Data Grouping and Aggregation In Pandas

Data Grouping: 'groupby'

The purpose of pivot tables for reporting and data visualization is to analyze the data based on certain grouping mechanics mthe pandas 'groupby' method will serve this purpose with high demand.

In [1]:
```
# pandas provides a flexible groupby interface, enabling you to slice, dice, and summarize datasets in a natural way.
# The pandas 'groupby' method will operate in three different stages named as 'split' 'apply' and 'combine'
```

In [2]:
```
import pandas as pd
import numpy as np
```

In [3]:
```
# The docstring is as follows:
# Docstring :Group series using mapper (dict or key function, apply given function to group, return result as series)
 or by
# a series of columns.
pd.DataFrame.groupby?
```

In [4]:
```
# Let's consider a simple 'marks' example
marks = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a', 'a'],
                      'key2' : ['one', 'two', 'one', 'two', 'one', 'one'],
                     'data1' : np.arange(10, 16),
                     'data2' : np.arange(16, 22)})
marks
```

Out[4]:

	key1	key2	data1	data2
0	*a*	*one*	*10*	*16*
1	*a*	*two*	*11*	*17*
2	*b*	*one*	*12*	*18*
3	*b*	*two*	*13*	*19*
4	*a*	*one*	*14*	*20*
5	*a*	*one*	*15*	*21*

In [5]:
```
# now take out the 'data1' column data out
marks['data1']
```

Out[5]:
```
0    10
1    11
2    12
3    13
4    14
5    15
Name: data1, dtype: int32
```

In [6]:
```
# Let's group this data besed on the key values of column 'key1'
# It results a grouped object which is stored at specific location and no operation it carries until we apply on it
grouped = marks['data1'].groupby(by=marks['key1'])
grouped
```

Out[6]:
```
<pandas.core.groupby.groupby.SeriesGroupBy object at 0x068D5F70>
```

In [7]:
```
# Now apply the aggregate function 'mean' to find out the mean value of the grouped data
# the data will be aggregated according to the group key here based on 'key1'
grouped.mean()
```

Out[7]:
```
key1
a    12.5
b    12.5
Name: data1, dtype: float64
```

In [8]:
```
# The result index has the name 'key1' because the DataFrame column marks['key1'] did the actual grouping work to yiel
d mean
```

In [9]:
```
# check the actual value for comfirmation, because if we apply it for the large data set we've to waste our lot time t
o check
# the result manually
print('a mean:', (10 + 11 + 14 + 15) / 4)
print('b mean:', (12 + 13)/2)
```

```
a mean: 12.5
b mean: 12.5
```

In [10]:
```
# check for some other function
grouped.sum()
```

Out[10]:
```
key1
a    50
b    25
Name: data1, dtype: int32
```

In [11]:
```
# Suppose if we pass two or more keys to groupby, we will get an hierarchically indexed data along with the result of
 the
# aggregate fucntion: it is actually looking for the unique pair of observed keys
group_tk = marks['data1'].groupby(by=[marks['key1'], marks['key2']])
group_tk
```

Out[11]:
```
<pandas.core.groupby.groupby.SeriesGroupBy object at 0x06E790B0>
```

In [12]:
```
print(marks)
print()
print('result:', group_tk.mean())
```

```
  key1 key2  data1  data2
0    a  one     10     16
1    a  two     11     17
2    b  one     12     18
3    b  two     13     19
4    a  one     14     20
5    a  one     15     21

result: key1  key2
a     one     13
      two     11
b     one     12
      two     13
Name: data1, dtype: int32
```

In [13]:
```
print((10 + 14 +15)/3)
```

```
13.0
```

In [14]:
```
# let's unstack the result to see what we are expecting actually
group_tk.mean().unstack()
```

