Christopher Rico MSDS 420 Exercise 5 3/8/20

Deliverables:

- Submit a single zip-compressed file that has the name: YourLastName_Exercise_1 that has the following files:

    1. Your **PDF document** that has your Source code and output
    2. Your **ipynb script** that has your Source code and output

Objectives:

In this exercise, you will:

- Construct hierarchical indexes
- Select and group data to create pivot-tables

Submission Formats :

Create a folder or directory with all supplementary files with your last name at the beginning of the folder name, compress that folder with zip compression, and post the zip-archived folder under the assignment link in Canvas. The following files should be included in an archive folder/directory that is uploaded as a single zip-compressed file. (Use zip, not StuffIt or any 7z or any other compression method.)

1. Complete IPYNB script that has the source code in Python used to access and analyze the data. The code should be submitted as an IPYNB script that can be be loaded and run in Jupyter Notebook for Python
2. Output from the program, such as console listing/logs, text files, and graphics output for visualizations. If you use the Data Science Computing Cluster or School of Professional Studies database servers or systems, include Linux logs of your sessions as plain text files. Linux logs may be generated by using the script process at the beginning of your session, as demonstrated in tutorial handouts for the DSCC servers.
3. List file names and descriptions of files in the zip-compressed folder/directory.

Formatting Python Code When programming in Python, refer to Kenneth Reitz' PEP 8: The Style Guide for Python Code: [http://pep8.org/ (http://pep8.org/)](http://pep8.org/) (Links to an external site.)Links to an external site. There is the Google style guide for Python at [https://google.github.io/styleguide/pyguide.html (https://google.github.io/styleguide/pyguide.html)](https://google.github.io/styleguide/pyguide.html) (Links to an external site.)Links to an external site. Comment often and in detail.

Specifications and Requirements

We're going to use the XYZ data again to construct hierarchical indexes and select, modify, group, and reshape data in a wide variety of ways. The data we want here, which we'll call xyzcustnew, are as follows:

In [1]:
```
import pandas as pd  # panda's nickname is pd
import numpy as np  # numpy as np
from pandas import DataFrame, Series, Categorical
from sqlalchemy import create_engine
```

In [3]:
```
engine=create_engine('sqlite:///xyz.db')             # the db is in my cu
rrent working directory
```

In [4]:
```
xyzcustnew=pd.read_sql_table('xyzcust',engine)
```

In [5]:
```
# Refer to exercise #7 how we calculated this value for xyz db
heavyCut=423  #heavyCut is a constant
```

In [6]:
```
heavyCat=Categorical(np.where(xyzcustnew.YTD_SALES_2009>heavyCut,1,0))
heavyCat.describe()
```

Out[6]:

	counts	freqs
categories		
0	25795	0.854733
1	4384	0.145267

In [7]:
```
heavyCat.rename_categories(['regular','heavy'],inplace=True)
```

In [8]:
```
heavyCat.describe()
```

Out[8]:

	counts	freqs
categories		
regular	25795	0.854733
heavy	4384	0.145267

In [9]:
```
heavyCat[:10]
```

Out[9]:
```
[regular, heavy, regular, regular, regular, regular, heavy, regular, re
gular, regular]
Categories (2, object): [regular, heavy]
```

In [10]:
```
xyzcustnew['heavyCat']=heavyCat
```

In [11]:
```
buyerType=pd.get_dummies(heavyCat)
```

In [12]:
```
buyerType[:3]
```

Out[12]:

	regular	heavy
0	1	0
1	0	1
2	1	0

In [13]:
```
xyzcustnew['typeReg']=buyerType['regular']
xyzcustnew['typeHeavy']=buyerType['heavy']
```

In [14]:
```
xyzcustnew.columns
```

Out[14]:
```
Index(['index', 'ACCTNO', 'ZIP', 'ZIP4', 'LTD_SALES', 'LTD_TRANSACTION
S',
       'YTD_SALES_2009', 'YTD_TRANSACTIONS_2009', 'CHANNEL_ACQUISITIO
N',
       'BUYER_STATUS', 'ZIP9_SUPERCODE', 'heavyCat', 'typeReg', 'typeHe
avy'],
      dtype='object')
```

In [15]:
```
# for this exercises we need to create trCountsChrono object similar to
 what we did in exercises #8

xyztrans=pd.read_sql('xyztrans', engine)

trandate=xyztrans.TRANDATE      # should be a Series

daystr=trandate.str[0:2]                # two digit date numbers slice

mostr=trandate.str[2:5]         # the three letter month abbreviations

yearstr=trandate.str[5:]                # four digit years
```

In [16]:
```
#create a dictionary for the months
monums={'JAN':'1', 'FEB':'2', 'MAR':'3', 'APR':'4', 'MAY':'5', 'JUN':'6'
, 'JUL':'7', 'AUG':'8', 'SEP':'9', 'OCT':'10', 'NOV':'11','DEC':'12'}
#month
monos=mostr.map(monums) # do a dict lookup for each value of mostr

transtr=yearstr+'-'+monos+'-'+daystr
```

transtr should be a Series. Now let's convert the string values in transtr into datetime values:

In [17]:
```
trDateTime=pd.to_datetime(transtr)
```

In [18]:
```
trCounts=trDateTime.value_counts()
```

The order of the counts in trDateTime is not chronological, so let's reorder them so that they go from earliest to most recent date.

In [19]:
```
newIndex=pd.date_range(trCounts.index.min(),trCounts.index.max())

trCountsChrono=trCounts.reindex(index=newIndex)
```

In [20]:
```
print(trCountsChrono.head())
```

```
2009-01-01    176
2009-01-02    305
2009-01-03    365
2009-01-04    231
2009-01-05    144
Freq: D, Name: TRANDATE, dtype: int64
```

One of the very handy things you can do with pandas DataFrames and Series is that you can create what are called hierarchical indexes. These are multi-level indexes (the are in fact called MultiIndexes). They make it easier to select, modify, group, and reshape data in a wide variety of ways. They make it possible to work with high dimensional data in data structures that are in just one or two dimensions. Let's change trCountsChrono a bit to produce a first simple example of a Series with a hierarchical index. First, let's put the Series into a DataFrame and then rename the columns:

In [21]:
```
trDF=DataFrame()
```

In [22]:
```
trDF['date'] = trCountsChrono.index
trDF['transactions'] = trCountsChrono.values
```

In [23]:
```
trDF.columns
```

Out[23]:
```
Index(['date', 'transactions'], dtype='object')
```

In [24]:
```
trDF.head()
```

Out[24]:

	date	transactions
0	2009-01-01	176
1	2009-01-02	305
2	2009-01-03	365
3	2009-01-04	231
4	2009-01-05	144

In [27]:
```
trDF.dtypes
```

Out[27]:
```
date            datetime64[ns]
transactions             int64
dtype: object
```

Note that the data types of the columns have not changed. Try trDF.dtypes. Now, let's create a new column that indicates whether the number of daily transactions are heavy or light depending on whether the are equal to or greater than the median number of transactions, or less than the median number. There are more succinct ways to do this, but this is transparent, if not efficient:

In [28]:
```
trMed=trDF.transactions.median()                    # here's the median
```

In [29]:
```
heavyLight=lambda x  : x >= trMed and 'heavy' or 'light'  # an example a
non function
```

In [30]:
```
trDF['vol']=trDF.transactions.map(heavyLight)   # 'vol' is the heavy/lig
ht column
```

Note that this lambda would stumble if trMed wasn't known at the time lambda was called by the map method. Anyway, next we're going to create, monum, a variable indicating the month of the calendar year that each day falls into:

In [32]:
```
trDF['monum']=trDF.date.dt.month         # .dt is the datetime accessor
```

