Lesson 10 Advanced Pandas Data Frames 3 Last update: Mar 14, 2021

Categorizing data into bins

In the last lesson we made a data frame of cars having columns make, model, doors, price and mpg. We will now group them together by mpg categories.

To use a pandas data frame we must first import pandas into your python program.

```
import pandas as pd
```

Next we load a data frame from the following csv file 'cars3.csv' from previous lesson.

cars3.csv

```
make,model,doors,price,mpg
Ford,Mustang,2,12000,23
GM,Spitfire,2,34000,56
Toyota,Yarris,4,26000,45
Nissan,Sentra,2,18000,78
BMW,Rover,2,24000,56
```

We use the **read_csv** function to read the 'cars.csv' file into a data frame.

```
df = pd.read_csv('cars3.csv')
```

We then print out the data frame using the python **print** statement and the data frame name df.

```
print(df)
```

```
     make     model doors  price   mpg
0    Ford   Mustang     2  12000    23
1      GM  Spitfire     2  34000    56
2  Toyota    Yarris     4  26000    45
3  Nissan    Sentra     2  18000    78
4     BMW     Rover     2  24000    56
```

We want to group cars together in the following mpg categories:

0 to 30 mpg
30 to 60 mpg
60 to 90 mpg

```
0    23
1    56
2    45
3    78
4    56
5    50
Name: mpg, dtype: int64
```

We will use the **cut** function to group cars together in the mpg categories:

```
# group the cars into categories
cats = pd.cut(df['mpg'],bins= [0,30,60,90])
print(cats)   # cats = categories
```

Our categories as a series are as follows:

```
Name: mpg, dtype: int64
0     (0, 30]
1    (30, 60]
2    (30, 60]
3    (60, 90]
4    (30, 60]
5    (30, 60]
Name: mpg, dtype: category

Categories (3, interval[int64]): [(0, 30] < (30, 60] < (60, 90]]
```

We then add the cat's (categories) column to our data frame

df['cats'] = cats
print(df)

```
     make     model doors  price  mpg      cats
0    Ford   Mustang     2  12000   23   (0, 30]
1      GM  Spitfire     2  34000   56  (30, 60]
2  Toyota    Yarris     4  26000   45  (30, 60]
3  Nissan    Sentra     2  18000   78  (60, 90]
4     BMW     Rover     2  24000   56  (30, 60]
```

We now count and sort the categories using the **value_counts** function and the **sort index** function

catcounts = df['cats'].value_counts().sort_index()
print(catcounts)

```
(0, 30]     1
(30, 60]    3
(60, 90]    1
Name: cats, dtype: int64
```

Our categories are: Good, Average and Best

Category (mpg)	**Rating**
0 – 30	Good
30-60	Average
60-90	Best

We add a new column to our data frame giving each category a **rating** label

df['rating']=pd.cut(df['mpg'],bins = [0,30,60,90],labels=['Good','Average','Best'])
print(df)

```
     make     model doors  price  mpg      cats   rating
0    Ford   Mustang     2  12000   23   (0, 30]     Good
1      GM  Spitfire     2  34000   56  (30, 60]  Average
2  Toyota    Yarris     4  26000   45  (30, 60]  Average
3  Nissan    Sentra     2  18000   78  (60, 90]     Best
4     BMW     Rover     2  24000   56  (30, 60]  Average
```

We want to group by rating, so we call the **groupby** function to return a group object that has a list of **data frames** for each category.

g = df.groupby('rating')

we print out each group as a **data frame** from the group list

for x in list(g.groups):

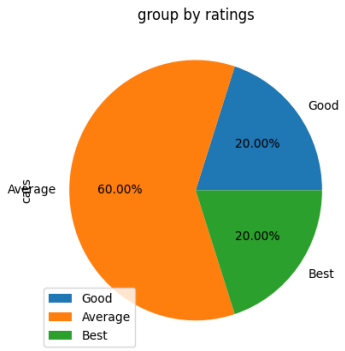
print(g.get_group(x))

```
   make    model doors  price  mpg     cats rating
0  Ford  Mustang     2  12000   23  (0, 30]   Good

     make     model doors  price  mpg      cats   rating
1      GM  Spitfire     2  34000   56  (30, 60]  Average
2  Toyota    Yarris     4  26000   45  (30, 60]  Average
4     BMW     Rover     2  24000   56  (30, 60]  Average

     make   model doors  price  mpg      cats rating
3  Nissan  Sentra     2  18000   78  (60, 90]   Best
```

Note: Each group is a separate data frame.