Out[14]:

key2	one	two
key1		
a	13	11
b	12	13

In [15]:
```
# The group keys need not be a Series always, they can be a array of equal length as that of the DF Series
feedb = np.array(['good', 'avg', 'good', 'avg', 'good', 'avg'])
actual = np.array(['good', 'med', 'good', 'med', 'good', 'med'])
mean1 = marks['data1'].groupby(by=[feedb, actual]).mean()
mean1
```

Out[15]:
```
avg   med     13
good  good    12
Name: data1, dtype: int32
```

In [16]:
```
# To work with the whole DF we can pass a sinlge or a list of column names to the groupby which is appying on the whol
e DF
mean_df = marks.groupby(by=['key1']).mean()
mean_df
```

Out[16]:

key1	data1	data2
a	12.5	18.5
b	12.5	18.5

In [17]:
```
# Notice that the column 'key2' is missing, this is because the column 'key2' is a non-numeric Series, so aggregarion
 cannot
# be applied to that, hence groupby will exclude such column if found as a 'nuisance column'
# In other words by default aggregation is applied to all of the numeric columns, but we can do filtering operations t
o operate
# on different types of columns.
```

In [18]:
```
# To check the group size of the DataFrame we can use size aggeration on groupby method
print(marks)
print()
marks.groupby(by=['key1', 'key2']).size()
```

```
  key1 key2  data1  data2
0    a  one     10     16
1    a  two     11     17
2    b  one     12     18
3    b  two     13     19
4    a  one     14     20
5    a  one     15     21
```

Out[18]:
```
key1  key2
a     one     3
      two     1
b     one     1
      two     1
dtype: int64
```

How To Iterate Over Groups?

In [19]:
```
# The GroupBy object supports direct iteration over the groups,returning each group as a Series or DataFrame
marks
```

Out[19]:

	key1	key2	data1	data2
0	*a*	*one*	*10*	*16*
1	*a*	*two*	*11*	*17*
2	*b*	*one*	*12*	*18*
3	*b*	*two*	*13*	*19*
4	*a*	*one*	*14*	*20*
5	*a*	*one*	*15*	*21*

In [20]:
```
marks.groupby(by='key1')
```

Out[20]:
```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x06E79250>
```

In [21]:
```
# Let's iterate over each group and check the type of these objects
for key_name, group_name in marks.groupby(by='key1'):
    print(key_name)
    print(group_name)
    print(type(key_name))
    print(type(group_name))
```

```
a
  key1 key2  data1  data2
0    a  one     10     16
1    a  two     11     17
4    a  one     14     20
5    a  one     15     21
<class 'str'>
<class 'pandas.core.frame.DataFrame'>
b
  key1 key2  data1  data2
2    b  one     12     18
3    b  two     13     19
<class 'str'>
<class 'pandas.core.frame.DataFrame'>
```

In [22]:
```
# we can pass multiple keys for group iteration in the form of tuple of keys
for (k1_name, k2_name), group_name in marks.groupby(by=['key1', 'key2']):
    print(k1_name, k2_name)
    print(group_name)
    #print(type(k1_name), type(k2_name))
    #print(type(group_name))
```

```
a one
  key1 key2  data1  data2
0    a  one     10     16
4    a  one     14     20
5    a  one     15     21
a two
  key1 key2  data1  data2
1    a  two     11     17
b one
  key1 key2  data1  data2
2    b  one     12     18
b two
  key1 key2  data1  data2
3    b  two     13     19
```

In [23]:
```
for (k1_name, k2_name), group_name in marks.groupby(by=['data1', 'data2']):
    print(k1_name, k2_name)
    print(group_name)
```

```
10 16
  key1 key2  data1  data2
0    a  one     10     16
11 17
  key1 key2  data1  data2
1    a  two     11     17
12 18
  key1 key2  data1  data2
2    b  one     12     18
13 19
  key1 key2  data1  data2
3    b  two     13     19
14 20
  key1 key2  data1  data2
4    a  one     14     20
15 21
  key1 key2  data1  data2
5    a  one     15     21
```

In [24]:
```
# The default axis in groupby iteration is axis=0, but we can also iterate over groupby through columns
# Let's create columns for iteration
marks.dtypes
```

Out[24]:
```
key1     object
key2     object
data1     int32
data2     int32
dtype: object
```

In [25]:
```
# you will the data iterated over group in column wise, eralier it was on rows, now it is on columns
grouped = marks.groupby(marks.dtypes, axis=1)
for datatype, group in grouped:
    print(datatype)
    print(group)
```

```
int32
   data1  data2
0     10     16
1     11     17
2     12     18
3     13     19
4     14     20
5     15     21
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
5    a  one
```

Column Selection For Aggregation via 'groupby'

In [26]:
```
# we have seen that in DataFrame can select any column by column indexing
# Ex: marks['key1'], marks['key2'] or marks['data1']
## to select multiple column names pass a list of column names for data indexing
print(marks['data1']); print(marks[['data1', 'data2']])
```

```
0    10
1    11
2    12
3    13
4    14
5    15
Name: data1, dtype: int32
   data1  data2
0     10     16
1     11     17
2     12     18
3     13     19
4     14     20
5     15     21
```

In [27]:
```
# Let's group the above data based on key1 and key2
sk_g = marks['data1'].groupby(by=marks['key1'])
dk_g = marks[['data1', 'data2']].groupby(by=marks['key2'])
sk_g
dk_g
```

Out[27]:
```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x06E987F0>
```

In [28]:
```
print(sk_g.sum())
print(dk_g.sum())
```

```
key1
a    50
b    25
Name: data1, dtype: int32
      data1  data2
key2
one      51     75
two      24     36
```

In [29]:
```
# in order to avoid repeated use of DataFrame name while grouping over the columns we can use alternative method
# that is indexing type groupby method

# single column indexing type
sk_g = marks.groupby(by='key1')['data1']
# multiple column indexing type by passing a list of column names to the column indexing
dk_g = marks.groupby(by='key2')[['data1', 'data2']]
print(sk_g.sum())
print(dk_g.sum())
```

```
key1
a    50
b    25
Name: data1, dtype: int32
      data1  data2
key2
one      51     75
two      24     36
```

How To Group With Dictionaries and Series?

In [30]:
```
# We can use dictionaries and Series to group the DataFrame objects instead of row and column indexing
# but the length of the dictionary or Series elments must match the column length of the DF
# Ex:
rmlist = pd.DataFrame(np.random.randn(4, 5),
                      columns=['a', 'b', 'c', 'd', 'e'],
                      index=['one', 'two', 'three', 'four'])
rmlist
```

Out[30]:

	a	b	c	d	e
one	*-0.238386*	*-1.693858*	*-0.800670*	*-1.170462*	*-1.182522*
two	*0.854548*	*0.723782*	*-0.584910*	*-2.127303*	*1.388898*
three	*1.117445*	*-1.324266*	*-1.552888*	*0.227395*	*0.352264*
four	*0.778488*	*-0.999548*	*-1.161931*	*-1.966660*	*1.684085*

In [31]:
```
# Let's use a dictionary of elements correspods to the columns of the DF
dic_map = {'a': 'red', 'b': 'red', 'c': 'blue', 'd': 'blue', 'e': 'red', 'f' : 'orange'}
dic_map
```

Out[31]:
```
{'a': 'red', 'b': 'red', 'c': 'blue', 'd': 'blue', 'e': 'red', 'f': 'orange'}
```

In [32]:
```
# Now let's pass this dictionary to the groupby method and notice that the grouping is along column hence, it care mus
t be
# taken while grouping or iterating over columns. So use axis=1 for columns selection
g_column = rmlist.groupby(by=dic_map, axis=1)
g_column
```

Out[32]:
```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x06E98670>
```

In [33]:
```
# Notice that the dictionary has the extra key-value pair and is excluded in grouping because groupby looks only for t
he matching
# objects and the non-matching objects are ingnored by default.
```

In [34]:
```
g_column.sum()
```

Out[34]:

	blue	red
one	*-1.971132*	*-3.114765*
two	*-2.712213*	*2.967228*
three	*-1.325493*	*0.145443*
four	*-3.128591*	*1.463025*

In [35]:
```
# we can also pass Series by first constructing a Series having length equal to or greater than the length of DF colum
n
s_map = pd.Series(dic_map)
```

In [36]:
```
g_column = rmlist.groupby(by=s_map, axis=1)
g_column.sum()
```

Out[36]:

	blue	red
one	*-1.971132*	*-3.114765*
two	*-2.712213*	*2.967228*
three	*-1.325493*	*0.145443*
four	*-3.128591*	*1.463025*

How To Group With Functions?

In [37]:
```
# we can use Python functions instead of Dictionaries and Series. This is the more generic method to group the data.
# Any function passed as a group key will be called once per index value, with the return values being used as the gro
up names.
```

In [38]:
```
print(rmlist)
rmlist.groupby(len).sum()
```

```
              a         b         c         d         e
one   -0.238386 -1.693858 -0.800670 -1.170462 -1.182522
two    0.854548  0.723782 -0.584910 -2.127303  1.388898
three  1.117445 -1.324266 -1.552888  0.227395  0.352264
four   0.778488 -0.999548 -1.161931 -1.966660  1.684085
```

Out[38]:

	a	b	c	d	e
3	*0.616162*	*-0.970076*	*-1.385580*	*-3.297765*	*0.206376*
4	*0.778488*	*-0.999548*	*-1.161931*	*-1.966660*	*1.684085*
5	*1.117445*	*-1.324266*	*-1.552888*	*0.227395*	*0.352264*

In [39]:
```
# here the DF index contains string values. hence the length of that string values are considered to group the data
# but if we use the DF with integer index we may get an error, because integer index itself is a integer, and hence
# cannot be used to calculate the length

#marks.groupby(len).sum()             # try running this code by removing '#'

# if you run the above code you will get the following error:
# TypeError: object of type 'int' has no len()
```

In [40]:
```
# we can mix, arrays, Dictionaries and Series with function inside the groupby method, everything will get convert int
o arrays
# by default internally. Take care while passing multiple keys, that means use [] to pass multiple keys to groupby
key_list = ['one', 'one', 'one', 'two']
print(rmlist)
rmlist.groupby([len, key_list]).sum() # here the grouping takes place on the index values length first and then on key
_list
```

```
              a         b         c         d         e
one   -0.238386 -1.693858 -0.800670 -1.170462 -1.182522
two    0.854548  0.723782 -0.584910 -2.127303  1.388898
three  1.117445 -1.324266 -1.552888  0.227395  0.352264
four   0.778488 -0.999548 -1.161931 -1.966660  1.684085
```

Out[40]:

		a	b	c	d	e
3	**one**	*0.616162*	*-0.970076*	*-1.385580*	*-3.297765*	*0.206376*
4	**two**	*0.778488*	*-0.999548*	*-1.161931*	*-1.966660*	*1.684085*
5	**one**	*1.117445*	*-1.324266*	*-1.552888*	*0.227395*	*0.352264*

In [41]:
```
# for better understading let's use the array values similar to the index values
key_list = ['one', 'two', 'three', 'four']
print(rmlist)
rmlist.groupby([len, key_list]).sum() # here the grouping takes place on the index values length
```

```
              a         b         c         d         e
one   -0.238386 -1.693858 -0.800670 -1.170462 -1.182522
two    0.854548  0.723782 -0.584910 -2.127303  1.388898
three  1.117445 -1.324266 -1.552888  0.227395  0.352264
four   0.778488 -0.999548 -1.161931 -1.966660  1.684085
```

Out[41]:

		a	b	c	d	e
3	**one**	*-0.238386*	*-1.693858*	*-0.800670*	*-1.170462*	*-1.182522*
	two	*0.854548*	*0.723782*	*-0.584910*	*-2.127303*	*1.388898*
4	**four**	*0.778488*	*-0.999548*	*-1.161931*	*-1.966660*	*1.684085*
5	**three**	*1.117445*	*-1.324266*	*-1.552888*	*0.227395*	*0.352264*

In [42]:
```
# Notice that the above method indicates that the passed arrays along with the original index values behaves like mult
iindex values
# or Hierarchically indexed type in groupby method. This is one interesting observation.
```

How To Group by Index Level?

In [43]:
```
# To group the data on index level pass the level name or number to the groupby method using level keyword
```

In [44]:
```
columns = pd.MultiIndex.from_arrays([['US', 'US', 'UK', 'RS', 'RS'],
                                     [1, 3, 5, 1, 3]],
                                     names=['city', 'tenor'])

hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)
hier_df
```

Out[44]:

city	US		UK	RS	
tenor	1	3	5	1	3
0	2.034812	-1.199343	0.209081	0.337497	-0.040967
1	-0.141667	1.142703	2.063086	1.168057	1.084399
2	-1.675321	-1.041685	-2.481958	-0.686362	0.807823
3	0.573257	-0.414272	-1.255019	-0.767197	2.830240

In [45]:
```
# Let's pass the index level name to the level keyword
hier_df.groupby(level='city', axis=1).count()
```

Out[45]:

city	RS	UK	US
0	2	1	2
1	2	1	2
2	2	1	2
3	2	1	2

In [46]:
```
# The next topic of interest is Data Aggregation
```

In [ ]:
```

```

Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays.

Some common aggregation methods are

- *count*
- *sum*
- *mean*
- *median*
- *std, var*
- *min, max*
- *prod*
- *first, last*

still you can find many methods, these are just to illustrate

In [47]:
```
# We can also use our own aggregation functions
```

In [48]:
```
# Ex: Let's use a simple 20 rows book data for analysis purpose
book = pd.read_csv(r'dataset/books_discount.csv', encoding='latin')
book.head() # which displays the first 5 rows of the DF object
```

Out[48]:

	book_name	price	author	min_dis	max_dis	feedback
0	*AAA*	*100*	*Author1*	*0.2*	*0.4*	*good*
1	*AAB*	*200*	*Author2*	*0.2*	*0.4*	*average*
2	*AAC*	*300*	*Author3*	*0.2*	*0.3*	*good*
3	*AAD*	*100*	*Author4*	*0.2*	*0.4*	*good*
4	*AAE*	*200*	*Author5*	*0.2*	*0.4*	*average*

In [49]:
```
# Let's use min and max aggregation funcions to know the min and max price of the 20 books
print(book['price'].min()); print(book['price'].max())
```

```
50
700
```

In [50]:
```
# Let's check the leat value of min and max discount
print(book['min_dis'].min()); print(book['max_dis'].min())
```

```
0.05
0.1
```

In [51]:
```
# We can use our own aggregation function on grouped object using Pandas 'aggregate' or 'agg' method
```

In [52]:
```
# Let's group the book data based on book_name
grouped = book.groupby(by=['feedback', 'author'], axis=0)
```

In [53]:
```
# Let's use the max amd min functions inside the user defined function to apply for the book data using 'agg' method
def max_min(arr):
    return arr.max(), arr.min()

grouped.agg(max_min)
```

Out[53]:

		book_name	price	min_dis	max_dis
feedback	**author**				
average	**Author2**	*(AAS, AAB)*	*(500, 200)*	*(0.5, 0.1)*	*(0.55, 0.4)*
	Author5	*(AAK, AAE)*	*(500, 200)*	*(0.3, 0.2)*	*(0.4, 0.35)*
good	**Author1**	*(AAM, AAA)*	*(400, 50)*	*(0.3, 0.2)*	*(0.4, 0.35)*
	Author3	*(AAT, AAC)*	*(700, 200)*	*(0.3, 0.05)*	*(0.4, 0.1)*
	Author4	*(AAP, AAD)*	*(100, 100)*	*(0.3, 0.1)*	*(0.45, 0.3)*
	Author6	*(AAL, AAL)*	*(300, 300)*	*(0.4, 0.4)*	*(0.5, 0.5)*

In [54]:
```
# some commonly used methods also work like aggregation function, but they are actually not
# ex: describe() method
grouped.describe() # here the book_name is not considered as it is a string object
```

Out[54]:

		max_dis								min_dis					price		
		count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std
feedback	author																
average	Author2	5.0	0.460000	0.065192	0.40	0.4000	0.450	0.5000	0.55	5.0	0.280000	...	0.400	0.5	5.0	260.000000	134.164079
	Author5	2.0	0.375000	0.035355	0.35	0.3625	0.375	0.3875	0.40	2.0	0.250000	...	0.275	0.3	2.0	350.000000	212.132034
good	Author1	3.0	0.383333	0.028868	0.35	0.3750	0.400	0.4000	0.40	3.0	0.266667	...	0.300	0.3	3.0	183.333333	189.296945
	Author3	6.0	0.291667	0.102062	0.10	0.3000	0.300	0.3375	0.40	6.0	0.141667	...	0.175	0.3	6.0	400.000000	200.000000
	Author4	3.0	0.383333	0.076376	0.30	0.3500	0.400	0.4250	0.45	3.0	0.200000	...	0.250	0.3	3.0	100.000000	0.000000
	Author6	1.0	0.500000	NaN	0.50	0.5000	0.500	0.5000	0.50	1.0	0.400000	...	0.400	0.4	1.0	300.000000	NaN

6 rows × 24 columns

How To Aggregate Column-wise and with Multiple Functions?

In [55]:
```
# Let's group the book data based on 'author' and 'feedback'
grouped = book.groupby(by=['feedback', 'author'])
```

In [56]:
```
# The descriptive type aggregation functions can be passed like strings inside the 'agg' method
grouped['price'].agg('min')
```

Out[56]:
```
feedback  author
average   Author2    200
          Author5    200
good      Author1     50
          Author3    200
          Author4    100
          Author6    300
Name: price, dtype: int64
```

In [57]:
```
grouped['price'].agg('max')
```

Out[57]:
```
feedback  author
average   Author2    500
          Author5    500
good      Author1    400
          Author3    700
          Author4    100
          Author6    300
Name: price, dtype: int64
```

In [58]:
```
grouped['price'].agg('mean')
```

Out[58]:
```
feedback  author
average   Author2    260.000000
          Author5    350.000000
good      Author1    183.333333
          Author3    400.000000
          Author4    100.000000
          Author6    300.000000
Name: price, dtype: float64
```

In [59]:
```
# we can pass a list of functions at once, so that the column names will be replaced be the function names
grouped['price'].agg(['min', 'max', 'mean', 'std'])
```

Out[59]:

		min	max	mean	std
feedback	author				
average	Author2	200	500	260.000000	134.164079
	Author5	200	500	350.000000	212.132034
good	Author1	50	400	183.333333	189.296945
	Author3	200	700	400.000000	200.000000
	Author4	100	100	100.000000	0.000000
	Author6	300	300	300.000000	NaN

In [60]:
```
# Notice that the 'Author6' std is NaN, this is because both min and max price remains same
```

In [61]:
```
# We can also use our own column names for the result of the above type that is for the aggregated object with a list
 of functions
# To do that, we need to pass a list tuples, first name in the tuple is taken as the column name instead of function n
ame
# Let's pass two tuples and the keep the last function names as it is
grouped['price'].agg([('min_value', 'min'), ('max_value', 'max'), 'mean', 'std'])
```

Out[61]:

		min_value	max_value	mean	std
feedback	author				
average	Author2	200	500	260.000000	134.164079
	Author5	200	500	350.000000	212.132034
good	Author1	50	400	183.333333	189.296945
	Author3	200	700	400.000000	200.000000
	Author4	100	100	100.000000	0.000000
	Author6	300	300	300.000000	NaN

In [62]:
```
# we can also compute aggregation on multiple columns:
# Just pass the multiple columns and a list of functions to do on them in the 'agg' method
functions = [('min_value', 'min'), ('max_value', 'max'), 'mean', 'std']
result = grouped['price', 'max_dis'].agg(functions)
result
```

Out[62]:

		price				max_dis			
		min_value	max_value	mean	std	min_value	max_value	mean	std
feedback	author								
average	Author2	200	500	260.000000	134.164079	0.40	0.55	0.460000	0.065192
	Author5	200	500	350.000000	212.132034	0.35	0.40	0.375000	0.035355
good	Author1	50	400	183.333333	189.296945	0.35	0.40	0.383333	0.028868
	Author3	200	700	400.000000	200.000000	0.10	0.40	0.291667	0.102062
	Author4	100	100	100.000000	0.000000	0.30	0.45	0.383333	0.076376
	Author6	300	300	300.000000	NaN	0.50	0.50	0.500000	NaN

In [63]:
```
# if it is a very large data set we can use indexing method to extract the individual column details
result['price']
```

Out[63]:

		min_value	max_value	mean	std
feedback	author				
average	Author2	200	500	260.000000	134.164079
	Author5	200	500	350.000000	212.132034
good	Author1	50	400	183.333333	189.296945
	Author3	200	700	400.000000	200.000000
	Author4	100	100	100.000000	0.000000
	Author6	300	300	300.000000	NaN

In [64]:
```
# we can also use dictionary type of aggregation to apply potentially for one or more columns
# the key in the dictionary is used for resulting column name and the value is used to do aggregate operation
functions = {'min_value': 'min', 'max_value':'max', 'mean_value':'mean', 'std_value':'std'}
result = grouped['price', 'max_dis'].agg(functions)
result
```

```
C:\Users\user\Anaconda3\lib\site-packages\pandas\core\groupby\groupby.py:4656: FutureWarning: using a dict with renam
ing is deprecated and will be removed in a future version
  return super(DataFrameGroupBy, self).aggregate(arg, *args, **kwargs)
```

Out[64]:

		min_value		max_value		mean_value		std_value	
		price	max_dis	price	max_dis	price	max_dis	price	max_dis
feedback	author								
average	Author2	200	0.40	500	0.55	260.000000	0.460000	134.164079	0.065192
	Author5	200	0.35	500	0.40	350.000000	0.375000	212.132034	0.035355
good	Author1	50	0.35	400	0.40	183.333333	0.383333	189.296945	0.028868
	Author3	200	0.10	700	0.40	400.000000	0.291667	200.000000	0.102062
	Author4	100	0.30	100	0.45	100.000000	0.383333	0.000000	0.076376
	Author6	300	0.50	300	0.50	300.000000	0.500000	NaN	NaN

In [65]:
```
# To return the aggregated data without row index we can pass as_index=False to the 'groupby' method
grouped = book.groupby(by=['feedback', 'author'], as_index=False).min()
# functions = [('min_value', 'min'), ('max_value', 'max'), 'mean', 'std']
# result = grouped['price', 'max_dis'].agg(functions)
# result
grouped
```

Out[65]:

	feedback	author	book_name	price	min_dis	max_dis
0	*average*	*Author2*	*AAB*	*200*	*0.10*	*0.40*
1	*average*	*Author5*	*AAE*	*200*	*0.20*	*0.35*
2	*good*	*Author1*	*AAA*	*50*	*0.20*	*0.35*
3	*good*	*Author3*	*AAC*	*200*	*0.05*	*0.10*
4	*good*	*Author4*	*AAD*	*100*	*0.10*	*0.30*
5	*good*	*Author6*	*AAL*	*300*	*0.40*	*0.50*

In [ ]:

In [ ]: