Siddikov Exercise 5 version b

August 16, 2019

Deliverables:

- Submit a single zip-compressed file that has the name: YourLastName_Exercise_5 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
3. You can zip these 2 files if you like; use the same naming convention for the zip file.

1 Objectives:

In this exercise, you will:

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

Formatting Python Code When programming in Python, refer to Kenneth Reitz' PEP 8: The Style Guide for Python Code: 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 (Links to an external site.)Links to an external site. Comment often and in detail.

2 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:

[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

import matplotlib

%matplotlib inline
```

[2]:
```
engine=create_engine('sqlite:///xyz.db')          # the db is in my current␣
↪working directory
```

[3]:
```python
# .info gives same feedback as .dtype and .count
xyzcustnew=pd.read_sql_table('xyzcust',engine)
xyzcustnew.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30179 entries, 0 to 30178
Data columns (total 11 columns):
index                    30179 non-null int64
ACCTNO                   30179 non-null object
ZIP                      30179 non-null int64
ZIP4                     30179 non-null int64
LTD_SALES                30179 non-null float64
LTD_TRANSACTIONS         30179 non-null int64
YTD_SALES_2009           30179 non-null float64
YTD_TRANSACTIONS_2009    30179 non-null int64
CHANNEL_ACQUISITION      30179 non-null object
BUYER_STATUS             30179 non-null object
ZIP9_SUPERCODE           30179 non-null int64
dtypes: float64(2), int64(6), object(3)
memory usage: 2.5+ MB
```

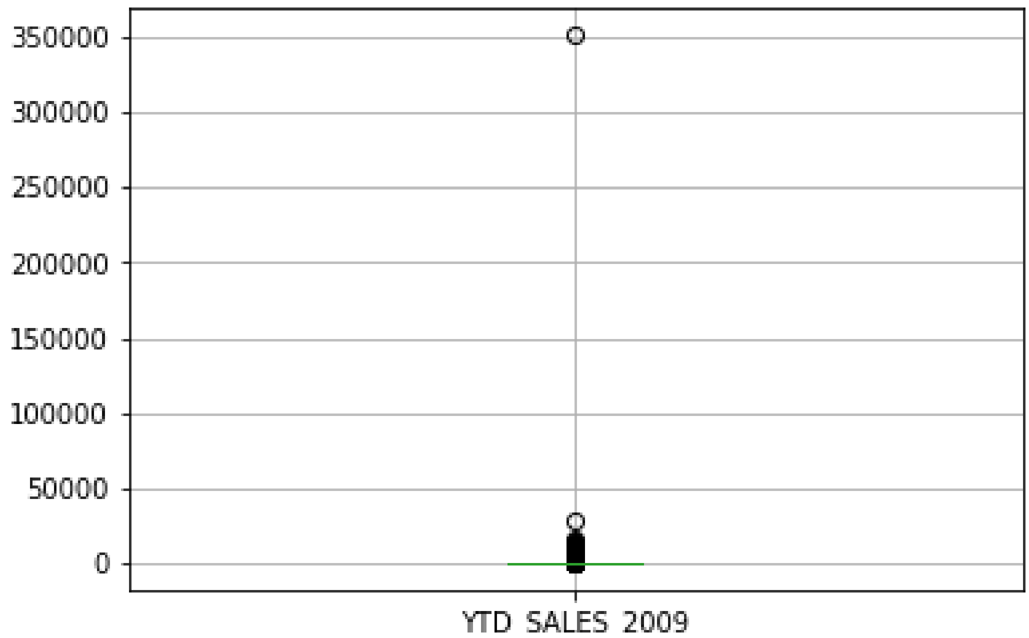
[4]:
```python
# heavyCut is a constant and was decided as where the data
# should be cut
heavyCut= 423
```

[5]:
```python
# look at characteristics
xyzcustnew['YTD_SALES_2009'].describe()
```

[5]:
```
count     30179.000000
mean        236.283972
std        2117.042293
min           0.000000
25%           0.000000
50%           0.000000
75%         207.000000
max      351000.000000
Name: YTD_SALES_2009, dtype: float64
```

[6]:
```python
# look at spread of 2009 sales
xyzcustnew.boxplot(column='YTD_SALES_2009')
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1fecbfa3b38>
```



[7]:
```
# create a categorial variable of either a 1 or 0 based upon the value of␣
↪heavyCut
# YTD_SALES_2009 greater than the heavyCut value will be assigned a 1
heavyCat=Categorical(np.where(xyzcustnew.YTD_SALES_2009>heavyCut,1,0))
heavyCat.describe()
```

[7]:
```
            counts     freqs
categories
0            25795  0.854733
1             4384  0.145267
```

[8]:
```
# be more descriptive than a 1 or a 0
heavyCat.rename_categories(['regular','heavy'],inplace=True)
heavyCat.describe()
```

[8]:
```
            counts     freqs
categories
regular      25795  0.854733
heavy         4384  0.145267
```

[9]:
```
# look at the first ten records
heavyCat[:10]
```

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

[10]:
```
# create a new column with this variable
xyzcustnew['heavyCat']=heavyCat
```

[11]:
```
# a dummy variable marks the field as either 1 or 0
buyerType=pd.get_dummies(heavyCat)
buyerType[:3]
```

[11]:
```
   regular  heavy
0        1      0
1        0      1
2        1      0
```

[12]:
```
# create new columns
xyzcustnew['typeReg']=buyerType['regular']
xyzcustnew['typeHeavy']=buyerType['heavy']
```

[13]:
```
xyzcustnew.columns
```

[13]:
```
Index(['index', 'ACCTNO', 'ZIP', 'ZIP4', 'LTD_SALES', 'LTD_TRANSACTIONS',
       'YTD_SALES_2009', 'YTD_TRANSACTIONS_2009', 'CHANNEL_ACQUISITION',
       'BUYER_STATUS', 'ZIP9_SUPERCODE', 'heavyCat', 'typeReg', 'typeHeavy'],
      dtype='object')
```

[14]:
```
# look at new variables
xyzcustnew.head()
```

[14]:
```
   index     ACCTNO    ZIP  ZIP4  LTD_SALES  LTD_TRANSACTIONS  YTD_SALES_2009  \
0      0  WDQQLLDQL  60084  5016       90.0                 1             0.0
1      1  WQWAYHYLA  60091  1750     4227.0                 9          1263.0
2      2  GSHAPLHAW  60067   900      420.0                 3           129.0
3      3  PGGYDYWAD  60068  3838     6552.0                 6             0.0
4      4  LWPSGPLLS  60090  3932      189.0                 3            72.0

   YTD_TRANSACTIONS_2009 CHANNEL_ACQUISITION BUYER_STATUS  ZIP9_SUPERCODE  \
0                      0                  IB     INACTIVE       600845016
1                      3                  RT       ACTIVE       600911750
2                      1                  RT       ACTIVE       600670900
3                      0                  RT     INACTIVE       600683838
4                      1                  RT       ACTIVE       600903932

  heavyCat  typeReg  typeHeavy
0  regular        1          0
1    heavy        0          1
2  regular        1          0
3  regular        1          0
4  regular        1          0
```

[15]:
```
# for this exercises we need to create trCountsChrono object

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

print(daystr[0],mostr[0],yearstr[0],xyztrans.TRANDATE[0])
```

```
09 JUN 2009 09JUN2009
```

[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
print(transtr[0])
```

```
2009-6-09
```

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

[17]:
```
# convert to datetime values
trDateTime=pd.to_datetime(transtr)
```

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

[18]:
```
2009-12-19    877
2009-12-21    836
2009-12-12    782
2009-12-23    765
2009-12-20    744
2009-12-22    717
2009-12-18    708
2009-12-14    615
2009-12-15    599
2009-12-16    571
2009-12-11    568
2009-11-21    561
2009-12-13    542
2009-11-22    507
2009-12-10    504
2009-12-04    488
2009-11-25    451
2009-12-24    425
2009-11-23    421
2009-11-27    419
```

```
2009-11-24    412
2009-04-10    404
2009-11-28    402
2009-11-14    402
2009-12-09    401
2009-05-09    398
2009-12-08    397
2009-11-07    394
2009-12-07    372
2009-01-17    372
             ...
2009-02-26     47
2009-04-30     47
2009-02-21     39
2009-03-01     36
2009-03-10     33
2009-06-14     29
2009-11-26     26
2009-03-16     24
2009-07-14     24
2009-06-15     24
2009-08-30     23
2009-11-04     23
2009-02-25     22
2009-02-02     22
2009-10-17     21
2009-03-08     20
2009-03-27     19
2009-02-03     19
2009-02-01     18
2009-03-11     17
2009-02-07     16
2009-07-02     15
2009-06-13     14
2009-07-16     14
2009-12-25     11
2009-08-15     11
2009-04-12     10
2009-10-13     10
2009-08-21      6
2009-03-15      5
Name: TRANDATE, Length: 365, dtype: int64
```

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.

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

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

[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:

[21]:
```
# initialize a dataframe
trDF=DataFrame()
```

[22]:
```
trDF
```

[22]:
```
Empty DataFrame
Columns: []
Index: []
```

[23]:
```
# load dataframe with 2 columns
trDF['date'] = trCountsChrono.index
trDF['transactions'] = trCountsChrono.values
trDF.columns
```

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

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

[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
```

[25]:
```
trDF.dtypes
```

[25]:
```
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:

[26]:
```
trMed=trDF.transactions.median()                    # here's the median
trMed
```

[26]:
```
136.0
```

[27]:
```
# if the value is greater than or equal to the median, then heavy
heavyLight = lambda x  : x >= trMed and 'heavy' or 'light'  # an example anon␣
 ↪function
```

[28]:
```
# use map to call lambda
trDF['vol']=trDF.transactions.map(heavyLight)        # 'vol' is the heavy/light␣
 ↪column
trDF.head(10)
```

[28]:

	date	transactions	vol
0	2009-01-01	176	heavy
1	2009-01-02	305	heavy
2	2009-01-03	365	heavy
3	2009-01-04	231	heavy
4	2009-01-05	144	heavy
5	2009-01-06	188	heavy
6	2009-01-07	166	heavy
7	2009-01-08	52	light
8	2009-01-09	194	heavy
9	2009-01-10	166	heavy

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:

[29]:
```
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:

[30]:
```
# making a copy of trDF while also dropping date as a column
# axis=1 means here a column is selected to drop
trDFnd=trDF.drop('date',axis=1)
trDFnd.head()
```

[30]:

	transactions	vol	monum
0	176	heavy	1
1	305	heavy	1
2	365	heavy	1
3	231	heavy	1
4	144	heavy	1

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

[31]:
```
trDFgrouped = trDFnd.groupby(['monum','vol']).sum()
trDFgrouped.head()
```

[31]:
```
             transactions
monum vol
1     heavy          5255
      light           572
2     heavy           761
      light          1625
3     heavy          1130
```

[32]:
```
# check out the indexes
trDFgrouped.index.levels
```

[32]:
```
FrozenList([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], ['heavy', 'light']])
```

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$_{a}rrays)ofindexlevels,orbyusingtuples(withMultiIndex.from_{t}uples).Inbothcasesallcombinationso$

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:

[33]:
```
trDFgrouped.loc[11,'heavy']
```

[33]:
```
transactions    8402
Name: (11, heavy), dtype: int64
```

[34]:
```
# first six months of data
trDFgrouped.loc[list(range(1,7))]
```

[34]:
```
             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
6     heavy          2878
```

```
        light          1495
```

[35]:
```
# or the first 6 rows of data:
trDFgrouped.iloc[0:6]                    # .iloc here, but .loc above.
```

[35]:
```
                transactions
monum vol
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:

[36]:
```
trDFgrouped[(3,'light'):(7,'heavy')]
```

[36]:
```
                transactions
monum vol
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:

[37]:
```
trDFgrouped[(3,'light'):6]
```

[37]:
```
                transactions
monum vol
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:

[38]:
```
trDFgrouped.xs('light',level='vol')
```

[38]:
```
       transactions
monum
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:

[39]:
```
trDFgrouped.xs('light',level='vol').T                # the transpose of the␣
↪above
```

[39]:
```
monum            1     2     3     4     5     6    7     8     9     10   11  \
transactions   572  1625  1664  1727  2076  1495  564  1938  1942  2241  49

monum           12
transactions   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.

[40]:
```
mo=trDFgrouped.index.get_level_values(0)          # the month numbers
```

[41]:
```
volType=trDFgrouped.index.get_level_values(1)         # vol
```

[42]:
```
trDFpiv=DataFrame({'month':mo,'vol': volType, 'transactions':trDFgrouped.
↪transactions})                    # data as a dict
```

[43]:
```
trDFpiv
```

[43]:
```
             month  transactions    vol
monum vol
1     heavy      1          5255  heavy
      light      1           572  light
2     heavy      2           761  heavy
      light      2          1625  light
3     heavy      3          1130  heavy
      light      3          1664  light
4     heavy      4          2327  heavy
      light      4          1727  light
5     heavy      5          2172  heavy
      light      5          2076  light
6     heavy      6          2878  heavy
      light      6          1495  light
7     heavy      7          4440  heavy
```

```
      light       7           564  light
8     heavy       8          1682  heavy
      light       8          1938  light
9     heavy       9          1921  heavy
      light       9          1942  light
10    heavy      10          2109  heavy
      light      10          2241  light
11    heavy      11          8402  heavy
      light      11            49  light
12    heavy      12         13168  heavy
      light      12           257  light
```

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:

[44]:
```python
trDFpived = trDFpiv.pivot(index='month',columns='vol',values='transactions')
trDFpived
```

[44]:
```
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
11      8402     49
12     13168    257
```

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:

[45]:
```python
trDFpiv['randy']=np.random.randn(len(trDFpiv))
trDFpiv.head()
```

[45]:
```
             month  transactions    vol     randy
monum vol
1     heavy      1          5255  heavy  2.072689
      light      1           572  light  1.551954
2     heavy      2           761  heavy  1.611733
      light      2          1625  light -0.551663
3     heavy      3          1130  heavy  0.224400
```

Now pivot trDFpiv like:

[46]:
```python
trDFpived2=trDFpiv.pivot(index='month',columns='vol')
trDFpived2.head()
```

[46]:
```
      transactions           randy
vol          heavy light     heavy     light
month
1             5255   572  2.072689  1.551954
2              761  1625  1.611733 -0.551663
3             1130  1664  0.224400  1.429708
4             2327  1727 -0.063512  1.104748
5             2172  2076  1.336558 -1.077305
```

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:

[47]:
```
# remember we read in xyzcustnew from xyz.db
xyzdata = xyzcustnew[['BUYER_STATUS','heavyCat','CHANNEL_ACQUISITION']]
xyzdata.head()
```

[47]:
```
  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
```

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

[48]:
```
xyzgrouped = xyzdata.groupby(['BUYER_STATUS','heavyCat','CHANNEL_ACQUISITION'])
```

[49]:
```
xyzCountData = xyzgrouped.size()        # a MultiIndexed Series of counts
xyzCountData
```

[49]:
```
BUYER_STATUS  heavyCat  CHANNEL_ACQUISITION
ACTIVE        regular   CB                      443
                        IB                     1112
                        RT                     7393
              heavy     CB                      356
                        IB                      703
                        RT                     3325
INACTIVE      regular   CB                      691
                        IB                     1249
                        RT                     7056
LAPSED        regular   CB                      372
                        IB                     1111
                        RT                     6368
dtype: int64
```

[50]:
```
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:

[51]:
```
unStackxyz = xyzCountData.unstack()                # what we had just above
unStackxyz
```

[51]:
```
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
```

[52]:
```
unStackxyz.T.stack()                # .T is the transpose
```

[52]:
```
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$_{S}TATUSwithinlevelsofCHANNEL_{A}CQUISITION$.

Try doing unStackxyz.T.stack(1) to get stacking by heavyCat instead of by BUYER$_{S}TATUS.Hereagain, cellsdonothaveobservationsaregivenaNaN$.

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:

[53]:
```
unStackxyz.T.stack(0).unstack(1)
```

[53]:
```
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:

[54]:
```
unStackxyz.T.stack(level=['heavyCat','BUYER_STATUS'])
```

[54]:
```
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:

[55]:
```
unStackxyz.T.stack(level=['BUYER_STATUS','heavyCat'])
```

[55]:
```
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:

[56]:
```
xyzcust = xyzcustnew[['BUYER_STATUS','heavyCat','LTD_SALES']].copy()
```

Now, let's melt xyzcust so that BUYER$_STATUSandheavyCatbecomeidentifiers$:

xyzcustm will look something like:

[58]:
```
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
5	ACTIVE	regular	LTD_SALES	4278.0
6	ACTIVE	heavy	LTD_SALES	1869.0
7	ACTIVE	regular	LTD_SALES	33.0
8	INACTIVE	regular	LTD_SALES	735.0
9	INACTIVE	regular	LTD_SALES	468.0
10	ACTIVE	regular	LTD_SALES	804.0
11	LAPSED	regular	LTD_SALES	219.0
12	ACTIVE	heavy	LTD_SALES	3240.0
13	INACTIVE	regular	LTD_SALES	180.0
14	ACTIVE	regular	LTD_SALES	423.0
15	INACTIVE	regular	LTD_SALES	306.0
16	LAPSED	regular	LTD_SALES	1002.0
17	ACTIVE	regular	LTD_SALES	1155.0
18	ACTIVE	regular	LTD_SALES	612.0
19	ACTIVE	regular	LTD_SALES	633.0
20	INACTIVE	regular	LTD_SALES	114.0
21	ACTIVE	regular	LTD_SALES	294.0
22	INACTIVE	regular	LTD_SALES	849.0
23	INACTIVE	regular	LTD_SALES	72.0
24	ACTIVE	heavy	LTD_SALES	3411.0
25	ACTIVE	heavy	LTD_SALES	1023.0
26	LAPSED	regular	LTD_SALES	873.0
27	ACTIVE	heavy	LTD_SALES	2778.0
28	ACTIVE	heavy	LTD_SALES	2676.0
29	LAPSED	regular	LTD_SALES	528.0
...	...	...	...	...
30149	ACTIVE	regular	LTD_SALES	861.0
30150	ACTIVE	regular	LTD_SALES	837.0
30151	ACTIVE	regular	LTD_SALES	2478.0
30152	ACTIVE	regular	LTD_SALES	84.0
30153	ACTIVE	heavy	LTD_SALES	2877.0
30154	INACTIVE	regular	LTD_SALES	1611.0
30155	LAPSED	regular	LTD_SALES	1860.0
30156	LAPSED	regular	LTD_SALES	48.0
30157	ACTIVE	regular	LTD_SALES	195.0
30158	LAPSED	regular	LTD_SALES	60.0
30159	INACTIVE	regular	LTD_SALES	252.0
30160	LAPSED	regular	LTD_SALES	594.0
30161	LAPSED	regular	LTD_SALES	1272.0
30162	ACTIVE	heavy	LTD_SALES	2184.0
30163	ACTIVE	regular	LTD_SALES	759.0
30164	INACTIVE	regular	LTD_SALES	756.0
30165	ACTIVE	regular	LTD_SALES	1365.0
30166	ACTIVE	heavy	LTD_SALES	2490.0
30167	ACTIVE	heavy	LTD_SALES	438.0

```
30168     INACTIVE  regular  LTD_SALES   549.0
30169       ACTIVE  regular  LTD_SALES   150.0
30170       ACTIVE  regular  LTD_SALES    93.0
30171     INACTIVE  regular  LTD_SALES   834.0
30172     INACTIVE  regular  LTD_SALES   147.0
30173       LAPSED  regular  LTD_SALES   816.0
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$_{t}o_{l}ongthatworkssimilarly,butcanbealittleeasiertouse.GiveitatryusingxyzcustortheDataFrameofyourchoice.$

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$_{t}ablethatprovidesconsiderableflexibilityintermsofhowdatacanbereorganizedandsummarized.Let'sconsiderthex$

[59]:
```
pd.
→pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS','heavyCat'],columns=['
```

[59]:
```
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:

[60]:
```
pd.
→pivot_table(xyzcustnew,values='YTD_SALES_2009',index=['BUYER_STATUS'],columns=['heavyCat','
```

[60]:
```
heavyCat                regular                                 heavy  \
CHANNEL_ACQUISITION          CB          IB          RT            CB
BUYER_STATUS
ACTIVE               205.334086  191.047662  167.993913   2397.606742
INACTIVE               0.000000    0.000000    0.000000           NaN
LAPSED                 0.000000    0.000000    0.000000           NaN

heavyCat
CHANNEL_ACQUISITION           IB           RT
BUYER_STATUS
ACTIVE               1251.559033  1158.506165
INACTIVE                     NaN          NaN
```

```
LAPSED                          NaN           NaN
```

Why are there NaN's?

pivot$_{t}able defaults to taking the mean (using np.mean) of the groups it defines. If you want some other aggregation inste$ $np.sum:$

[61]:
```
pd.pivot_table(xyzcustnew,values='YTD_SALES_2009',
                ␣
 ↪index=['BUYER_STATUS'],columns=['heavyCat','CHANNEL_ACQUISITION'],aggfunc=np.
 ↪sum)
```

[61]:
```
heavyCat              regular                              heavy            \
CHANNEL_ACQUISITION        CB        IB         RT          CB         IB
BUYER_STATUS
ACTIVE                90963.0  212445.0  1241979.0  853548.0   879846.0
INACTIVE                  0.0       0.0        0.0       NaN        NaN
LAPSED                    0.0       0.0        0.0       NaN        NaN

heavyCat
CHANNEL_ACQUISITION         RT
BUYER_STATUS
ACTIVE                3852033.0
INACTIVE                    NaN
LAPSED                      NaN
```

You can also add margins to pivot$_{t}ables by using the margins = True option. For example, to get row and column totals:$

[62]:
```
heavyCat              regular                              heavy            \
CHANNEL_ACQUISITION        CB        IB         RT          CB         IB
BUYER_STATUS
ACTIVE                90963.0  212445.0  1241979.0  853548.0   879846.0
INACTIVE                  0.0       0.0        0.0       NaN        NaN
LAPSED                    0.0       0.0        0.0       NaN        NaN
All                   90963.0  212445.0  1241979.0  853548.0   879846.0

heavyCat                                   All
CHANNEL_ACQUISITION         RT
BUYER_STATUS
ACTIVE                3852033.0  7130814.0
INACTIVE                    NaN        0.0
LAPSED                      NaN        0.0
All                   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$_{S}TATUS and heavyCat:$

[63]:
```
xyzGrouper=xyzcustnew.groupby(['BUYER_STATUS','heavyCat'])
```

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

[64]:
```python
xyzGrouper.agg({'YTD_SALES_2009': [np.mean, np.std],'LTD_SALES':[np.mean,np.std]})
```

[64]:
```
                       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$_{S}ALES_{2}009andLTD_{S}ALESforeachofthegroupsdefinedinxyzGr$

Try using a version of this command to get statistics for the columns YTD$_{T}RANSACTIONS_{2}009andLTD_{T}RANSACTIONS.Thesearebothcountvariables.Whatdescriptivestatisticsdoyo$

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:

[65]:
```python
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$_{S}TATUS.Let'sfirstgetasimplerDataFrametofiddlewith$:

[66]:
```python
buyerStats=xyzcustnew[['BUYER_STATUS','LTD_SALES','LTD_TRANSACTIONS']]
buyerGrouper=buyerStats.groupby(['BUYER_STATUS'])
buyerGrouper.agg(coefV)
```

[66]:
```
              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:

[67]:
```python
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:

[68]:
```
buyerGrouper.agg([np.mean, ptiles])
```

[68]:
```
                LTD_SALES                             LTD_TRANSACTIONS  \
                     mean                     ptiles              mean
BUYER_STATUS
ACTIVE        2019.364086  (81.0, 6544.349999999997)          6.935794
INACTIVE       568.014784              (60.0, 1776.0)          2.263895
LAPSED         841.467329              (63.0, 2904.0)          3.498280


                   ptiles
BUYER_STATUS
ACTIVE        (1.0, 20.0)
INACTIVE       (1.0, 6.0)
LAPSED         (1.0, 9.0)
```

What kind of object is the above command printing out for you?

You can select particular results from this, of course, e.g.:

[69]:
```
buyerGrouper.agg([np.mean,ptiles]).loc['ACTIVE','LTD_SALES']
```

[69]:
```
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.

3 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

[70]:
```
# Write your python code that meets the above requirements in this cell
# Question 1: Get the trDFgrouped data starting from the May heavy day counts␣
 ↪to the August heavy counts
trDFgrouped[(5,'heavy'):(8,'heavy')]
```

[70]:
```
                  transactions
monum vol
5     heavy           2172
      light           2076
6     heavy           2878
      light           1495
7     heavy           4440
      light            564
8     heavy           1682
```

[71]:
```
# Question 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
xyzGrouper = xyzcustnew.groupby(['BUYER_STATUS','heavyCat', 'ZIP'])
xyzGrouper.agg({'YTD_SALES_2009': [np.sum],'LTD_SALES':[np.sum]})
```

[71]:
```
                              YTD_SALES_2009   LTD_SALES
                                         sum         sum
BUYER_STATUS heavyCat ZIP
ACTIVE       regular  60056          68913.0    332196.0
                      60060          68520.0    339567.0
                      60061          68328.0    400569.0
                      60062         141237.0    762387.0
                      60064           2169.0      9129.0
                      60065           1002.0      2784.0
                      60067         156429.0    922680.0
                      60068         140133.0    802815.0
                      60069          43623.0    280686.0
                      60070          24051.0    134265.0
                      60071           4311.0     20112.0
                      60072           2037.0     14583.0
                      60073          29877.0    143901.0
                      60074          72999.0    349026.0
                      60076          53040.0    252438.0
                      60077          39546.0    183588.0
                      60078           1878.0      7410.0
                      60081          16446.0     76662.0
                      60083          14445.0     81954.0
                      60084          39834.0    243837.0
                      60085          18714.0     88857.0
                      60087          13749.0     59997.0
                      60088           1053.0      2538.0
                      60089         100038.0    481086.0
                      60090          32934.0    153108.0
                      60091         178533.0   1127982.0
                      60093         169671.0   1449606.0
                      60094            357.0       543.0
                      60096           5544.0     34929.0
```

```
                        60097        5805.0    29565.0
...                                     ...        ...
LAPSED        regular   60064           0.0     3537.0
                        60065           0.0     7359.0
                        60067           0.0   682167.0
                        60068           0.0   571056.0
                        60069           0.0   134685.0
                        60070           0.0    75333.0
                        60071           0.0    11232.0
                        60072           0.0     2463.0
                        60073           0.0   100932.0
                        60074           0.0   245877.0
                        60076           0.0   207912.0
                        60077           0.0   135801.0
                        60078           0.0     4173.0
                        60079           0.0     2928.0
                        60081           0.0    50397.0
                        60082           0.0      225.0
                        60083           0.0    71463.0
                        60084           0.0   157020.0
                        60085           0.0    60144.0
                        60087           0.0    45030.0
                        60088           0.0     3354.0
                        60089           0.0   407976.0
                        60090           0.0   137544.0
                        60091           0.0   820053.0
                        60093           0.0   955428.0
                        60095           0.0      300.0
                        60096           0.0    17559.0
                        60097           0.0    30564.0
                        60098           0.0   149418.0
                        60192           0.0     4548.0

[132 rows x 2 columns]
```