Advanced Pandas3 Homework Question 1

Calculate the percentage of each category, and plot a pie chart plotting the percentages,title and labels as legends. We used pie plots in previous lessons. You should get something like this:



USING PIVOT TABLES

A Pivot Table is used to summaries, sort, reorganize, group, count, total or average data stored in a table. It allows us to transform columns into rows and rows into columns. It allows grouping by any field (column), and using mathematical calculations on them. (aggregate functions)

We use the pandas **pivot_table** function to create pivot tables

pandas.pivot_table
(data, values=None, index=None, columns=None, aggfunc='mean',) → 'DataFrame'

parameter	Description
Data	data frame
Values	column to use the aggregate function on
Index	the feature columns that allows you to group your data. The index feature will appear as an index in the resultant table
column	Columns that are to be grouped
aggfunc	an aggregate function or functions like '**mean'** or '**max**' that the ***pivot_table*** applies to your grouped data

The **aggregate** function can be one of the **numpy** aggregate and statistical function.

The pivot_table returns a reshaped DataFrame based on column values.

We first do a simple pivot table to count the cars in each group using the **count** function. The **count** function is assigned to the **aggfunc** parameter to calculate the count for each category Good, Average and Best.

```
# count the cars in each group
pivot = df.pivot_table(values='mpg',index='rating',aggfunc='count')
print(pivot)
```

```
          mpg
rating
Good        1
Average     3
Best        1
```

We can also calculate the mean mpg of each group using the numpy **mean** function. We assign the numpy **mean** function to the **aggfunc** parameter**.**

import numpy as np

pivot=df.pivot_table(values='mpg',index='rating',aggfunc=np.mean)
print(pivot)

```
            mpg
rating
Good      23.00
Average   52.33
Best      78.00
```

We now add the price column values as price column's which are the values from the price column of the df dataframe.

import numpy as np

pivot=df.pivot_table(values='mpg',index='rating',columns=['price'], aggfunc=np.mean)
print(pivot)

```
price    12000  18000  24000  26000  34000
rating
Good      23.0    NaN    NaN    NaN    NaN
Average    NaN    NaN   56.0   45.0   56.0
Best       NaN   78.0    NaN    NaN    NaN
price    12000  18000 24000  26000    34000
```

The NaN's means there are no values available for the category and the price column.

We can add more columns like make and model

```
import numpy as np

pivot=df.pivot_table(values='mpg',index='rating',
   columns=['price','make','model'],aggfunc=np.mean)
print(pivot)
```

```
model   Mustang Sentra Rover Yarris Spitfire
rating
Good       23.0    NaN   NaN    NaN      NaN
Average     NaN    NaN  56.0   45.0     56.0
Best        NaN   78.0   NaN    NaN      NaN
```

The Nan means there are no values for the category and the price, model or make column.

Here are a list of numpy aggregate functions that you could use:

Function	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.
np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min()	Return the minimum of an array or minimum along an axis.
np.max()	Return the maximum of an array or minimum along an axis
np.argmin(),	Returns the indices of the minimum / maximum values along an axis
np.argmax()	Returns the indices of the minimum / maximum values along an axis
np.all()	Test whether all array elements along a given axis evaluate to True.
np.any()	Test whether any array element along a given axis evaluates to True.

Advanced Pandas3 Homework Question 2

Try another aggregate function like std or mode. Try 2 different aggregate function enclosed in a list like: aggfunc=['max','sum']. You should get something like this:

```
         max   sum
         mpg   mpg
rating
Good      23    23
Average   56   207
Best      78    78
```

Working with Dates

Dates can be stored as **strings** objects or python datetime objects located in the python **datetime** module. Storing data as **datetime** objects may be preferred since you may want to compare date values. We first make a csv file with some dates and an additional column of related values. The date format is YYYY-MM-DD.

dates.csv

```
dates,values
2020-06-01,23
2020-06-02,45
2020-06-03,23
2020-06-04,45
2020-06-05,34
2020-06-06,23
2020-06-07,56
2020-06-08,25
2020-06-09,65
2020-06-10,34
```

We then read in the csv file using the pandas **read_csv** function

df3 = pd.read_csv('dates.csv', parse_dates=True, index_col='dates')

We have used the parameter **parse_dates=True** to read in the date strings, and convert them to a **datetime** object. The **index_col** parameter is used to set the index column to 'dates'

print(df3)

```
            Values
dates
2020-06-01      23
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
2020-06-09      65
2020-06-10      34
```

We then print out the data frame **info** , to confirm we have a DateTime index.

