PRACTICE SESSION 7

DATA AGGREGATION AND GROUP OPERATIONS

Data Aggregation and Group Operations

- Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow.
- After loading, merging, and preparing a dataset, you may need to compute group statistics or possibly *pivot tables* for reporting or visualization purposes.
- pandas provides a flexible **groupby** interface, enabling you to slice, dice, and summarize datasets in a natural way.
- we can perform quite complex group operations by utilizing any function that accepts a pandas object or NumPy array with the help of the expressiveness of Python and pandas,

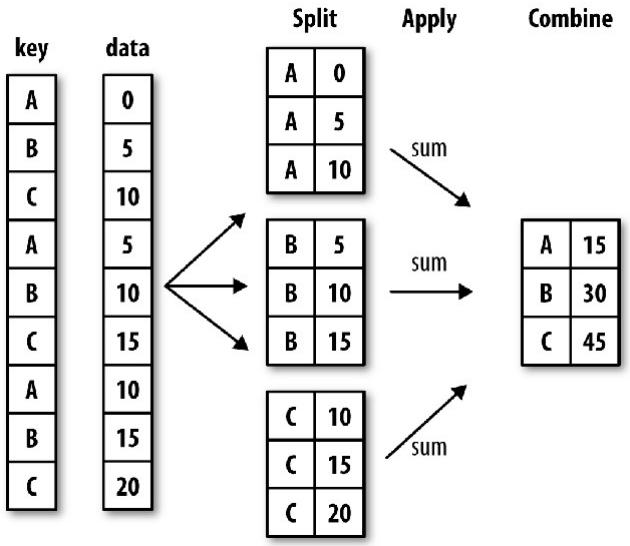
Data Aggregation and Group Operations

> In this module, you will learn how to:

- Split a pandas object into pieces using one or more keys (in the form of functions, arrays, or DataFrame column names)
- Calculate group summary statistics, like count, mean, or standard deviation, or a user-defined function
- Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection
- Compute pivot tables and cross-tabulations
- Perform quantile analysis and other statistical group analyses

GROUPBY MECHANICS

GroupBy Mechanics



GroupBy Mechanics

> Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame
- A dict or Series giving a correspondence between the values on the axis being grouped and the group names
- A function to be invoked on the axis index or the individual labels in the index

GroupBy Mechanics

```
In [10]: 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)})

In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>
```

GroupBy Mechanics

> This **`grouped`** variable is now a *GroupBy* object.

> It has not actually computed anything yet except for some intermediate data about the group key **`df['key1']`**.

> The idea is that this object has all of the information needed to then apply some operation to each of the groups.

```
In [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

GroupBy Mechanics

> If instead we had passed multiple arrays as a list, we'd get something different:

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
key1  key2
a     one      0.880536
      two      0.478943
b     one     -0.519439
      two     -0.555730
Name: data1, dtype: float64
```

```
In [17]: means.unstack()
Out[17]:
key2        one       two
key1
a      0.880536  0.478943
b     -0.519439 -0.555730
```

GroupBy Mechanics

> The group keys can be all Series

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
            2006    1.965781
Name: data1, dtype: float64
```

GroupBy Mechanics

> Frequently the grouping information is found in the same DataFrame as the data you want to work on.

> In that case, you can pass column names as the group keys:

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384
In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
              data1     data2
key1 key2
a    one   0.880536  1.319920
     two   0.478943  0.092908
b    one  -0.519439  0.281746
     two  -0.555730  0.769023
```

GroupBy Mechanics

> Regardless of the objective in using groupby, a generally useful GroupBy method is size, which returns a Series containing group sizes:

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

Iterating Over Groups

> The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

Iterating Over Groups

> With multiple keys, the first element in the tuple will be a tuple of key values

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

Iterating Over Groups

> By default **`groupby`** groups on **`axis=0`**, but you can group on any of the other axes

> For example, we could group the columns of our example **`df`** here by **`dtype`**

```
In [28]: df.dtypes
Out[28]:
data1    float64
data2    float64
key1      object
key2      object
dtype: object
```

```
In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

Iterating Over Groups

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
   ....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

Selecting a Column or Subset of Columns

> Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of column subsetting for aggregation

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

> Syntactic sugar for

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Selecting a Column or Subset of Columns

> Especially for large datasets, it may be desirable to aggregate only a few columns

> To compute means for just the **`data2`** column and get the result as a DataFrame

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

Selecting a Column or Subset of Columns

> The object returned by this indexing operation is a grouped DataFrame if a list or array is passed or a grouped Series if only a single column name is passed as a scalar:

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa30c78da0>

In [34]: s_grouped.mean()
Out[34]:
key1  key2
a     one     1.319920
      two     0.092908
b     one     0.281746
      two     0.769023
Name: data2, dtype: float64
```

Grouping with Dicts and Series

> Grouping information may exist in a form other than an array

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....:                       columns=['a', 'b', 'c', 'd', 'e'],
   ....:                       index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Grouping with Dicts and Series

> Now, suppose I have a group correspondence for the columns and want to sum together the columns by group

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

> You could construct an array from this dict to pass to **groupby**

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
           blue       red
Joe    0.503905  1.063885
Steve  1.297183 -1.553778
Wes   -1.021228 -1.116829
Jim    0.524712  1.770545
Travis -4.230992 -2.405455
```

Grouping with Dicts and Series

> The same functionality holds for Series, which can be viewed as a fixed-size mapping

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a       red
b       red
c      blue
d      blue
e       red
f    orange
dtype: object
```

```
In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

Grouping with Functions

> Using Python functions is a more generic way of defining a group mapping compared with a dict or Series

> Any function passed as a group key will be called once per index value, with the return values being used as the group names

> Suppose you wanted to group by the length of the names

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Grouping with Functions

> Mixing functions with arrays, dicts, or Series is not a problem as everything gets converted to arrays internally

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Grouping by Index Levels

> A final convenience for hierarchically indexed datasets is the ability to aggregate using one of the levels of an axis index.

```
In [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....:                                      [1, 3, 5, 1, 3]],
   ....:                                      names=['cty', 'tenor'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)

In [49]: hier_df
Out[49]:
cty          US                            JP
tenor         1         3         5         1         3
0      0.560145 -1.265934  0.119827 -1.063512  0.332883
1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2      0.286350  0.377984 -0.753887  0.331286  1.349742
3      0.069877  0.246674 -0.011862  1.004812  1.327195
```

Grouping by Index Levels

> To group by level, pass the level number or name using the **`level`** keyword:

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

DATA AGGREGATION

Data Aggregation

- Aggregations refer to any data transformation that produces scalar values from arrays
- The preceding examples have used several of them, including **`mean`**, **`count`**, **`min`**, and **`sum`**
- You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object

Data Aggregation

- While quantile is not explicitly implemented for GroupBy, it is a Series method and thus available for use.
- Internally, GroupBy efficiently slices up the Series, calls piece.quantile(0.9) for each piece, and then assembles those results together into the result object:

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a    1.668413
b   -0.523068
Name: data1, dtype: float64
```

Data Aggregation

- To use your own aggregation functions, pass any function that aggregates an array to the **`aggregate`** or **`agg`** method

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

Data Aggregation

> Some methods like describe also work, even though they are not aggregations

```
In [56]: grouped.describe()
Out[56]:
     data1                                                                \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943  1.222362
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 -0.528512
              data2                                                       \
          max count      mean       std       min       25%       50%
key1
a     1.965781   3.0  0.910916  0.712217  0.092908  0.669671  1.246435
b    -0.519439   2.0  0.525384  0.344556  0.281746  0.403565  0.525384

          75%       max
key1
a     1.319920  1.393406
b     0.647203  0.769023
```

Optimized **groupby** methods

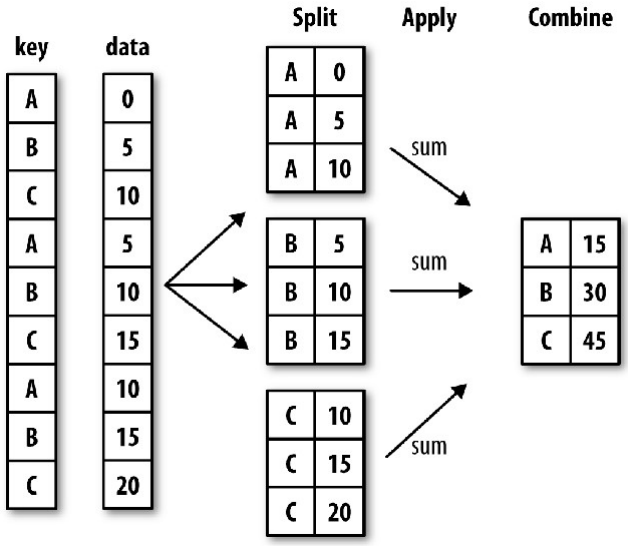
Function name	Description
count	Number of non-NA values in the group
sum	Sum of non-NA values
mean	Mean of non-NA values
median	Arithmetic median of non-NA values
std, **var**	Unbiased (n – 1 denominator) standard deviation and variance
min, **max**	Minimum and maximum of non-NA values
prod	Product of non-NA values
first, **last**	First and last non-NA values

Column-Wise and Multiple Function Application

Returning Aggregated Data Without Row Indexes

APPLY: GENERAL SPLIT-APPLY-COMBINE

Apply: General split-apply-combine



Apply: General split-apply-combine

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_pct
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
232       11.61  3.39     No  Sat  Dinner     2  0.291990
67         3.07  1.00    Yes  Sat  Dinner     1  0.325733
178        9.60  4.00    Yes  Sun  Dinner     2  0.416667
172        7.25  5.15    Yes  Sun  Dinner     2  0.710345
```

Apply: General split-apply-combine

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     88        24.71  5.85     No  Thur   Lunch     2  0.236746
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       232       11.61  3.39     No   Sat  Dinner     2  0.291990
Yes    109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

Apply: General split-apply-combine

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[77]:
                 total_bill    tip smoker   day    time  size   tip_pct
smoker day
No     Fri  94        22.75   3.25     No   Fri  Dinner     2  0.142857
       Sat  212       48.33   9.00     No   Sat  Dinner     4  0.186220
       Sun  156       48.17   5.00     No   Sun  Dinner     6  0.103799
       Thur 142       41.19   5.00     No  Thur   Lunch     5  0.121389
Yes    Fri  95        40.17   4.73    Yes   Fri  Dinner     4  0.117750
       Sat  170       50.81  10.00    Yes   Sat  Dinner     3  0.196812
       Sun  182       45.35   3.50    Yes   Sun  Dinner     3  0.077178
       Thur 197       43.11   5.00    Yes  Thur   Lunch     4  0.115982
```

Apply: General split-apply-combine

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()

In [79]: result
Out[79]:
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.185014
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.195059

             max
smoker
No      0.291990
Yes     0.710345
```

Apply: General split-apply-combine

```
In [80]: result.unstack('smoker')
Out[80]:
              smoker
count  No     151.000000
       Yes     93.000000
mean   No       0.159328
       Yes      0.163196
std    No       0.039910
       Yes      0.085119
min    No       0.056797
       Yes      0.035638
25%    No       0.136906
       Yes      0.106771
50%    No       0.155625
       Yes      0.153846
75%    No       0.185014
       Yes      0.195059
max    No       0.291990
       Yes      0.710345
dtype: float64
```

Suppressing the Group Keys

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
     total_bill   tip smoker   day    time  size   tip_pct
88        24.71  5.85     No  Thur   Lunch     2  0.236746
185       20.69  5.00     No   Sun  Dinner     5  0.241663
51        10.29  2.60     No   Sun  Dinner     2  0.252672
149        7.51  2.00     No  Thur   Lunch     2  0.266312
232       11.61  3.39     No   Sat  Dinner     2  0.291990
109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

Example: Filling Missing Values with Group-Specific Values

> When cleaning up missing data, in some cases you will replace data observations using **dropna**, but in others you may want to impute (fill in) the null (NA) values using a fixed value or some value derived from the data.

> **fillna** is the right tool to use