Next, we're going to collapse the daily transaction counts into monthly counts. When we do this we'll keep the heavy versus light daily volume distinction. First we're going to drop the 'date' column because we no longer need it. To be safe we'll copy the result to a new DataFrame just in case something goes wrong:

In [33]:
```
trDFnd=trDF.drop('date',axis=1) # axis=1 means here a column is selected
to drop
```

Now using this DataFrame's groupby() method, sum up the transactions within month by heavy volume days and light volume days:

In [34]:
```
trDFgrouped=trDFnd.groupby(['monum','vol']).sum()
```

In [36]:
```
trDFgrouped.head(10)
```

Out[36]:

		transactions
monum	**vol**	
1	heavy	5255
	light	572
2	heavy	761
	light	1625
3	heavy	1130
	light	1664
4	heavy	2327
	light	1727
5	heavy	2172
	light	2076

Now if you look at this DataFrame you'll see that it has two levels of indexing, monum, and within the levels of monum, vol. If you enter trDFgrouped.index you'll get back a MultiIndex object. Also, try trDFgrouped.index.levels to see what you get. pandas has pretty seamlessly created this index for you, but you can construct MultiIndexes manually by combining equal length arrays (using MultiIndex.from_arrays) of index levels, or by using tuples (with MultiIndex.from_tuples). In both cases all combinations of the levels need to be included. Or, you can use MultiIndex.from_product to get a cross set of the values of iterables. Note that if you look at trDFgrouped you may see here and there that for a particular month, the number of heavy day transactions is less than the number of light day transactions. How do you think that could happen? You can use MultiIndexes to select and subset DataFrames and Series in many of the same ways you can use simple indexes. For example, to get the heavy days transaction count data for November, you can do:

In [37]:
```
trDFgrouped.loc[11,'heavy']
```

Out[37]:
```
transactions    8402
Name: (11, heavy), dtype: int64
```

The first six months of data:

In [31]:
```
trDFgrouped.loc[list(range(1,7))]
```

Out[31]:

monum	vol	transactions
1	heavy	5255
	light	572
2	heavy	761
	light	1625
3	heavy	1130
	light	1664
4	heavy	2327
	light	1727
5	heavy	2172
	light	2076
6	heavy	2878
	light	1495

or the first 6 rows of data:

In [32]:
```
trDFgrouped.iloc[0:6]            # .iloc here, but .loc above.
```

Out[32]:

monum	vol	transactions
1	heavy	5255
	light	572
2	heavy	761
	light	1625
3	heavy	1130
	light	1664

The data starting from the March heavy day counts to the July light counts:

```
In [33]: trDFgrouped[(3,'light'):(7,'heavy')]
```

Out[33]:

monum	vol	transactions
3	light	1664
4	heavy	2327
	light	1727
5	heavy	2172
	light	2076
6	heavy	2878
	light	1495
7	heavy	4440

The above uses a range defined by a slice of tuples. So does:

```
In [34]: trDFgrouped[(3,'light'):6]
```

Out[34]:

monum	vol	transactions
3	light	1664
4	heavy	2327
	light	1727
5	heavy	2172
	light	2076
6	heavy	2878
	light	1495

Try selecting some data and slicing a few times yourself. It takes a little practice to get the hang of getting what you want. There are many other ways to slice using MultiIndexes. One other you might find interesting is the cross-section method .xs. Here's an example that picks out data for the light days:

```
In [35]: trDFgrouped.xs('light',level='vol')
```

Out[35]:

monum	transactions
1	572
2	1625
3	1664
4	1727
5	2076
6	1495
7	564
8	1938
9	1942
10	2241
11	49
12	257

As you probably know, DataFrames have a transpose method, .T:

```
In [38]: trDFgrouped.xs('light',level='vol').T           # the transpose of the a
         bove
```

Out[38]:

monum	1	2	3	4	5	6	7	8	9	10	11	12
transactions	572	1625	1664	1727	2076	1495	564	1938	1942	2241	49	257

Did you get a table of transactions with cells labeled by monum across the top? You can also pivot DataFrames in various ways. Let's make some data to create a DataFrame we can pivot. We'll put the monum and vol indexes from trDFgrouped into our new DataFrame as columns, and then we'll add transactions as a third column.

```
In [39]: mo=trDFgrouped.index.get_level_values(0)          # the month numbers
```

```
In [40]: volType=trDFgrouped.index.get_level_values(1)    # vol
```

```
In [41]: trDFpiv=DataFrame({'month':mo,'vol': volType, 'transactions':trDFgrouped
         .transactions})          # data as a dict
```

In [43]:
```
trDFpiv.head(10)
```

Out[43]:

		month	vol	transactions
monum	vol			
1	heavy	1	heavy	5255
	light	1	light	572
2	heavy	2	heavy	761
	light	2	light	1625
3	heavy	3	heavy	1130
	light	3	light	1664
4	heavy	4	heavy	2327
	light	4	light	1727
5	heavy	5	heavy	2172
	light	5	light	2076

Now, let's pivot trDFpiv. Let's make a new DataFrame with month as the index, vol the columns, and the transaction counts as the values:

In [42]:
```
trDFpived=trDFpiv.pivot(index='month',columns='vol',values='transaction
s')
```

In [45]:
```
trDFpived.head(10)
```

Out[45]:

vol	heavy	light
month		
1	5255	572
2	761	1625
3	1130	1664
4	2327	1727
5	2172	2076
6	2878	1495
7	4440	564
8	1682	1938
9	1921	1942
10	2109	2241

How does trDFpived look to you? If trDFpiv had more than one column for values not used as a column or an index, hierarchical columns would be created to reflect them. For example, let's add an additional column to trDFpiv:

In [46]:
```
trDFpiv['randy']=np.random.randn(len(trDFpiv))
```

Now pivot trDFpiv like:

In [47]:
```
trDFpived2=trDFpiv.pivot(index='month',columns='vol')
```

In [48]:
```
trDFpived2.head(10)
```

Out[48]:

	transactions		randy	
vol	**heavy**	**light**	**heavy**	**light**
month				
1	5255	572	-3.154672	0.118033
2	761	1625	-0.672371	-1.111379
3	1130	1664	-0.834920	-0.777409
4	2327	1727	0.632008	0.786350
5	2172	2076	-0.933926	-0.087121
6	2878	1495	0.624558	0.721839
7	4440	564	0.055488	0.823045
8	1682	1938	1.135343	0.010104
9	1921	1942	0.099393	0.298868
10	2109	2241	-1.661886	2.055617

How does trDFpived2 look? OK, let's drop randy from trDFpiv and try some other things. Feeling lucky? Then do trDFpiv.drop('randy',axis=1,inplace=True). You can also stack and unstack DataFrames. These methods come in handy when you need to shape some data in a particular way to be input to an algorithm. Let's aggregate some of the xyzcustnew data (see above) to get a DataFrame we can stack and unstack:

In [49]:
```
xyzdata=xyzcustnew[['BUYER_STATUS','heavyCat','CHANNEL_ACQUISITION']]
```

Use xyzdata because it's just easier. It has just the three columns we're now going to work with.

In [50]:
```
xyzgrouped=xyzdata.groupby(['BUYER_STATUS','heavyCat','CHANNEL_ACQUISITI
ON'])
```

In [53]:
```
xyzgrouped.head(10)
```

Out[53]:

	BUYER_STATUS	heavyCat	CHANNEL_ACQUISITION
0	INACTIVE	regular	IB
1	ACTIVE	heavy	RT
2	ACTIVE	regular	RT
3	INACTIVE	regular	RT
4	ACTIVE	regular	RT
...	...	...	...
657	INACTIVE	regular	CB
659	ACTIVE	regular	CB
695	ACTIVE	heavy	CB
741	ACTIVE	heavy	CB
810	ACTIVE	heavy	CB

120 rows × 3 columns

In [51]:
```
xyzCountData = xyzgrouped.size()          # a MultiIndexed Series of count
s
```

In [52]:
```
print(xyzCountData.unstack())
```

```
CHANNEL_ACQUISITION     CB    IB    RT
BUYER_STATUS heavyCat
ACTIVE       regular   443  1112  7393
             heavy     356   703  3325
INACTIVE     regular   691  1249  7056
LAPSED       regular   372  1111  6368
```

xyzCountData is a Series with a MultiIndex, and so it can be unstacked, changing it from tall and narrow to short and wide. Note that by default, only the lowest level of the MultiIndex is used for unstacking. Do you know why there are no heavy buyers in the INACTIVE or LAPSED categories? Let's “restack” this into a different version of xyzCountData:

In [54]:
```
unStackxyz=xyzCountData.unstack()                    # what we had just above
```

In [55]:
```
unStackxyz.T.stack()            # .T is the transpose
```

Out[55]:

	BUYER_STATUS	ACTIVE	INACTIVE	LAPSED
CHANNEL_ACQUISITION	**heavyCat**			
CB	regular	443	691.0	372.0
	heavy	356	NaN	NaN
IB	regular	1112	1249.0	1111.0
	heavy	703	NaN	NaN
RT	regular	7393	7056.0	6368.0
	heavy	3325	NaN	NaN

Note how in the above, combinations of the levels of the three variables that do not actually occur in the data are given an NaN, a missing value. NaN means "not a number." The cells are stacked using levels of BUYER_STATUS within levels of CHANNEL_ACQUISITION. Try doing unStackxyz.T.stack(1) to get stacking by heavyCat instead of by BUYER_STATUS. Here again, cells do not have observations are given a NaN. The unstack method can return a stacked object as it was when it was stacked, but it can also return it in a different unstacked form. For example, see what this does:

In [49]:
```
unStackxyz.T.stack(0).unstack(1)
```

Out[49]:

heavyCat	**regular**			**heavy**		
BUYER_STATUS	**ACTIVE**	**INACTIVE**	**LAPSED**	**ACTIVE**	**INACTIVE**	**LAPSED**
CHANNEL_ACQUISITION						
CB	443	691	372	356.0	NaN	NaN
IB	1112	1249	1111	703.0	NaN	NaN
RT	7393	7056	6368	3325.0	NaN	NaN

You can stack or unstack on multiple levels at one time. See what this does for you:

In [56]:
```
unStackxyz.T.stack(level=['heavyCat','BUYER_STATUS'])
```

Out[56]:
```
CHANNEL_ACQUISITION  heavyCat  BUYER_STATUS
CB                   regular   ACTIVE           443.0
                               INACTIVE         691.0
                               LAPSED           372.0
                     heavy     ACTIVE           356.0
IB                   regular   ACTIVE          1112.0
                               INACTIVE        1249.0
                               LAPSED          1111.0
                     heavy     ACTIVE           703.0
RT                   regular   ACTIVE          7393.0
                               INACTIVE        7056.0
                               LAPSED          6368.0
                     heavy     ACTIVE          3325.0
dtype: float64
```

and compare to:

In [57]:
```
unStackxyz.T.stack(level=['BUYER_STATUS','heavyCat'])
```

Out[57]:
```
CHANNEL_ACQUISITION  BUYER_STATUS  heavyCat
CB                   ACTIVE        regular      443.0
                                   heavy        356.0
                     INACTIVE      regular      691.0
                     LAPSED        regular      372.0
IB                   ACTIVE        regular     1112.0
                                   heavy        703.0
                     INACTIVE      regular     1249.0
                     LAPSED        regular     1111.0
RT                   ACTIVE        regular     7393.0
                                   heavy       3325.0
                     INACTIVE      regular     7056.0
                     LAPSED        regular     6368.0
dtype: float64
```

The pandas melt() method provides some similar functionality. You can use it to turn a short and wide DataFrame into a taller, narrower one by identifying columns that contain values to be used as record identifiers. Let's go back to the xyzcustnew data and select a few columns from it to do some melting on:

In [58]:
```
xyzcust=xyzcustnew[['BUYER_STATUS','heavyCat','LTD_SALES']].copy()
```

Now, let's melt xyzcust so that BUYER_STATUS and heavyCat become identifiers:

In [59]:
```
xyzcustm=pd.melt(xyzcust,id_vars=['BUYER_STATUS','heavyCat'],var_name="L
TD_SALES")
```

xyzcustm will look something like:

In [60]:
```
print(xyzcustm)
```

```
      BUYER_STATUS heavyCat  LTD_SALES   value
0         INACTIVE  regular  LTD_SALES    90.0
1           ACTIVE    heavy  LTD_SALES  4227.0
2           ACTIVE  regular  LTD_SALES   420.0
3         INACTIVE  regular  LTD_SALES  6552.0
4           ACTIVE  regular  LTD_SALES   189.0
...            ...      ...        ...     ...
30174       ACTIVE  regular  LTD_SALES  2736.0
30175       ACTIVE  regular  LTD_SALES  2412.0
30176     INACTIVE  regular  LTD_SALES   429.0
30177     INACTIVE  regular  LTD_SALES   651.0
30178       ACTIVE    heavy  LTD_SALES  4527.0

[30179 rows x 4 columns]
```

You'll probably realize that the leftmost column is a simple numerical index that this pandas method created. There's a pandas method called wide_to_long that works similarly, but can be a little easier to use. Give it a try using xyzcust or the DataFrame of your choice. So at this point we've pivoted, grouped, and reshaped. The pivoting example we did was pretty simple. pandas also provides a method called pivot_table that provides considerable flexibility in terms of how data can be reorganized and summarized. Let's consider the xyzcustnew data once again. Suppose we want to average YTD_SALES_2009 by BUYER_STATUS, CHANNEL_ACQUISITION, and heavyCAT. WE could do:

In [55]:

```
pd.pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS',
'heavyCat'],columns=['CHANNEL_ACQUISITION'])
```

Out[55]:

	CHANNEL_ACQUISITION	CB	IB	RT
BUYER_STATUS	**heavyCat**			
ACTIVE	**regular**	205.334086	191.047662	167.993913
	heavy	2397.606742	1251.559033	1158.506165
INACTIVE	**regular**	0.000000	0.000000	0.000000
	heavy	NaN	NaN	NaN
LAPSED	**regular**	0.000000	0.000000	0.000000
	heavy	NaN	NaN	NaN

Do you see some rows in the result that only have zeros? Why are they there? Or, try doing:

In [56]:

```
pd.pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS'
],columns=['heavyCat','CHANNEL_ACQUISITION'])
```

Out[56]:

heavyCat	regular			heavy		
CHANNEL_ACQUISITION	**CB**	**IB**	**RT**	**CB**	**IB**	**RT**
BUYER_STATUS						
ACTIVE	205.334086	191.047662	167.993913	2397.606742	1251.559033	1158.506
INACTIVE	0.000000	0.000000	0.000000	NaN	NaN	N
LAPSED	0.000000	0.000000	0.000000	NaN	NaN	N

Why are there NaN's? pivot_table defaults to taking the mean (using np.mean) of the groups it defines. If you want some other aggregation instead, you can define it as a keyword parameter, e.g. aggfunc=np.sum:

In [57]:

```
pd.pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS'
],columns=['heavyCat','CHANNEL_ACQUISITION'],aggfunc=np.sum)
```

Out[57]:

heavyCat	regular			heavy		
CHANNEL_ACQUISITION	**CB**	**IB**	**RT**	**CB**	**IB**	**RT**
BUYER_STATUS						
ACTIVE	90963.0	212445.0	1241979.0	853548.0	879846.0	3852033.0
INACTIVE	0.0	0.0	0.0	NaN	NaN	NaN
LAPSED	0.0	0.0	0.0	NaN	NaN	NaN

