# Categorizing a dataset and applying a function to each group, whether an aggregation or transformation

After loading, merging, and preparing a dataset, you may need to compute group statistics Pandas provides a flexible groupby interface, enabling slice,dice, and summarize datasets in a natural way.

Groupby mechanics: split-apply-combine

In [6]:
```
# A. Groupby Mechanics

import pandas as pd
import numpy as np
df=pd.DataFrame({'key1':['a','a','b','b','a'],
                 'key2':['one','two','one','two','one'],
                 'data1':np.random.randn(5),
                 'data2':np.random.randn(5)})
print(df)


#compute the man of the data1 column using the labels from key1
grouped=df['data1'].groupby(df['key1'])
print(grouped.mean())

m=df['data1'].groupby([df['key1'],df['key2']]).mean()
print(m)

#Data is grouped using two keys, and the resulting series now has a hierarchical
#index consisting of the unique pairs of keys observed:
print(m.unstack())
```

```
  key1 key2     data1     data2
0    a  one -0.331707  0.743593
1    a  two -1.284510  1.618296
2    b  one  0.823005 -0.143227
3    b  two -0.731866  0.158002
4    a  one  0.128044 -0.738243
key1
a   -0.496058
b    0.045570
Name: data1, dtype: float64
key1  key2
a     one    -0.101831
      two    -1.284510
b     one     0.823005
      two    -0.731866
Name: data1, dtype: float64
key2       one       two
key1
a    -0.101831 -1.284510
b     0.823005 -0.731866
```

In [7]:
```
states=np.array(['ohio','cali','cali','ohio','ohio'])
years=np.array([2005,2005,2006,2005,2006])
df['data1'].groupby([states,years]).mean()
```

Out[7]:
```
cali  2005    0.241678
      2006    0.331612
ohio  2005   -0.497429
      2006    0.342135
Name: data1, dtype: float64
```

In [ ]:
```

```

In [10]:
```
#grouping information may found in the same data frame
print(df.groupby('key1').mean())  #no key2 in the result because it is not numeric(nuis
#by default, all of the numeric columns are aggregated.

print(df.groupby(['key1','key2']).mean())
print(df.groupby(['key1','key2']).size())
#missing values in a group key will be excluded from the result
```

```
         data1     data2
key1
a     0.364609  0.793851
b    -0.586629 -0.291139
              data1     data2
key1 key2
a    one   0.426074  0.157417
     two   0.241678  2.066719
b    one   0.331612 -0.094451
     two  -1.504871 -0.487827
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

In [11]:
```
#Iterating over groups
#groupby objectbsupports iteration,generating a sequence of 2-tuples containing
#the group name along with the chunk of data.
for name,group in df.groupby('key1'):
    print(name)
    print(group)
```

```
a
  key1 key2     data1     data2
0    a  one  0.510013 -0.321910
1    a  two  0.241678  2.066719
4    a  one  0.342135  0.636743
b
  key1 key2     data1     data2
2    b  one  0.331612 -0.094451
3    b  two -1.504871 -0.487827
```

In [12]:
```
#In case of multiple keys, the first element in the tuple will be a tuple of key values
for (k1,k2),group in df.groupby(['key1','key2']):
    print((k1,k2))
    print(group)
```

```
('a', 'one')
  key1 key2     data1     data2
0    a  one  0.510013 -0.321910
4    a  one  0.342135  0.636743
('a', 'two')
  key1 key2     data1     data2
1    a  two  0.241678  2.066719
('b', 'one')
  key1 key2     data1     data2
2    b  one  0.331612 -0.094451
('b', 'two')
  key1 key2     data1     data2
3    b  two -1.504871 -0.487827
```

In [13]:
```
p=dict(list(df.groupby('key1')))
print(p)
```

```
{'a':   key1 key2     data1     data2
0    a  one  0.510013 -0.321910
1    a  two  0.241678  2.066719
4    a  one  0.342135  0.636743, 'b':   key1 key2     data1     data2
2    b  one  0.331612 -0.094451
3    b  two -1.504871 -0.487827}
```

In [14]:
```
print(df.dtypes)
```

```
key1      object
key2      object
data1    float64
data2    float64
dtype: object
```

In [15]:
```
grp=df.groupby(df.dtypes,axis=1)
for dtype,group in grp:
    print(dtype)
    print(group)
```

```
float64
      data1     data2
0  0.510013 -0.321910
1  0.241678  2.066719
2  0.331612 -0.094451
3 -1.504871 -0.487827
4  0.342135  0.636743
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

In [21]:
```
#Selecting a column or subset of columns
#df.groupby('key1')['data1'] = df['data1'].groupby(df['key1'])
print(df.groupby('key1')['data1'].groups)
print(df['data1'].groupby(df['key1']).groups)
```

```
{'a': [0, 1, 4], 'b': [2, 3]}
{'a': [0, 1, 4], 'b': [2, 3]}
```

In [24]:
```
#For large datasets, it may be desirable to aggregate only a few columns
print(df.groupby(['key1','key2'])[['data2']].mean())
print(df.groupby(['key1','key2'])['data2'].mean())
```

```
              data2
key1 key2
a    one   0.157417
     two   2.066719
b    one  -0.094451
     two  -0.487827
key1  key2
a     one     0.157417
      two     2.066719
b     one    -0.094451
      two    -0.487827
Name: data2, dtype: float64
```

In [5]:
```
#grouping with functions

#Any function passed as a group key will be called once per index value,
#with the return values being used as the group names.
import pandas as pd
import numpy as np
people = pd.DataFrame(np.random.randn(5, 5),
     columns=['a', 'b', 'c', 'd', 'e'],
     index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
people.groupby(len).groups
people.groupby(len).sum()
```

Out[5]:

| | a | b | c | d | e |
|---|---|---|---|---|---|
| **3** | -0.867807 | -0.538142 | 3.062285 | 0.365190 | 2.009963 |
| **5** | 0.216183 | -0.213166 | 0.223998 | 0.708886 | -2.389257 |
| **6** | -0.274615 | -0.366069 | -1.682027 | -1.000841 | 1.875392 |

In [ ]:
```
# B. Data Aggregation
#Aggregations refer to any data transformation that produces scalar values from
#arrays.
# Optimized groupby methods
count     Number of non-NA values in the group
sum       Sum of non-NA values
mean      Mean of non-NA values
median    Arithmetic median of non-NA values
std, var  Unbiased (n - 1 denominator) standard deviation and variance
min, max  Minimum and maximum of non-NA values
prod      Product of non-NA values
```

```
first, last First and last non-NA values

#You can use aggregations of your own devising and additionally call any method that
is also defined on the grouped object.
```

In [8]:
```
df
grouped = df.groupby('key1')
grouped['data1'].quantile(0.9)
```

Out[8]:
```
key1
a    0.036094
b    0.667518
Name: data1, dtype: float64
```

In [10]:
```
#To use your own aggregation functions, pass any function that aggregates an array to
#the aggregate or agg method:
def peak_to_peak(arr):
     return arr.max() - arr.min()
grouped.agg(peak_to_peak)
grouped.describe()
```

Out[10]:

| | data1 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | count | mean | std | min | 25% | 50% | 75% | max | count | mean |
| key1 | | | | | | | | | | |
| a | 3.0 | -0.496058 | 0.720476 | -1.284510 | -0.808108 | -0.331707 | -0.101831 | 0.128044 | 3.0 | 0.541215 |
| b | 2.0 | 0.045570 | 1.099459 | -0.731866 | -0.343148 | 0.045570 | 0.434287 | 0.823005 | 2.0 | 0.007388 |

In [15]:
```
df['data1'].agg(['mean', 'std',peak_to_peak])
```

Out[15]:
```
mean           -0.279407
std             0.806073
peak_to_peak    2.107515
Name: data1, dtype: float64
```

In [17]:
```
df.groupby('key1').agg(['mean', 'std',peak_to_peak])['data1']
```

Out[17]:

| | mean | std | peak_to_peak |
|---|---|---|---|
| key1 | | | |
| a | -0.496058 | 0.720476 | 1.412554 |
| b | 0.045570 | 1.099459 | 1.554870 |

In [19]:
```
#if you pass a list of (name, function) tuples, the first element of each tuple will be
#the DataFrame column names
df.groupby('key1').agg([('c1','mean'),('c2','std'),('c3',peak_to_peak)])['data1']
```

Out[19]:

| key1 | c1 | c2 | c3 |
|---|---|---|---|
| a | -0.496058 | 0.720476 | 1.412554 |
| b | 0.045570 | 1.099459 | 1.554870 |

In [21]:
```
flist=['mean','std']
df.groupby('key1')[['data1','data2']].agg(flist)
```

Out[21]:

| | data1 | | data2 | |
|---|---|---|---|---|
| | mean | std | mean | std |
| key1 | | | | |
| a | -0.496058 | 0.720476 | 0.541215 | 1.191233 |
| b | 0.045570 | 1.099459 | 0.007388 | 0.213001 |

In [22]:
```
# apply potentially different functions to one or more of
#the columns. To do this, pass a dict to agg that contains a mapping of column names
#to any of the function specifications
df.groupby('key1')[['data1','data2']].agg({'data1':'mean','data2':'std'})
```

Out[22]:

| key1 | data1 | data2 |
|---|---|---|
| a | -0.496058 | 1.191233 |
| b | 0.045570 | 0.213001 |

In [23]:
```
df.groupby('key1')[['data1','data2']].agg({'data1':['mean','min','max'],'data2':'std'})
```

Out[23]:

| | data1 | | | data2 |
|---|---|---|---|---|
| | mean | min | max | std |
| key1 | | | | |
| a | -0.496058 | -1.284510 | 0.128044 | 1.191233 |
| b | 0.045570 | -0.731866 | 0.823005 | 0.213001 |

In [26]:
```
#Returning Aggregated Data Without Row Indexes
#The aggregated data comes back with an index, potentially hierarchical, composed from
#this isn't always desirable, you can disable this behavior in most cases by passing
#as_index=False to groupby:

df.groupby(['key1','key2'],as_index=False)[['data1','data2']].agg(flist)
```

Out[26]:

| | | data1 | | data2 | |
|---|---|---|---|---|---|
| | | mean | std | mean | std |
| key1 | key2 | | | | |
| a | one | -0.101831 | 0.325093 | 0.002675 | 1.047816 |
| | two | -1.284510 | NaN | 1.618296 | NaN |
| b | one | 0.823005 | NaN | -0.143227 | NaN |
| | two | -0.731866 | NaN | 0.158002 | NaN |

In [1]:

```
#17-1-2023

#Group-wise Operations and Transformations
#Aggregation is only one kind of group operation which reducesreduce a one-dimensional
#array to a scalar value.   transform and apply will enable to do other kinds of group
import pandas as pd
import numpy as np
df=pd.DataFrame({'key1':['a','a','b','b','a'],
                 'key2':['one','two','one','two','one'],
                 'data1':np.random.randn(5),
                 'data2':np.random.randn(5)})
print(df)
```

```
  key1 key2     data1     data2
0    a  one  0.633329  0.279669
1    a  two  0.760641 -0.432298
2    b  one  0.112353  1.449803
3    b  two  2.775838 -2.066355
4    a  one -0.221395  1.748210
```

In [2]:

```
cmeans= df.groupby('key1').mean().add_prefix('mean_')
cmeans
```

Out[2]:

| | mean_data1 | mean_data2 |
|---|---|---|
| key1 | | |
| a | 0.390858 | 0.531860 |
| b | 1.444096 | -0.308276 |

In [13]:

```
#transform applies a function to each group, then places the results
#in the appropriate locations. If each group produces a scalar value, it will be propag
def demean(arr):
    return arr - arr.mean()
y=df.groupby('key2').mean()
print(y)
x=df.groupby('key1').transform(demean)
x
```

```
         data1     data2
key2
```

```
one    0.174763  1.159227
two    1.768239 -1.249327
```

Out[13]:

| | data1 | data2 |
|---|---|---|
| **0** | 0.242471 | -0.252191 |
| **1** | 0.369782 | -0.964159 |
| **2** | -1.331742 | 1.758079 |
| **3** | 1.331742 | -1.758079 |
| **4** | -0.612253 | 1.216350 |

In [ ]:

```
#Apply: General split-apply-combine
#apply splits the object being manipulated into pieces, invokes the passed function on
#attempts to concatenate the pieces together.
```

In [ ]: