让数据更加适合后续的分析工作。提前对数据进行预处理，后面的挖掘和分析工作会更加高效。这些预处理包括数据格式的处理，数据分组和对有价值信息的提取。下面来介绍这部分的操作过程和使用到的函数。

使用pandas进行数据清洗

更改数据格式

第一步是更改和规范数据格式，所使用的函数是astype。对loan_amnt列中的数据，由于贷款金额通常为整数，因此数据格式改为int64。如果是利息字段，由于会有小数，因此通常设置为float64。

```
loandata['loan_amnt']=loandata['loan_amnt'].astype(np.int64)
```

在数据格式中还要特别注意日期型的数据。日期格式的数据需要使用to_datatime函数进行处理。

```
loandata['issue_d']=pd.to_datetime(loandata['issue_d'])
```

使用pandas进行数据清洗

格式更改后可以通过dtypes函数来查看，下面显示了每个字段的数据格式。

```
loandata.dtypes

member_id                int64
loan_amnt                int64
grade                   object
emp_length              object
annual_inc             float64
issue_d         datetime64[ns]
loan_status             object
open_acc                int64
total_pymnt            float64
total_rec_int          float64
dtype: object
```

使用pandas进行数据清洗

数据分组

第二步是对数据进行分组处理，在数据表的open_acc字段记录了贷款用户的账户数量，可以根据账户数量的多少对用户进行分级，5个账户以下为A级，5-10个账户为B级，依次类推。

```
bins = [0, 5, 10, 15, 20]
group_names = ['A', 'B', 'C', 'D']
loandata['categories'] = pd.cut(loandata['open_acc'], bins,
labels=group_names)
loandata
```

首先设置了数据分组的依据，然后设置每组对应的名称。最后使用cut函数对数据进行分组并将分组后的名称添加到数据表中。

使用pandas进行数据清洗

数据分列

第三步是数据分列，这个操作和Excel中的分列功能很像，在原始数据表中grade列中包含了两个层级的用户等级信息，现在我们通过数据分列将分级信息进行拆分。数据分列操作使用的是split函数，下面是具体的代码和分列后的结果

```
grade_split = pd.DataFrame((x.split('-') for x in loandata.grade),index=loandata.index,columns=['grade','sub_grade'])

grade_split
```

使用pandas进行数据清洗

完成数据分列操作后，使用merge函数将数据匹配回原始数据表，这个操作类似Excel中的Vlookup函数的功能。通过匹配原始数据表中包括了分列后的等级信息。以下是具体的代码和匹配后的结果。

```
loandata=pd.merge(loandata,grade_split,right_index=True, left_index=True)

loandata
```

使用pandas进行数据清洗

merge()介绍

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。

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

使用pandas进行数据清洗

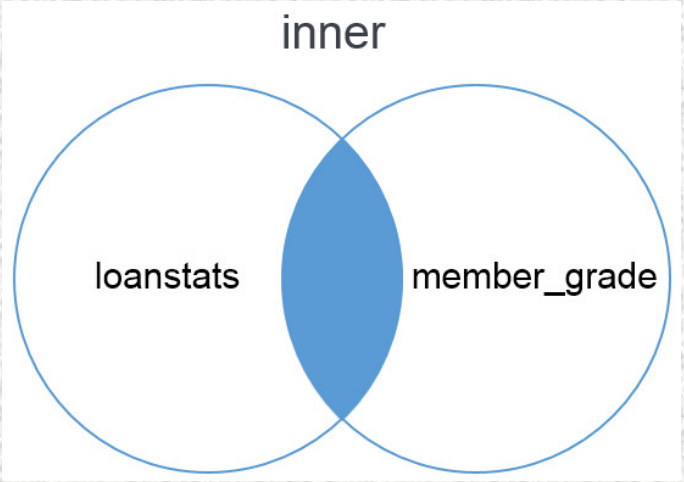
merge中第一个出现的数据表是拼接后的left部分，第二个出现的数据表是拼接后的right部分。第三个是数据匹配模式，默认是inner模式。第四个参数on表示数据匹配所依据的字段名称，如果这个字段名称同时出现在两个数据表中，可省略on参数，merge默认会按照两个数据表中共有的字段名称进行匹配和拼接。若两个数据表中的匹配字段名称不一致，则需分别在left_on和right_on参数中指明两个表匹配字段的名称。如果两个数据表中没有匹配字段，需要使用索引列进行匹配和拼接，可以对left_index和right_index参数设置为True。merge还有一些排序和其他的参数。