You can also add margins to pivot_tables by using the margins=True option. For example, to get row and column totals:

In [58]:
```
pd.pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS'
],columns=['heavyCat','CHANNEL_ACQUISITION'],aggfunc=np.sum,margins=True
)
```

Out[58]:

heavyCat	regular			heavy			All
CHANNEL_ACQUISITION	CB	IB	RT	CB	IB	RT	
BUYER_STATUS							
ACTIVE	90963.0	212445.0	1241979.0	853548.0	879846.0	3852033.0	7130814.0
INACTIVE	0.0	0.0	0.0	NaN	NaN	NaN	0.0
LAPSED	0.0	0.0	0.0	NaN	NaN	NaN	0.0
All	90963.0	212445.0	1241979.0	853548.0	879846.0	3852033.0	7130814.0

Should give you the same table as above but with row and column totals added. It has probably dawned on you that you can manipulate data objects in many different ways to group them and to apply descriptive statistics to them. Let's group xyz customers using BUYER_STATUS and heavyCat:

In [59]:
```
xyzGrouper=xyzcustnew.groupby(['BUYER_STATUS','heavyCat'])
```

groupby can apply conventional as well as custom functions to aggregated data. For example:

In [60]:
```
xyzGrouper.agg({'YTD_SALES_2009': [np.mean, np.std],'LTD_SALES':[np.mean
,np.std]})
```

Out[60]:

		YTD_SALES_2009		LTD_SALES	
		mean	std	mean	std
BUYER_STATUS	heavyCat				
ACTIVE	regular	172.707532	107.584023	1001.845105	1466.075631
	heavy	1274.048130	5434.616517	4096.179745	34210.646330
INACTIVE	regular	0.000000	0.000000	568.014784	850.966479
LAPSED	regular	0.000000	0.000000	841.467329	1374.447756

calculates the mean and standard deviation of YTD_SALES_2009 and LTD_SALES for each of the groups defined in xyzGrouper. Note the little dict with a couple of key/value pairs there in the curly brackets, the {}. Try using a version of this command to get statistics for the columns YTD_TRANSACTIONS_2009 and LTD_TRANSACTIONS. These are both count variables. What descriptive statistics do you think are appropriate for summarizing them? Note that you can apply custom functions to data aggregates. Suppose we wanted to compute the coefficient of variation,,"CV," for data. The CV is a standardized measure of dispersion, and is the ratio of the standard deviation to to the mean. It's estimated by the ratio of the estimates of these two statistics. We could write our own function do do this:

In [63]:
```
def coefV(x):                         # a baby CV function that accepts a
 sequence
    return np.std(x)/np.mean(x)
```

This will work assuming that the mean and std numpy methods are available in this function's namespace, of course. Note that our baby function doesn't do anything smart regarding missing values and other inconveniences, but it's good enough to demonstrate what we want, here. What do you think it means if what it produces is negative? How could that happen? We can apply this function to selected groups. Here we apply it to customers grouped by BUYER_STATUS. Let's first get a simpler DataFrame to fiddle with:

In [64]:
```
buyerStats=xyzcustnew[['BUYER_STATUS','LTD_SALES','LTD_TRANSACTIONS']]
buyerGrouper=buyerStats.groupby(['BUYER_STATUS'])
buyerGrouper.agg(coefV)
```

Out[64]:

	LTD_SALES	LTD_TRANSACTIONS
BUYER_STATUS		
ACTIVE	9.758480	1.153501
INACTIVE	1.498058	0.784441
LAPSED	1.633290	0.987139

Did you get a table of CV's? We could combine our own function or functions with existing functions and apply them on a group by group basis. Let's play with a function that returns 5th and 95th percentiles of some data:

In [62]:
```
def ptiles(x):
    p5=np.percentile(x,5)
    p95=np.percentile(x,95)
    return p5, p95
```