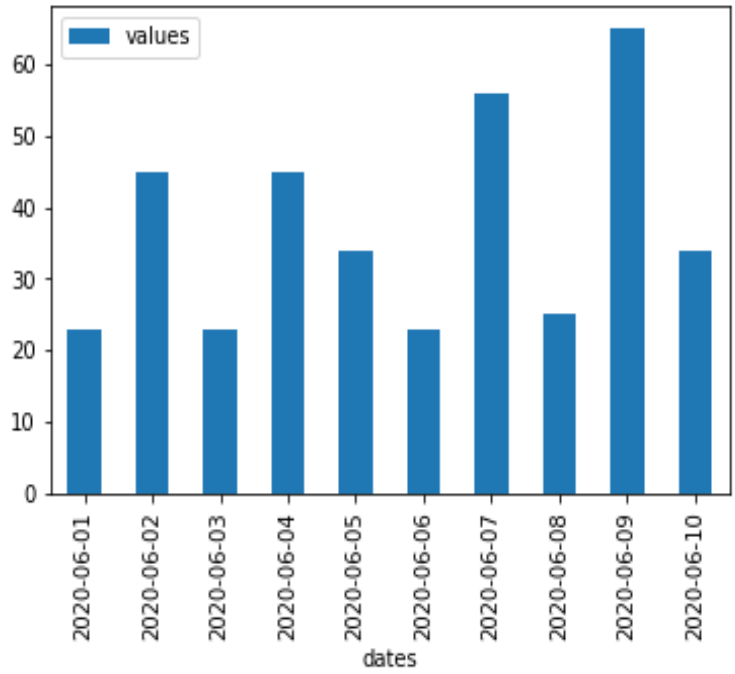
df3.info()

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 10 entries, 2020-06-01 to 2020-06-10
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   values  10 non-null     int64
dtypes: int64(1)
memory usage: 160.0 bytes
```

Plotting date-time

We use pandas bar chart with formatting to plot the date time

```
import matplotlib.pyplot as plt
ax = df3.plot.bar()
ax.set_xticklabels(df3.index.strftime('%Y-%m-%d'))
plt.show()
```



Alternatively you can make a dictionary of dates

```
# dictionary of dates
mydata = {'dates':  ['2020-06-01','2020-06-02','2020-06-03','2020-06-04',
                     '2020-06-05','2020-06-06','2020-06-07','2020-06-08',
                     '2020-06-09','2020-06-10'],
          'values': [23,45,23,45,34,23,56,25,65,34]}

# make data frame
df3 = pd.DataFrame(mydata, columns = ['dates','values'])
```

we then print out the data frame

```
print(df3)
```

```
        dates  values
0  2020-06-01      23
1  2020-06-02      45
2  2020-06-03      23
3  2020-06-04      45
4  2020-06-05      34
5  2020-06-06      23
6  2020-06-07      56
7  2020-06-08      25
8  2020-06-09      65
9  2020-06-10      34
```

Next we print out the data frame info, notice the date columns are **string** objects not **datetime** objects.

df3.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   dates   10 non-null     object
 1   values  10 non-null     int64
dtypes: int64(1), object(1)
memory usage: 288.0+ bytes
```

The dates are stored as strings , you need to convert the date strings to **datetime** objects using the pandas **to_datetime function**

df3['dates'] = pd.to_datetime(df3['dates'], format='%Y-%m-%d')
print(df3)

```
       dates  values
0 2020-06-01      23
1 2020-06-02      45
2 2020-06-03      23
3 2020-06-04      45
4 2020-06-05      34
5 2020-06-06      23
6 2020-06-07      56
7 2020-06-08      25
8 2020-06-09      65
9 2020-06-10      34
```

When we print out the data frame info, we now have datetime object for our dates.