```
In [91]: s = pd.Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0         NaN
1   -0.125921
2         NaN
3   -0.884475
4         NaN
5    0.227290
dtype: float64
```

```
In [94]: s.fillna(s.mean())
Out[94]:
0   -0.261035
1   -0.125921
2   -0.261035
3   -0.884475
4   -0.261035
5    0.227290
dtype: float64
```

Example: Filling Missing Values with Group-Specific Values

> Suppose you need the fill value to vary by group.

> One way to do this is to group the data and use **apply** with a function that calls **fillna** on each data chunk

Example: Filling Missing Values with Group-Specific Values

> Some sample data on US states divided into eastern and western regions

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   ....:           'Oregon', 'Nevada', 'California', 'Idaho']

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=states)

In [98]: data
Out[98]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.365757
Florida      -0.375842
Oregon        0.329939
Nevada        0.981994
California    1.105913
Idaho        -1.613716
dtype: float64
```

Example: Filling Missing Values with Group-Specific Values

> Let's set some values in the data to be missing:

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
Ohio          0.922264
New York     -2.153545
Vermont            NaN
Florida      -0.375842
Oregon        0.329939
Nevada             NaN
California    1.105913
Idaho              NaN
dtype: float64
```

```
In [101]: data.groupby(group_key).mean()
Out[101]:
East   -0.535707
West    0.717926
dtype: float64
```

Example: Filling Missing Values with Group-Specific Values

> We can fill the NA values using the group means like so:

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.535707
Florida      -0.375842
Oregon        0.329939
Nevada        0.717926
California    1.105913
Idaho         0.717926
dtype: float64
```

Example: Filling Missing Values with Group-Specific Values

> You might have predefined fill values in your code that vary by group.

> Since the groups have a name attribute set internally, we can use that:

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio          0.922264
New York     -2.153545
Vermont       0.500000
Florida      -0.375842
Oregon        0.329939
Nevada       -1.000000
California    1.105913
Idaho        -1.000000
dtype: float64
```

Example: Random Sampling and Permutation

> Suppose you wanted to draw a random samplefrom a large dataset for Monte Carlo simulation purposes or some other application.

> There are a number of ways to perform the “draws”

> To demonstrate, here’s a way to construct a deck of English-style playing cards:

```
# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

```
In [108]: deck[:13]
Out[108]:
AH       1
2H       2
3H       3
4H       4
5H       5
6H       6
7H       7
8H       8
9H       9
10H     10
JH      10
KH      10
QH      10
dtype: int64
```

Example: Random Sampling and Permutation

> Drawing a hand of five cards from the deck could be written as:

```
In [109]: def draw(deck, n=5):
   .....:     return deck.sample(n)

In [110]: draw(deck)
Out[110]:
AD     1
8C     8
5H     5
KC    10
2C     2
dtype: int64
```

Example: Random Sampling and Permutation

> Suppose you wanted two random cards from each suit.

> Because the suit is the last character of each card name, we can group based on this and use apply:

```
In [111]: get_suit = lambda card: card[-1] # last letter is suit

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
C  2C     2
   3C     3
D  KD    10
   8D     8
H  KH    10
   3H     3
S  2S     2
   4S     4
dtype: int64
```

Example: Random Sampling and Permutation

> Alternatively, we could write

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
KC    10
JC    10
AD     1
5D     5
5H     5
6H     6
7S     7
KS    10
dtype: int64
```

Example: Group Weighted Average and Correlation

```
In [114]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
   .....:                                 'b', 'b', 'b', 'b'],
   .....:                    'data': np.random.randn(8),
   .....:                    'weights': np.random.rand(8)})

In [115]: df
Out[115]:
  category      data   weights
0        a  1.561587  0.957515
1        a  1.219984  0.347267
2        a -0.482239  0.581362
3        a  0.315667  0.217091
4        b -0.047852  0.894406
5        b -0.454145  0.918564
6        b -0.556774  0.277825
7        b  0.253321  0.955905
```

Example: Group Weighted Average and Correlation

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
```

```
In [118]: grouped.apply(get_wavg)
Out[118]:
category
a    0.811643
b   -0.122262
dtype: float64
```

```
In [119]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dates=True,
   .....:                        index_col=0)

In [120]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

Example: Group Weighted Average and Correlation

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

```
In [123]: rets = close_px.pct_change().dropna()
```

Example: Group Weighted Average and Correlation

```
In [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
Out[126]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265  1.0
2004  0.374283  0.588531  0.557742  1.0
2005  0.467540  0.562374  0.631010  1.0
2006  0.428267  0.406126  0.518514  1.0
2007  0.508118  0.658770  0.786264  1.0
2008  0.681434  0.804626  0.828303  1.0
2009  0.707103  0.654902  0.797921  1.0
2010  0.710105  0.730118  0.839057  1.0
2011  0.691931  0.800996  0.859975  1.0
```

Example: Group Weighted Average and Correlation

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

Example: Group-Wise Linear Regression

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
          SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

PIVOT TABLES AND CROSS-TABULATION

Pivot Tables and Cross-Tabulation

> A *pivot table* is a data summarization tool frequently found in spreadsheet programs and other data analysis software.

> It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns.

> Pivot tables in Python with pandas are made possible through the **`groupby`** facility described in this module combined with reshape operations utilizing hierarchical indexing.

> DataFrame has a **`pivot_table`** method, and there is also a top-level **`pandas.pivot_table`** function.

> In addition to providing a convenience interface to groupby, **`pivot_table`** can add partial totals, also known as *margins*.

Pivot Tables and Cross-Tabulation

> Suppose you wanted to compute a table of group means (the default **`pivot_table`** aggregation type) arranged by **`day`** and **`smoker`** on the rows:

```
In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                 size       tip   tip_pct  total_bill
day  smoker
Fri  No      2.250000  2.812500  0.151650   18.420000
     Yes     2.066667  2.714000  0.174783   16.813333
Sat  No      2.555556  3.102889  0.158048   19.661778
     Yes     2.476190  2.875476  0.147906   21.276667
Sun  No      2.929825  3.167895  0.160113   20.506667
     Yes     2.578947  3.516842  0.187250   24.120000
Thur No      2.488889  2.673778  0.160298   17.113111
     Yes     2.352941  3.030000  0.163863   19.190588
```

Pivot Tables and Cross-Tabulation

> Now, suppose we want to aggregate only **`tip_pct`** and **size**, and additionally group by **time**

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker')
Out[131]:
                size             tip_pct
smoker             No       Yes        No       Yes
time   day
Dinner Fri   2.000000  2.222222  0.139622  0.165347
       Sat   2.555556  2.476190  0.158048  0.147906
       Sun   2.929825  2.578947  0.160113  0.187250
       Thur  2.000000       NaN  0.159744       NaN
Lunch  Fri   3.000000  1.833333  0.187735  0.188937
       Thur  2.500000  2.352941  0.160311  0.163863
```

Pivot Tables and Cross-Tabulation

> We could augment this table to include partial totals by passing margins=True.

> This has the effect of adding **`All`** row and column labels, with corresponding values being the group statistics for all the data within a single tier:

```
In [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker', margins=True)
Out[132]:
                size                     tip_pct
smoker             No       Yes       All        No       Yes       All
time   day
Dinner Fri   2.000000  2.222222  2.166667  0.139622  0.165347  0.158916
       Sat   2.555556  2.476190  2.517241  0.158048  0.147906  0.153152
       Sun   2.929825  2.578947  2.842105  0.160113  0.187250  0.166897
       Thur  2.000000       NaN  2.000000  0.159744       NaN  0.159744
Lunch  Fri   3.000000  1.833333  2.000000  0.187735  0.188937  0.188765
       Thur  2.500000  2.352941  2.459016  0.160311  0.163863  0.161301
All          2.668874  2.408602  2.569672  0.159328  0.163196  0.160803
```

Pivot Tables and Cross-Tabulation

> To use a different aggregation function, pass it to **`aggfunc`**.

```
In [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day',
   .....:                  aggfunc=len, margins=True)
Out[133]:
day             Fri   Sat   Sun  Thur    All
time   smoker
Dinner No       3.0  45.0  57.0   1.0  106.0
       Yes      9.0  42.0  19.0   NaN   70.0
Lunch  No       1.0   NaN   NaN  44.0   45.0
       Yes      6.0   NaN   NaN  17.0   23.0
All            19.0  87.0  76.0  62.0  244.0
```

Pivot Tables and Cross-Tabulation

> If some combinations are empty (or otherwise NA), you may wish to pass a **`fill_value`**:

```
In [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],
   .....:                  columns='day', aggfunc='mean', fill_value=0)
Out[134]:
day                     Fri       Sat       Sun      Thur
time   size smoker
Dinner 1    No      0.000000  0.137931  0.000000  0.000000
            Yes     0.000000  0.325733  0.000000  0.000000
       2    No      0.139622  0.162705  0.168859  0.159744
            Yes     0.171297  0.148668  0.207893  0.000000
       3    No      0.000000  0.154661  0.152663  0.000000
            Yes     0.000000  0.144995  0.152660  0.000000
       4    No      0.000000  0.150096  0.148143  0.000000
            Yes     0.117750  0.124515  0.193370  0.000000
       5    No      0.000000  0.000000  0.206928  0.000000
            Yes     0.000000  0.106572  0.065660  0.000000
...                      ...       ...       ...       ...
```

Pivot Tables and Cross-Tabulation

```
...                      ...       ...       ...       ...
Lunch  1    No       0.000000  0.000000  0.000000  0.181728
            Yes      0.223776  0.000000  0.000000  0.000000
       2    No       0.000000  0.000000  0.000000  0.166005
            Yes      0.181969  0.000000  0.000000  0.158843
       3    No       0.187735  0.000000  0.000000  0.084246
            Yes      0.000000  0.000000  0.000000  0.204952
       4    No       0.000000  0.000000  0.000000  0.138919
            Yes      0.000000  0.000000  0.000000  0.155410
       5    No       0.000000  0.000000  0.000000  0.121389
       6    No       0.000000  0.000000  0.000000  0.173706
[21 rows x 4 columns]
```

`pivot_table` options

Function name	Description
`values`	Column name or names to aggregate; by default aggregates all numeric columns
`index`	Column names or other group keys to group on the rows of the resulting pivot table
`columns`	Column names or other group keys to group on the columns of the resulting pivot table
`aggfunc`	Aggregation function or list of functions ('mean' by default); can be any function valid in a groupby context
`fill_value`	Replace missing values in result table
`dropna`	If True, do not include columns whose entries are all NA
`margins`	Add row/column subtotals and grand total (False by default)

Cross-Tabulations: Crosstab

> A cross-tabulation (or crosstab for short) is a special case of a pivot table that computes group frequencies

```
In [138]: data
Out[138]:
   Sample Nationality    Handedness
0       1         USA  Right-handed
1       2       Japan   Left-handed
2       3         USA  Right-handed
3       4       Japan  Right-handed
4       5       Japan   Left-handed
5       6       Japan  Right-handed
6       7         USA  Right-handed
7       8         USA   Left-handed
8       9       Japan  Right-handed
9      10         USA  Right-handed
```

Cross-Tabulations: Crosstab

> As part of some survey analysis, we might want to summarize this data by nationality and handedness.

> You could use **`pivot_table`** to do this, but the **`pandas.crosstab`** function can be more convenient

```
In [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True)
Out[139]:
Handedness   Left-handed  Right-handed  All
Nationality
Japan                  2             3    5
USA                    1             4    5
All                    3             7   10
```

Cross-Tabulations: Crosstab

> The first two arguments to crosstab can each either be an array or Series or a list of arrays

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[140]:
smoker        No  Yes  All
time   day
Dinner Fri     3    9   12
       Sat    45   42   87
       Sun    57   19   76
       Thur    1    0    1
Lunch  Fri     1    6    7
       Thur   44   17   61
All          151   93  244
```