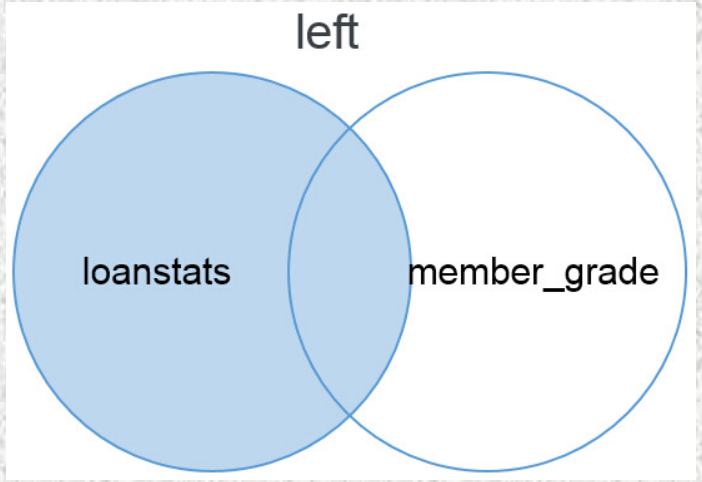
使用pandas进行数据清洗

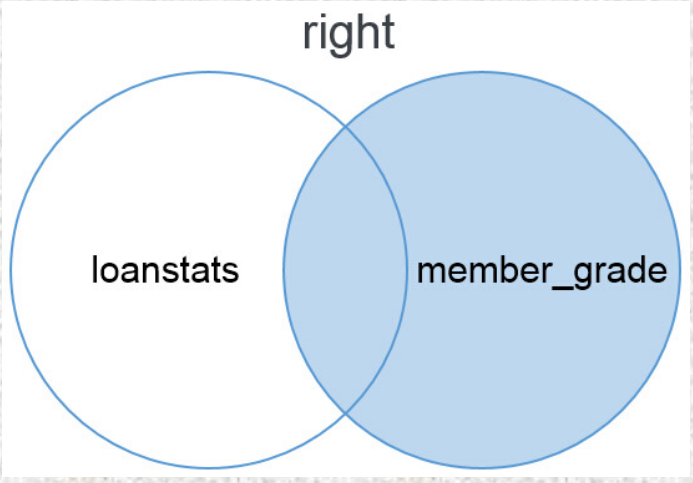
- **Inner模式匹配**
- **left模式匹配**
- **right模式匹配**
- **outer模式匹配**
- **NaN值匹配问题**

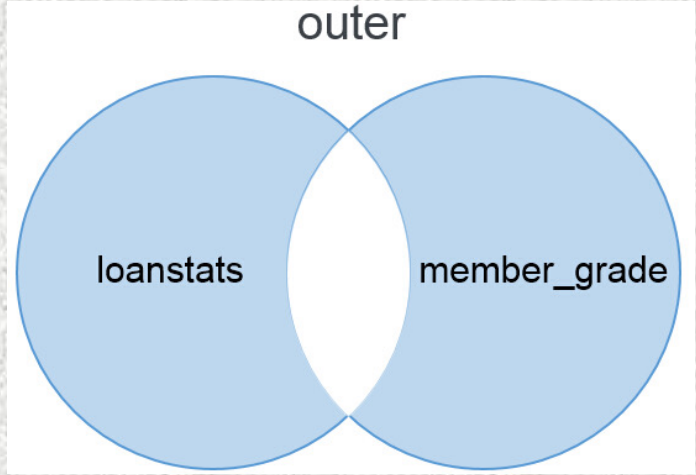
在进行数据匹配和拼接的过程中经常会遇到NaN值。这种情况下merge函数会如何处理呢？

使用pandas进行数据清洗









使用pandas进行数据清洗

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。当使用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。

使用Pandas对数据进行筛选和排序

筛选和排序是Excel中使用频率最多的功能，通过这个功能可以很方便的对数据表中的数据使用指定的条件进行筛选和计算，以获得需要的结果。在Pandas中通过.sort和.loc函数也可以实现这两个功能。.sort函数可以实现对数据表的排序操作，.loc函数可以实现对数据表的筛选操作。下面将介绍如果通过Pandas的这两个函数完成Excel中的筛选和排序操作。

创建数据表后，开始使用Pandas的.sort函数对数据表进行排序操作

使用Pandas对数据进行筛选和排序

.sort函数主要包含6个参数，by 为要进行排序的行或列的名称；axis为排序的轴，0表示index，1表示columns，当对数据列进行排序时，axis必须设置为0；ascending为排序的方式true为升序，False为降序，默认为true。inplace默认为False，表示对数据表进行排序，不创建新实例。Kind可选择排序的方式，如快速排序等。na_position对NaN值的处理方式，可以选择first和last两种方式，默认为last，也就是将NaN值放在排序的结尾。

使用Pandas对数据进行筛选和排序

对单列数据进行排序

升序

```
lc=loandata
lc.sort_values(["loan_amnt"])
#lc.sort_values (["loan_amnt"],ascending=True)

#lc.sort (["loan_amnt"])
#lc.sort (["loan_amnt"],ascending=True)
```

降序

```
lc.sort_values(["loan_amnt"],ascending=False)
#lc.sort ["loan_amnt"],ascending=False)
```

使用Pandas对数据进行筛选和排序

对多列数据进行排序

下面分别对loan_amnt和annual_inc字段进行降序排列

```
lc.sort_values(["loan_amnt","annual_inc"],ascending=
False)
```

两个列名称互换位置，再次执行降序排列操作。

```
lc.sort_values
(["annual_inc","loan_amnt"],ascending=False)
```

使用Pandas对数据进行筛选和排序

获取金额最小前10项

在完成了对数据表排序的操作后，可以对数据表进行简单的筛选，例如获取loan_amnt金额最小的前10名数据。具体的方法是先对lc数据表按loan_amnt升序排列，然后取前10名的数据。NaN值默认在排序结果的结尾显示。

```
lc.sort_values(["loan_amnt"],ascending=True).head(10)

#lc.sort(["loan_amnt"],ascending=True).head(10)
```

使用Pandas对数据进行筛选和排序

获取金额最大前10项

获取金额最大前10项的代码与获取金额最小前10项略有差异，本来只需要复制前面的代码，然后将.head()函数改为tail()函数即可，但由于NaN值在排序的尾部，因此，我们将lc数据表按loan_amnt按降序排列，并取排名前10的数据。当然这并不是唯一的方法，我们还可以通过放弃NaN值的排序或者将NaN值在排序前部显示来解决这个问题。以下是具体的代码和执行结果。

```
lc.sort_values(["loan_amnt"],ascending=False).head(10)
#lc.sort(["loan_amnt"],ascending=False).head(10)
```

使用Pandas对数据进行筛选和排序

介绍完排序功能后再来看下筛选，在筛选功能上Pandas使用的是.loc函数。以下是Pandas对.loc函数的语法和使用方法的说明。

.loc[行标签,列标签]

.loc['a':'b']#选取ab两行数据

.loc[:,'one']#选取one列的数据

.loc的第一个参数是行标签，第二个参数为列标签（可选参数，默认为所有列标签），两个参数既可以是列表也可以是单个字符，如果两个参数都为列表则返回的是DataFrame，否则，则为Series。

使用Pandas对数据进行筛选和排序

单列数据筛选并排序

使用.loc对lc数据表中grade列为B值的数据条目进行了筛选操作。在代码中lc.loc[]是.loc函数的语法，lc[“grade”] == “B”是具体的筛选条件。在最后使用了head()函数只显示前5行筛选结果。从筛选结果来看grade列的值都为B。

```
lc.loc[lc["grade_y"] == "B"].head()
lc.loc[lc["grade_y"] != "B"].head()
```

使用Pandas对数据进行筛选和排序

若只关注数据表中某几列的数据，可以在前面筛选代码的基础上增加要显示的列名称和显示顺序。

```
lc.loc[lc["grade_y"] == "B", ["member_id",
"loan_amnt", "grade_y"]].head()
```

若要对筛选结果进行排序可以联合使用.loc函数和.sort函数。

```
 lc.loc[lc["grade_y"] == "B", ["member_id", "loan_amnt",
"grade_y"]].sort_values (["loan_amnt"])

 lc.loc[lc["grade_y"] != "B", ["member_id", "loan_amnt",
"grade_y"]].sort_values (["loan_amnt"],ascending=False)
```

使用Pandas对数据进行筛选和排序

多列数据筛选并排序

Pandas的.loc参数还可以同时对多列数据进行筛选，并且支持不同筛选条件逻辑组合。

```
lc.loc[(lc["grade_y"] == "B") & (lc["loan_amnt"]>5000),
["member_id",  "loan_amnt", "grade_y","sub_grade",
"total_rec_int"]].head()

lc.loc[(lc["grade_y"] != "B") & (lc["loan_status"] !=
"Charged Off"),["member_id", "loan_amnt", "grade_y",
"sub_grade", "loan_status"]].head()

lc.loc[(lc["grade_y"] == "B") | (lc["loan_amnt"] > 5000),
["member_id", "loan_amnt",
"grade_y","sub_grade", ]].head()
```

使用Pandas对数据进行筛选和排序

多列筛选也可以进行排序，方法与单列筛选后排序基本一样。

```
lc.loc[(lc["grade_y"] == "B") & (lc["loan_amnt"] > 5000), ["member_id",  "loan_amnt", "grade_y","sub_grade"]].sort_values (["loan_amnt"])

lc.loc[(lc["grade_y"] == "B") | (lc["loan_amnt"] > 5000), ["member_id", "loan_amnt", "grade_y","sub_grade"]].sort_values (["loan_amnt"],ascending=False)
```

使用Pandas对数据进行筛选和排序

按筛选条件求和(sumif, sumifs)

在单列筛选的代码后增加求和条件就相当于Excel中的sumif函数的功能。

```
lc.loc[lc["grade_y"] == "B",].loan_amnt.sum()

lc.loc[lc["grade_y"] != "B",].loan_amnt.sum()

lc.loc[(lc["grade_y"] == "B") & (lc["loan_amnt"] >
5000)].loan_amnt.sum()
```

使用Pandas对数据进行筛选和排序

按筛选条件计数(countif, countifs)

将前面的.sum()函数换为.count()函数就变成了Excel中的countif函数的功能。

```
lc.loc[lc["grade_y"] == "B"].loan_amnt.count()

lc.loc[lc["grade_y"] != "B"].loan_amnt.count()

lc.loc[(lc["grade_y"] == "B") & (lc["loan_amnt"] >
5000)].loan_amnt.count()
```

使用Pandas对数据进行筛选和排序

- 按筛选条件计算均值(averageif, averageifs)

  在Pandas中.mean()是用来计算均值的函数，将.sum()和.count()替换为.mean()。就是pandas版的averageif和averageifs。

```
lc.loc[lc["grade_y"] == "B"].loan_amnt.mean()

lc.loc[lc["grade_y"] != "B"].loan_amnt.mean()

lc.loc[(lc["grade_y"] == "B") | (lc["loan_amnt"] >
5000)].loan_amnt.mean()
```

使用Pandas对数据进行筛选和排序

按筛选条件获取最大值和最小值

最后两个是Excel中没有的函数功能，就是对筛选后的数据表计算最大值和最小值。方法很简单，将之前的sum()和count()换成max()和min()函数即可。下面是具体的代码和结果。

```
lc.loc[lc["grade_y"] == "B"].loan_amnt.max()
#lc.loan_amnt.max()

lc.loc[lc["grade_y"] != "B"].loan_amnt.min()
```

使用Pandas创建数据透视表

pandas数据透视表函数

数据透视表是Excel中最常用的数据汇总工具，它可以根据一个或多个制定的维度对数据进行聚合。在python中同样可以通过pandas.pivot_table函数来实现这些功能。

pandas.pivot_table函数中包含四个主要的变量，以及一些可选择使用的参数。四个主要的变量分别是数据源data，行索引index，列columns，和数值values。可选择使用的参数包括数值的汇总方式，NaN值的处理方式，以及是否显示汇总行数据等。

使用Pandas创建数据透视表

创建简单的数据透视表

按贷款期限维度对贷款总额进行聚合，将贷款期限字段(emp_length)放在行索引Index中，贷款总额字段 (loan_amnt)放在值values中，生成按不同贷款期限维度聚合的贷款总额数据。在**默认情况下**pandas.pivot_table对指标的汇总方式是**计算平均值**。 因此下面的表中显示的是不同贷款期限的贷款平均值数据。这个简单的数据透视表只有一个维度和一个指标

```
pd.pivot_table(lc,index=["emp_length"],values=["loan_amnt"])
```

使用Pandas创建数据透视表

增加一个行维度(index)

在贷款期限的维度上增加贷款用户等级维度，创建一个双维度的数据透视表，在pandas.pivot_table的行索引index中增加贷款用户等级字段(grade_y)。这样在行索引维度index中共包含了两个维 度，主维度工作年限(emp_length)和次级维度贷款用户等级(grade_y)。指标是按不同贷款期限下贷款用户等级分布进行汇总贷款金额平均值。经过次级维度的细分变的更加精细。 。

```
pd.pivot_table(lc,index=["emp_length","grade_y"],values=["loan_amnt"])
```

使用Pandas创建数据透视表

通过调整pandas.pivot_table函数中不同维度的位置可以更改数据透视表中维度的层级，以及数据的显示方式。这里我们将前面代码行索引中两个字段位置互换，此时贷款用户等级(grade_y)成了主维度，工作年限(emp_length)变成了次级维度。

```
pd.pivot_table(lc,index=["grade_y","emp_length"],values=["loan_amnt"])
```

使用Pandas创建数据透视表

增加一个值变量(value)

除了增加次级维度以外，还可以增加需要汇总的数据值。在前面数据透视表的基础上增加总利息字段作为第二个汇总值。方法与前面增加次级维度很相似，将需要增加的字段放在值values中即可。下面是具体的代码和生成的数据透视表，其中total_rec_int是新增的值values变量。这里需要再次说明的是，默认情况下pandas.pivot_table按平均值对数据进行汇总。

```
pd.pivot_table(lc,index=["grade_y","emp_length"],values=["loan_amnt","total_rec_int"])
```

使用Pandas创建数据透视表

更改数值汇总方式

若要更改pandas.pivot_table对值values的汇总方式需要在代码中进行设置，下面将贷款总额和总利息字段的汇总方式改为求和。方法是在代码中加入aggfunc=np.sum。新生成的数据透视表中值字段的计算方式就由之前的平均值改为了求和值。

```
pd.pivot_table(lc,index=["grade_y","emp_length"],values=["loan_amnt","total_rec_int"],aggfunc=np.sum)
```

使用Pandas创建数据透视表

增加数值汇总方式

除了可以对值变量values计算平均值和求和以外，还可以进行计数。下面分别对贷款总额和总利息字段进行求和，平均值和计数的计算。具体方法是代码中增加以下内容aggfunc=[np.sum,np.mean,len])，aggfunc是汇总方式，np.sum表示求和，np.mean表示计算平均值，len表示计数。

```
pd.pivot_table(lc,index=["grade_y","emp_length"],values=["loan_amnt","total_rec_int"],aggfunc=[np.sum,np.mean,len])
```

使用Pandas创建数据透视表

如果数据表中包含有NaN值，并且在之前的清洗中没有进行处理，也可以在生成数据透视表的过程中进行处理或替换。在pandas.pivot_table函数中有两种处理NaN值的方式，第一种是将NaN值替换为0。第二种为放弃NaN值，也就是说包含有NaN值的数据条目不参加计算。这里使用第一种方法，将NaN值替换为0。具体方法是在代码中添加以下部分fill_value=0。

```
pd.pivot_table(lc,index=["grade_y","emp_length"],values=["loan_amnt","total_rec_int"],aggfunc=[np.sum,np.mean,len],fill_value=0)
```

使用Pandas创建数据透视表

增加一个列维度(columns)

pandas.pivot_table函数也支持列维度。在Excel中需要将对应的字段拖到列区域中，在Pandas中的方法是增加列columns，并将对应的字段名称放在列columns变量的值中。

```
pd.pivot_table(lc,index=["grade_y"],values=["loan_amnt"],columns=["emp_length"],aggfunc=[np.sum],fill_value=0)
```

使用Pandas创建数据透视表

- 增加数据汇总值

pandas.pivot_table函数中的margins参数用于增加数据透视表的汇总值。默认情况下margins的状态为False。需要增加透视表的汇总值时将margins值改为True即可。此时数据透视表将显示不同维度下数据的汇总值。汇总值的计算方式以aggfunc的一致。换句话说，如果aggfunc中设置的是求和，那么汇总值也是求和值。

```
pd.pivot_table(lc,index=["grade_y"],values=["loan_amnt"],columns=["emp_length","open_acc"],aggfunc=[np.sum],fill_value=0,margins=True)
```

使用Pandas创建数据透视表

最后，总结下pandas.pivot_table函数与数据透视表的对应关系。将每部分以不同颜色进行区分，index对应了数据透视表中行的索引部分(浅蓝色)，values对应了数值的部分(绿色)，columns对应了列的部分，(橙色表示主维度，黄色表示次级维度)，aggfunc对应了数值的计算方式(紫色)，并显示数据透视表的最顶部进行说明(sum)。Margins对应了数据透视表中值汇总的部分 (深蓝色)。

使用Pandas创建数据透视表

```
In [13]: pd.pivot_table(lc, index=["grade"], values=["loan_amnt"], columns=["home_ownership", "term"], aggfunc=[np.sum], fill_value=0, margins=True)
```

Out[13]:

	sum									
	loan_amnt									
home_ownership	MORTGAGE		NONE	OTHER		OWN		RENT		All
term	36 months	60 months	36 months	36 months	60 months	36 months	60 months	36 months	60 months	
grade										
A	44591100	3143100	35200	168975	0	6669850	306450	31643800	997625	87556100
B	42889550	25719525	6800	395825	0	7203875	3143700	45530150	11982350	136871775
C	23880800	22141075	0	223100	0	4016100	2165975	30211400	12385375	95023825
D	16399375	16471425	0	248725	16000	2904725	1773175	20671400	12796850	71281675
E	6443050	19665225	0	148150	0	827700	2405375	7261125	13153350	49903975
F	2430200	9933300	15000	64500	0	286425	841225	2613075	6129250	22312975
G	1566575	3139325	0	59650	0	210225	370675	996900	2407675	8751025
All	138200650	100212975	57000	1308925	16000	22118900	11006575	138927850	59852475	471701350

使用Pandas进行数据提取

数据提取是分析师日常工作中经常遇到的需求。如某个用户的贷款金额，某个月或季度的利息总收入，某个特定时间段的贷款金额和笔数，大于5000元的贷款数量等等。如何通过python按特定的维度或条件对数据进行提取，完成数据提取需求。

```
loandata=pd.DataFrame(pd.read_csv('loandata.csv'))

#设置索引字段
Loandata = loandata.set_index('member_id')
```

使用Pandas进行数据提取

按行与列提取信息

```
#第一步是按行提取数据
Loandata.loc[1303503]  #Loandata.ix[1303503]
#第二步是按列提取数据
Loandata.loc[:,'emp_length'] #Loandata.ix[:,'emp_length']
#第三步是按行和列提取信息
Loandata. loc[1303503,'emp_length']

Loandata. loc[[1303503,1304871],'loan_amnt']
Loandata. loc[[1303503,1304871],'loan_amnt'].sum()

Loandata. loc[1303503,['loan_amnt','annual_inc']]
Loandata. loc[1303503,['loan_amnt','annual_inc']].sum()
```

使用Pandas进行数据提取

提取特定日期的信息

数据提取中还可按日期维度对数据进行汇总和提取，如按月，季度的汇总数据提取和按特定时间段的数据提取等等。

设置索引字段

首先将索引字段改为数据表中的日期字段，这里将issue_d设置为数据表的索引字段。按日期进行查询和数据提取。

```
loandata = loandata.set_index('issue_d')
from datetime import datetime
loandata.index = pd.to_datetime(loandata.index)
# 将字符串索引转换成时间索引
```

使用Pandas进行数据提取

按日期提取信息:

```
#查询了所有2016年的数据。
loandata['2016']
loandata['2016-06']
loandata['2016-06-14']
loandata['2016-01':'2016-05'].sort_index()
#loandata['2016-01':'2016-05'].sort()
```

使用Pandas进行数据提取

按日期汇总信息

Pandas中的resample函数可以完成日期的聚合工作，包括按小时维度，日期维度，月维度，季度及年的维度等等。W表示聚合方式是按周，how表示数据的计算方式，默认是计算平均值，这里设置为sum，进行求和计算。

```
loandata.resample('W',how=sum).head(10)
loandata.resample('M',how=sum)   #按月聚合
loandata.resample('Q',how=sum)  #按季度对数据进行聚合
loandata.resample('A',how=sum)   #按年对数据进行聚合
loandata[['loan_amnt','total_rec_int']].resample('M',how=[
len,sum])
loandata[loandata['loan_amnt']>5000].resample('M',how=
sum).fillna(0)
```

使用python进行简单的数据分析

- 开始前的准备工作

使用Python进行数据分析之前，需要预先导入相对应的功能库。数据分析最常用的库包括用于数值计算的numpy，基于numpy构建的用于科学计算的Pandas库，用于数据可视化的matplotlib和提供各种操作系统功能接口的OS库。

导入功能库后，将数据所在位置的路径设置为工作目录。读取LoanStats3a.csv文件，并设置标题行header=1。然后将读取的csv文件转成DataFrame并将这个数据表取名lc。

```
lc=pd.DataFrame(pd.read_csv('LoanStats3a.csv',header=1))
```

使用python进行简单的数据分析

- 数据内容预览

数据读取工作完成后，可以开始对数据进行简单的预览。预览内容主要包括了解数据表的大小，字段的名称，数据格式等等。

```
lc.shape
lc.columns
lc.head()
lc.tail(3)
lc.dtypes
```

在数据内容概览部分大致了解了数据表中的信息，同时也发现表中包含很多NaN值，以及日期格式转换的问题。这些问题将在后面的数据清洗部分进行解决。

使用python进行简单的数据分析

- 数据清洗

这部分主要解决前面发现的NaN值和日期格式转换问题。通过查看数据表中的空值情况可以发现，前面的字段中NaN值较少，NaN值主要集中在后面的一些字段中。

```
lc.isnull()
lc.isnull().sum()
```

从查看的结果中可以看出通过格式转换issue_d字段的格式已经从object变成了datetime64。

```
lc['issue_d']=pd.to_datetime(lc['issue_d'])
```

使用python进行简单的数据分析

- 关键指标概览

清洗后的数据表可以开始进行分析工作，首先是对一些关键汇总指标的统计，对于Lending Club这些关键指标包括总贷款次数，总贷款金额，总利息收入等等。

```
lc['member_id'].count()
len(lc['member_id'].unique())
```

对数据表中member_id字段进行唯一值计数，获得总贷款人数。这里贷款次数与人数基本一致。猜测这个值应为贷款次数。

使用python进行简单的数据分析

对数据表中loan_amnt字段进行求和，获得总贷款金额。

```
lc['loan_amnt'].sum()
471701350.0
```

将贷款金额分布绘制成箱线图可以看出，贷款均值为10000元。大部分贷款的金额都集中在5000到15000之间。

```
lc['loan_amnt'].plot.box()
lc['loan_amnt'].plot.hist()

#plt.boxplot(lc['loan_amnt'])
#plt.show()
```

使用python进行简单的数据分析

换成直方图再看下，5000美金一个区间，与之前的结果类似，15000以下的贷款居多，其中最多的是5000到10000的区间。大额的贷款也有但笔数较少，最高的一笔35000美金。

```
bins = [0, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000]
plt.hist(lc['loan_amnt'],bins,range=[0,40000],normed=1, histtype='bar',rwidth=1)
plt.show()
```

使用python进行简单的数据分析

从2007到2011年贷款笔数增长趋势看，虽然中间贷款笔数整体呈快速增长趋势，并且增长速度非常快，从开始每月几十笔到近2500笔。这应该与美国人的理财观念有关系。在后面的贷款用途分析中也可以发现，他们的贷款目的和用途非常广泛，有些甚至有点奇怪。贷款目的可能是为了一次旅行，结婚，甚至还有一个贷款目的写的是为了自由。

```
#lc.sort(columns='issue_d',ascending=False)
issue_date=lc.groupby('issue_d').count()
plt.plot(issue_date['member_id'],linewidth=3)
plt.show()
```

使用python进行简单的数据分析

继续对数据表中total_rec_in字段进行求和，获得总利息收入金额。除了利息收入以外，Lending Club还有一部分收入是罚息和类似手续费的收入。这里并没有包括这些收入，只是简单sum了贷款利息收入。

```
lc['total_rec_int'].sum()
95263452.530000001
```

最后再对表中各个字段做个描述统计，看下数据的集中度与离散情况，这里主要关注最大值，最小值和标准差等。

```
lc.describe().T
```

使用python进行简单的数据分析

- 用户属性分析

Lending Club的数据表中包含很多与用户相关的字段，例如用户收入，贷款用途，用户等级，职位，所在地区等等。可以通过这些字段从多个维度做个贷款用户画像。

首先是收入情况，Lending Club的贷款用户收入差距还是挺高的，从1-2万美金/年到30-40万美金/年的都有。以5万美金/年对贷款用户的收入分布进行了统计。

```
bins = [0, 50000, 100000, 150000, 200000, 250000,
300000, 350000, 400000, 450000]
plt.hist(lc['annual_inc'],bins,range=[0,450000],
histtype='bar', rwidth=1)
plt.show()
```

使用python进行简单的数据分析

前面的贷款金额分布与用户收入非常相似，小额贷款高于大额贷款。那么贷款金额的大小与贷款用户的收入之间是否有联系呢？对用户收入和贷款金额进行相关分析。从下面的结果中可以看出两者的相关系数仅为0.27，推翻了之前的假设，也就是说并不是收入越高的人贷款金额也越高。

```
c=lc[['annual_inc','loan_amnt']]
c.corr()
```

使用python进行简单的数据分析

了解了用户的收入分布情况后，再来逐一看下其他维度的用户属性。贷款目的分布中，debt_consolidation(债务合并)的数量最高，也就是借新还旧。其次是credit_card(还信用卡)。第三名是其他，第四名的是home_improvement(家装)。后面还有婚礼，医疗，教育和度假等等。贷款目的非常多元化。下面的这些贷款目的是Lending Club整理并汇总后的信息。

```
lc['purpose'].value_counts()
lc['title'].value_counts()
#再细一级的贷款目的内容就更加庞杂和有意思了。真的能感觉到美国人做什么事情都通过贷款来实现。
```

使用python进行简单的数据分析

从信用等级分布来看，Lending Club中大部分用户为B和A级，G级用户数量较少。大部分用户都拥有较高的信用等级。

```
lc['grade'].value_counts()
```

数据表中的emp_title字段记录了用户的职位名称，简单的统计后发现贷款用户的职业分布非常广泛。其中排第一的是US Army。其次为Bank of America。后面还包括IBM，AT&T，UPS的用户。

```
lc['emp_title'].value_counts()
```

使用python进行简单的数据分析

在工作年限上贷款用户的分布非常极端，数量排名第一的是工作10年以上，第二名是工作小于1年。第三名是工作2年。可见刚毕业的年轻人和30岁以上的人是贷款的主要人群。年轻人刚开始工作，欲望大于收入，需要贷款来实现需求。岁数大一些的人竞争力逐渐下降，负担变重，也需要靠贷款来周转。

```
lc['emp_length'].value_counts()
```

使用python进行简单的数据分析

从贷款人的房屋情况来看，大部分用户在租房，其次排名第二的用户房屋进行了抵押贷款，自己拥有房屋的仅为3251人，排名第三。

```
lc['home_ownership'].value_counts()
```

从地域的分布来看排名第一个的是加利福尼亚州，第二是纽约州，第三是佛罗里达州 。Lending Club为了降低坏账率有一套投资人与贷款人的职能匹配系统，里面包含地理位置的匹配。

```
lc['addr_state'].value_counts()
```

使用python进行简单的数据分析

- 产品数据分析

Lending Club产品主要关注Lending Club的贷款利率和贷款期限情况。

从贷款利率的角度来看，Lending Club的贷款利率从5%到25%跨度较大，并且每个人的利率都不一样，即使同样借款金额和期限的两个贷款人，也可能有不同的贷款利率，这些利息是Lending Club对贷款人评估后计算出来的。粗略来看贷款利息依据用户的信用等级变化而变化，A级用户贷款利率低，D和E级或者更低的等级用户贷款利率普遍较高。

```
lc['int_rate'].value_counts()
```

使用python进行简单的数据分析

从贷款期限角度来看，Lending Club的贷款都为长期贷款，分为36个月和60个月两类。很难想象美国人借500美金还要分36个月还清贷款。

```
lc['term'].value_counts()
```

从饼图上可以更清楚的看到两类贷款期限的占比情况，约75%的贷款为36个月的，25%为60个月的期限。

```
labels = '36 months', '60 months '
colors = ['lightskyblue', 'lightcoral']
plt.pie(lc['term'].value_counts(),labels=labels,colors=co
lors,shadow=True,startangle=90)
plt.show()
```

使用python进行简单的数据分析

两个贷款期限对应的金额和笔数情况来看，60个月的贷款金额占比要高于36个月的贷款笔数的占比，可见60个月中应该有一部分为大金额贷款。

```
lc.groupby('term')['loan_amnt'].agg(['count','sum'])
```

使用python进行简单的数据分析

- 运营数据分析

从运营的角度来分析，Lending Club的贷款都放给了那类用户，又从哪些用户身上获得了最多的利息收入呢？

以下是分等级贷款金额和利息收入金额笔数对比情况，具体代码如下：这里使用了分类汇总功能，将数据表中的loan_amnt和total_rec_int按grade维度进行汇总，并分别进行计数和求和。

```
lc.groupby('grade')['loan_amnt','total_rec_int'].agg(['count','sum'])
#随着等级的下降获得的贷款笔数和金额逐渐减少,利息增高
```

使用python进行简单的数据分析

换个角度从贷款期限维度来分析，60个月的贷款笔数和金额明显小于36个月。但从获得的利息收入情况来看正好相反，Lending Club从60个月的贷款中获得的利息收入要明显高于36个月。贷款期限越长，对于投资人的风险越大，但相应的收益也越高。

以下是分期限贷款金额和利息收入金额及笔数对比情况，具体代码如下：这里同样使用了分类汇总功能。

```
lc.groupby('term')['loan_amnt','total_rec_int'].agg(['count','sum'])
```

使用python进行简单的数据分析

- 风控数据分析

  从风险控制的角度分析，随着Lending Club贷款笔数的增长，Charged Off的数量也随之增加，以下是Charged Off的变化趋势图

。

```
x=lc.loc[(lc["loan_status"]=="Charged Off")]
y=x.groupby('issue_d').count()
plt.plot(y["loan_status"])
plt.show()
```

使用python进行简单的数据

从贷款状态的角度分析，共有约6000笔贷款出现Charged Off的情况。另外还有9单在宽恕期限内还款，3笔贷款延迟半个月到1一个月还款，10比贷款延迟1-3个月还款。

以下是分贷款状态金额和笔数对比情况，具体代码如下：这里使用了类似Excel数据透视表的功能，行为贷款状态，数值为贷款的金额和笔数汇总值。

```
pd.pivot_table(lc,index=["loan_status"],values=["loan_amnt"],aggfunc=[np.sum,len])
```

使用python进行简单的数据

在上面数据表的基础上增加用户信用等级维度再来看下，信用等级较高的用户Charged Off情况较少，信用等级低的用户由于获得贷款的笔数相对较低，所以看起来Charged Off的数量也较低，但占比却较高。

以下是分等级贷款状况金额和笔数对比的情况，具体代码如下：这里依然使用了类似Excel数据透视表的功能，在之前的基础上增加了列字段，并设定贷款用户等级为列字段的值

```
pd.pivot_table(lc,index=["loan_status",],values=['loan_amnt'],columns=["grade"],aggfunc=[np.sum,len],fill_value=0)
```

使用python进行简单的数据

再从贷款用途的维度来分析，Charged Off最高的是debt_consolidation(债务合并)，也就是借新债还旧债的情况，这种属于风险较高的一种贷款目的。这是一种比较粗糙的推测结果，更准确的情况还需要看每类贷款目的自身Charged Off的比率情况才能准确判断。

```
pd.pivot_table(lc,index=["purpose"],values=["loan_amnt"],columns=["loan_status"],aggfunc=[len,np.sum],fill_value=0)
```