There's our toy function. coefV, it may break with “bad” data. (So, watch out.) What kind of object does ptiles return? Now, applying np.mean and ptiles:

In [65]:
```
buyerGrouper.agg([np.mean, ptiles])
```

Out[65]:

	LTD_SALES		LTD_TRANSACTIONS	
	mean	ptiles	mean	ptiles
BUYER_STATUS				
ACTIVE	2019.364086	(81.0, 6544.349999999997)	6.935794	(1.0, 20.0)
INACTIVE	568.014784	(60.0, 1776.0)	2.263895	(1.0, 6.0)
LAPSED	841.467329	(63.0, 2904.0)	3.498280	(1.0, 9.0)

In [ ]:

What kind of object is the above command printing out for you? You can select particular results from this, of course, e.g.:

In [66]:
```
buyerGrouper.agg([np.mean,ptiles]).loc['ACTIVE','LTD_SALES']
```

Out[66]:
```
mean                           2019.36
ptiles    (81.0, 6544.349999999997)
Name: ACTIVE, dtype: object
```

As a quick little exercise to do on you own, write a tiny function that calculates the “interquartile range,” or IQR, for data, and then apply it to the above data. The IQR is the difference between the 75th and the 25th percentile values. Well, that wraps it up for this, and last, Python Practice. No surprisingly, there's a lot more to data management using Python and packages like Pandas, and there's something new all the time. If you're an R user, and you use it on Linux or OS X, you'll want to check out the package rpy2, which provides some capability for transferring data between R and Python. It's under development, and the plan is that it will eventually allow doing things like calling R functions from within Python. It is apparently pretty tough to install and use from in Windows at the present time.

Requirements :

1. Get the trDFgrouped data starting from the May heavy day counts to the August heavy counts
2. Group xyz customers using BUYER_STATUS, heavyCat, and ZIP, and apply np.sum function on the aggregated data for YTD_SALES_2009 and LTD_SALES columns

1

Get the trDFgrouped data starting from the May heavy day counts to the August heavy counts

In [73]:
```
trDF=DataFrame()
trDF['date'] = trCountsChrono.index
trDF['transactions'] = trCountsChrono.values
heavyLight=lambda x  : x >= trMed and 'heavy' or 'light'  # an example a
non function
trDF['vol']=trDF.transactions.map(heavyLight)   # 'vol' is the heavy/lig
ht column
trDF['monum']=trDF.date.dt.month        # .dt is the datetime accessor
trDFnd=trDF.drop('date',axis=1) # axis=1 means here a column is selected
to drop
trDFgrouped=trDFnd.groupby(['monum','vol']).sum()

trDFgrouped.iloc[8:15]
```

Out[73]:

		transactions
monum	**vol**	
5	heavy	2172
	light	2076
6	heavy	2878
	light	1495
7	heavy	4440
	light	564
8	heavy	1682

2

Group xyz customers using BUYER_STATUS, heavyCat, and ZIP, and apply np.sum function on the aggregated data for YTD_SALES_2009 and LTD_SALES columns

In [80]:
```
xyzdata=xyzcustnew[['BUYER_STATUS','heavyCat','ZIP', 'YTD_SALES_2009',
'LTD_SALES']]
xyzdataGrouper = xyzdata.groupby(['BUYER_STATUS', 'heavyCat', 'ZIP'])
xyzdataGrouper.agg({'YTD_SALES_2009':[np.sum], 'LTD_SALES':[np.sum]})
```

Out[80]:

			YTD_SALES_2009	LTD_SALES
			sum	sum
BUYER_STATUS	heavyCat	ZIP		
ACTIVE	regular	0	NaN	NaN
		60056	68913.0	332196.0
		60060	68520.0	339567.0
		60061	68328.0	400569.0
		60062	141237.0	762387.0
...	...	...	...	...
LAPSED	heavy	60095	NaN	NaN
		60096	NaN	NaN
		60097	NaN	NaN
		60098	NaN	NaN
		60192	NaN	NaN

222 rows × 2 columns