df3.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   dates   10 non-null     datetime64[ns]
 1   values  10 non-null     int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 288.0 bytes
```

Lastly we set the date column as our index column using **inplace = true**

df3.set_index('dates',inplace=True)
print(df3)

```
            Values
dates
2020-06-01      23
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
2020-06-09      65
2020-06-10      34
```

df3.info()

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 10 entries, 2020-06-01 to 2020-06-10
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   values  10 non-null     int64
dtypes: int64(1)
memory usage: 160.0 bytes
```

Accessing date data

To print out the row for datetime **'2020-06-04'**

print(df3.loc['2020-06-04'])

```
values    45

Name: 2020-06-04
00:00:00, dtype: int64
```

To print out the value for datetime **'2020-06-04'**

print(df3.loc['2020-06-04','values'])

```
45
```

Making a range of dates

Using the **date_range** function we specify the start date the end date and the time range span frequency **freq='D'**

```
mydates = pd.date_range('2020-06-01', '2020-06-10', freq='D')

print(mydates)
```

```
DatetimeIndex(['2020-06-01', '2020-06-02', '2020-06-03', '2020-06-04',
               '2020-06-05', '2020-06-06', '2020-06-07', '2020-06-08',
               '2020-06-09', '2020-06-10'],
              dtype='datetime64[ns]', freq='D')
```

We now set the data frame with our date range as well as the list of values, and set the index to out date column using **inplace = True.**

```
df3 = pd.DataFrame({'dates': mydates, 'values': [23,45,23,45,34,23,56,25,65,34]})
df3.set_index('dates',inplace=True)
print(df3)
```

```
            values
dates
2020-06-01      23
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
2020-06-09      65
2020-06-10      34
```

Here are the available time span frequencies:

Code	Description
D	Calendar day
W	Weekly
M	Month end
Q	Quarter end
A	Year end
H	Hours
T	Minutes
S	Seconds
L	Milliseconds
U	Microseconds
N	Nanoseconds

Selecting dates using the dates index column

We just use slicing and use a start date and a end date

print(df3['2020-06-02' : '2020-06-08'])

```
            values
dates
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
```

Select dates not using a index

We first restore our dates column, to get rid of the indexes.

df3.reset_index(inplace=True)
print(df3)

```
        dates  values
0 2020-06-01      23
1 2020-06-02      45
2 2020-06-03      23
3 2020-06-04      45
4 2020-06-05      34
5 2020-06-06      23
6 2020-06-07      56
7 2020-06-08      25
8 2020-06-09      65
9 2020-06-10      34
```

We then select all dates from '2020-06-02' to '2020-06-08'

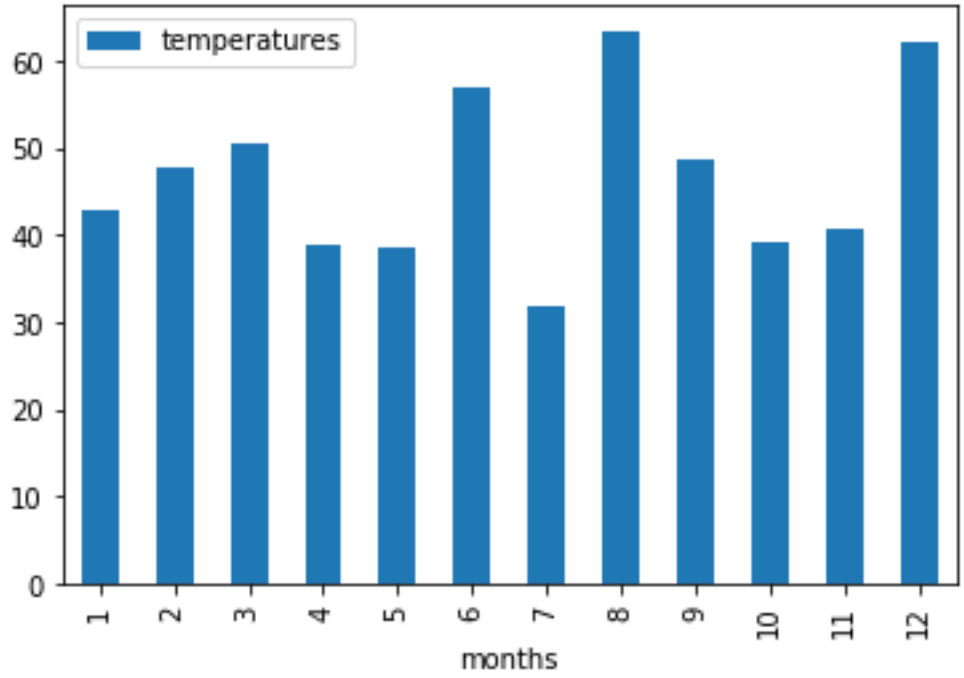
dates = df3[(df3['dates'] > '2020-06-02') & (df3['dates'] <= '2020-06-08')]
print(dates)

```
        dates  values
2 2020-06-03      23
3 2020-06-04      45
4 2020-06-05      34
5 2020-06-06      23
6 2020-06-07      56
7 2020-06-08      25
```

Advanced Pandas3 Homework Question 3

Make a DataFrame with 5 to 10 years of random value temperatures. Group the dates by month, and plot the average of the month temperatures .

You should get something like this:



We restore the index column for next section

df3.set_index('dates',inplace=True)

or you can re-assign like this:

df3 = pd.DataFrame({'dates': mydates, 'values': [23,45,23,45,34,23,56,25,65,34]})
df3.set_index('dates',inplace=True)
print(df3)

```
            values
dates
2020-06-01      23
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
2020-06-09      65
2020-06-10      34
```

USING APPLY, MAP, APPLYMAP AND TRANSFORM

The **apply**, **map**, **applymap** and **transform** pandas functions use an additional function to be applied to a cell in a pandas data frame or a cell in a series. Each apply, map, applymap and transform is used in different situations as specified in the following chart.

Function	Description
Apply	can be applied both to series and dataframes , return a new Series or DataFrame
Map	only works on a pandas series where type of operation to be applied depends on argument passed as a function, dictionary or a list. return a new Series
applymap	only works on a pandas dataframe where the function is applied on every element individually, returns a new DataFrame
transform	apply a function to a whole data frame or specified column, return a new Series or DataFrame

apply

The apply method can be applied both to series and dataframes

df.apply(func)
df['column'] = df['column'].apply(func)

We first make a function that just increments a value:

```
def inc(x):
    x = x + 1
    return x
```

We then use the apply method on our datetime data frame df3 from the above previous section. The **inc** function is to increment every value in the values column of the df3 data frame.

```
            Values
dates
2020-06-01      23
2020-06-02      45
2020-06-03      23
2020-06-04      45
2020-06-05      34
2020-06-06      23
2020-06-07      56
2020-06-08      25
2020-06-09      65
2020-06-10      34
```

df3 = df3.apply(inc)
print(df3)

```
            Values
dates
2020-06-01      24
2020-06-02      46
2020-06-03      24
2020-06-04      46
2020-06-05      35
2020-06-06      24
2020-06-07      57
2020-06-08      26
2020-06-09      66
2020-06-10      35
```

using a lambda function

A **lambda** is just a function with no name that you can use as an inline function with inside your other code.

df3=df3.apply(lambda x: x + 1)

lambda x: x + 1 means

```
def inc(x):
    x = x + 1
    return x
```

where **x** is the input parameter and **x+1** increments x

Using the lambda we increment's every value in a data frame column as follows:

print(df3)

```
            Values
dates
2020-06-01      25
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

We can also use apply with columns

df3['column_name'].apply(func)

```
df3['values']=df3['values'].apply(inc)
print(df3)
```

```
            Values
dates
2020-06-01      26
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

using a lamda function on columns

df['column_name'].apply (lambda x: x + 1)

```
df3['values']=df3['values'].apply(lambda x: x + 1)
print(df3)
```

```
            Values
dates
2020-06-01      25
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

using map

The **map** method can be only be applied to a **Series**

df['column'] = df['column'].map(func)

We first make a function that just increments a value:

```
def  inc(x):
    x = x + 1
    return x
```

We then use the map on our datetime data frame df3 from the above previous section. The **inc** function is to increment every value in the values column of the df3 data frame.

df3['column_name'].apply(func)

```
df3['values']=df3['values'].map (inc)
print(df3)
```

```
            Values
dates
2020-06-01      26
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

using a lamda function on columns

df['column_name'].map(lambda x: x + 1)

df3['values']=df3['values'].map(lambda x: x + 1)
print(df3)

```
            Values
dates
2020-06-01      25
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

applymap

The **applymap** can be applied only to DataFrames

df.applymap(func)

We first make a function that just increments a value:

```
def  inc(x):
    x = x + 1
    return x
```

We then use the **applymap** function on our datetime data frame df3 from the above previous section. The **inc** function is to increment every value in the values column of the df3 data frame.

```
df3=df3.applymap(inc)
print(df3)
```

```
            Values
dates
2020-06-01      24
2020-06-02      46
2020-06-03      24
2020-06-04      46
2020-06-05      35
2020-06-06      24
2020-06-07      57
2020-06-08      26
2020-06-09      66
2020-06-10      35
```

using a lambda function

A lamda is just a function with no name that you can use as a inline function with all your other code.

```
df3=df3.applymap(lambda x: x + 1)
print(df3)
```

Using the lambda we increment's every value in a data frame column as follows:

```
            Values
dates
2020-06-01      25
2020-06-02      47
2020-06-03      25
2020-06-04      47
2020-06-05      36
2020-06-06      25
2020-06-07      58
2020-06-08      27
2020-06-09      67
2020-06-10      36
```

transform

Transform allows us to apply a function to a whole data frame or specified column.

df.transform(func)

We first make a function that just increments a value:

```
def  inc(x):

    x = x + 1
    return x
```

We then use a transform on our **inc** function to increment every value in the previous data frame

```
df3 = df3.transform(inc)
print(df3)
```

	Values
dates	
2020-06-01	24
2020-06-02	46
2020-06-03	24
2020-06-04	46
2020-06-05	35
2020-06-06	24
2020-06-07	57
2020-06-08	26
2020-06-09	66
2020-06-10	35

using a lambda function

A lamda is just a function with no name that you can use as a inline function with all your other code.

```
df3=df3.transform(lambda x: x + 1)
print(df3)
```

increment's every value in a data frame column

	Values
dates	
2020-06-01	24
2020-06-02	46
2020-06-03	24
2020-06-04	46
2020-06-05	35
2020-06-06	24
2020-06-07	57
2020-06-08	26
2020-06-09	66
2020-06-10	35

We can also use transforms with columns

df3['column_name'].transform(func)

```
df3['values']=df3['values'].transform(inc)
print(df3['values'])
```

using a lambda function on columns

df['column_name'].transform(lambda x: x + 1)

```
df3['values']=df3['values'].transform(lambda x: x + 1)
print(df3['values'])
```

	Values
dates	
2020-06-01	24
2020-06-02	46
2020-06-03	24
2020-06-04	46
2020-06-05	35
2020-06-06	24
2020-06-07	57
2020-06-08	26
2020-06-09	66
2020-06-10	35

Advanced Pandas3 Homework Question 4

Use all on a column of a dataframe **apply, map, applymap** and **transform.** Use a **lambda** function for convenience. Hint: use chaining. You should get something like this:

```
   values
0      27
1      49
2      27
3      49
4      38
5      27
6      60
7      29
8      69
9      38
```

Advanced Pandas3 Homework Question 5

Make a monthly time series spanning about 1 year using the pandas **date range** function. Make some columns to represent stocks prices , or home sales prices.

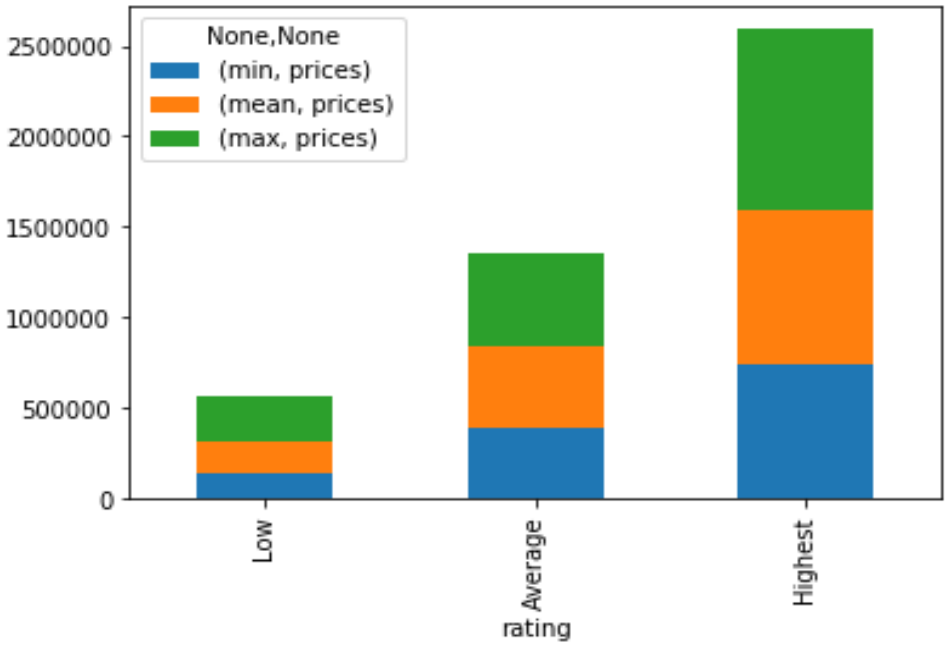
Use the **cut** function to group the prices into low prices, average price and highest prices. Make a results column labeled low, average and highest. Use a **pivot table** to calculate the count, min, mean, max price and of each group. Plot the results on a bar chart or a stacked bar chart. Also make a pie chart to show the percentages of each group..

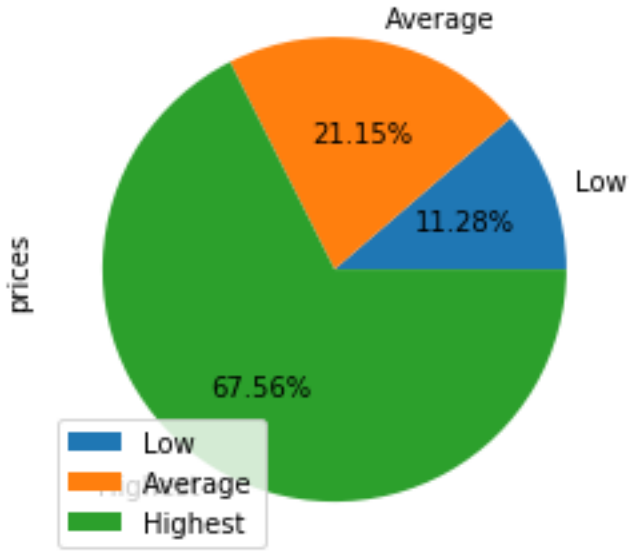
Call your homework file advpandas3_homework.py

You should get something like this:

```
            prices
months
2010-01-31  392367
2010-02-28  511114
2010-03-31  443000
2010-04-30  843374
2010-05-31  997491
```

```
2010-06-30  139415
2010-07-31  901244
2010-08-31  176224
2010-09-30  735788
2010-10-31  246655
2010-11-30  155809
2010-12-31  822645

months
2010-01-31  392367   (300000, 700000]  Average
2010-02-28  511114   (300000, 700000]  Average
2010-03-31  443000   (300000, 700000]  Average
2010-04-30  843374  (700000, 1000000]  Highest
2010-05-31  997491  (700000, 1000000]  Highest
2010-06-30  139415   (100000, 300000]      Low
2010-07-31  901244  (700000, 1000000]  Highest
2010-08-31  176224   (100000, 300000]      Low
2010-09-30  735788  (700000, 1000000]  Highest
2010-10-31  246655   (100000, 300000]      Low
2010-11-30  155809   (100000, 300000]      Low
2010-12-31  822645  (700000, 1000000]  Highest
months
2010-06-30  139415  (100000, 300000]    Low
2010-08-31  176224  (100000, 300000]    Low
2010-10-31  246655  (100000, 300000]    Low
2010-11-30  155809  (100000, 300000]    Low
            prices              cats   rating
months
2010-01-31  392367  (300000, 700000]  Average
2010-02-28  511114  (300000, 700000]  Average
2010-03-31  443000  (300000, 700000]  Average
            prices              cats   rating
months
2010-04-30  843374  (700000, 1000000]  Highest
2010-05-31  997491  (700000, 1000000]  Highest
2010-07-31  901244  (700000, 1000000]  Highest
2010-09-30  735788  (700000, 1000000]  Highest
2010-12-31  822645  (700000, 1000000]  Highest


            min       mean     max
         prices     prices  prices
rating
Low      139415  179525.75  246655
Average  392367  448827.00  511114
Highest  735788  860108.40  997491
```





END