

Reference

This is a DataCamp course.

Extracting and transforming data

Index, slice, filter, and transform DataFrames.

Positional and labeled indexing

```
In [1]: import pandas as pd
election = pd.read_csv("pennsylvania2012_turnout.csv", index_col = 0)
x = 4
y = 4
print(election.iloc[x, y] == election.loc['Bedford', 'winner'])
```

True

Indexing and column rearrangement

```
In [6]: import pandas as pd
election = pd.read_csv("pennsylvania2012_turnout.csv", index_col='county')
results = election[['winner', 'total', 'voters']]
print(results.head())
```

	winner	total	voters
county			
Adams	Romney	41973	61156
Allegheny	Obama	614671	924351
Armstrong	Romney	28322	42147
Beaver	Romney	80015	115157
Bedford	Romney	21444	32189

Slicing rows

```
In [2]: p_counties = election.loc['Perry':'Potter']  
# print(p_counties)  
p_counties_rev = election.loc['Potter':'Perry':-1]  
# print(p_counties_rev)
```

Slicing columns

```
In [3]: left_columns = election.loc[:, : 'Obama']  
middle_columns = election.loc[:, 'Obama': 'winner']  
right_columns = election.loc[:, 'Romney':]
```

Subselecting DataFrames with lists

```
In [4]: rows = ['Philadelphia', 'Centre', 'Fulton']  
cols = ['winner', 'Obama', 'Romney']  
three_counties = election.loc[rows, cols]  
print(three_counties)
```

	winner	Obama	Romney
county			
Philadelphia	Obama	85.224251	14.051451
Centre	Romney	48.948416	48.977486
Fulton	Romney	21.096291	77.748861

Thresholding data

```
In [8]: high_turnout = election['turnout'] > 70  
high_turnout_df = election.loc[high_turnout]  
# print(high_turnout_df)
```

Filtering columns using other columns

```
In [7]: import numpy as np
too_close = election['margin'] < 1
election.loc[too_close, 'winner'] = np.nan
# print(election.info())
```

Filtering using NaNs

```
In [9]: import pandas as pd
titanic = pd.read_csv("titanic.csv")
df = titanic[['age', 'cabin']]
print(df.dropna(how='any').shape)
print(df.dropna(how='all').shape)
print(titanic.dropna(thresh=1000, axis='columns').info())
```

```
(272, 2)
(1069, 2)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 10 columns):
pclass      1309 non-null int64
survived     1309 non-null int64
name        1309 non-null object
sex         1309 non-null object
age         1046 non-null float64
sibsp       1309 non-null int64
parch       1309 non-null int64
ticket      1309 non-null object
fare        1308 non-null float64
embarked    1307 non-null object
dtypes: float64(2), int64(4), object(4)
memory usage: 102.3+ KB
None
```

Using apply() to transform a column

```
In [ ]: def to_celsius(F):
        return 5/9*(F - 32)

df_celsius = weather[['Mean TemperatureF', 'Mean Dew PointF']].apply(to_celsius)
#Here better use lambda expression
df_celsius.columns = ['Mean TemperatureC', 'Mean Dew PointC']
print(df_celsius.head())
```

Using .map() with a dictionary

```
In [20]: red_vs_blue = {'Obama':'blue', 'Romney':'red'}

# Use the dictionary to map the 'winner' column to the new column: election['color']
election['color'] = election['winner'].map(red_vs_blue)
print(election.head())
```

	state	total	Obama	Romney	winner	voters	turnout	\
county								
Adams	PA	41973	35.482334	63.112001	Romney	61156	68.632677	
Allegheny	PA	614671	56.640219	42.185820	Obama	924351	66.497575	
Armstrong	PA	28322	30.696985	67.901278	Romney	42147	67.198140	
Beaver	PA	80015	46.032619	52.637630	Romney	115157	69.483401	
Bedford	PA	21444	22.057452	76.986570	Romney	32189	66.619031	

	margin	color
county		
Adams	27.629667	red
Allegheny	14.454399	blue
Armstrong	37.204293	red
Beaver	6.605012	red
Bedford	54.929118	red

Using vectorized functions

```
In [21]: from scipy.stats import zscore
turnout_zscore = zscore(election['turnout'])
print(type(turnout_zscore))
election['turnout_zscore'] = turnout_zscore
```

```
<class 'numpy.ndarray'>
      state  total      Obama      Romney  winner  voters  turnout \
county
Adams      PA   41973  35.482334  63.112001  Romney   61156  68.632677
Allegheny  PA  614671  56.640219  42.185820   Obama  924351  66.497575
Armstrong  PA   28322  30.696985  67.901278  Romney   42147  67.198140
Beaver     PA   80015  46.032619  52.637630  Romney  115157  69.483401
Bedford    PA   21444  22.057452  76.986570  Romney   32189  66.619031

      margin color  turnout_zscore
county
Adams      27.629667   red         0.853734
Allegheny  14.454399  blue         0.439846
Armstrong  37.204293   red         0.575650
Beaver      6.605012   red         1.018647
Bedford    54.929118   red         0.463391
```

Advanced indexing

MultIndexes, or hierarchical indexes. keep in mind that **indexes are immutable objects**. Note the whole index can be replaced, although we cannot modify individually.

Index values and names

```
sales.index[0]
```

```
In [2]: import pandas as pd
sales = pd.read_csv("sales.csv")
sales.index = range(len(sales))
print(sales.head())
```

	month	eggs	salt	spam
0	Jan	47	12.0	17
1	Feb	110	50.0	31
2	Mar	221	89.0	72
3	Apr	77	87.0	20
4	May	132	NaN	52

Changing index of a DataFrame

```
In [6]: import pandas as pd
sales = pd.read_csv("sales.csv", index_col = 'month')
sales.head()

new_idx = [month.upper() for month in sales.index]
sales.index = new_idx

print(sales)
```

	eggs	salt	spam
JAN	47	12.0	17
FEB	110	50.0	31
MAR	221	89.0	72
APR	77	87.0	20
MAY	132	NaN	52
JUN	205	60.0	55

Changing index name labels

Usually the index was not labeled with a name. Here we set its name to 'MONTHS'. Similarly, if all the columns are related in some way, you can provide a label for the set of columns. **Normally we don't have a name for columns.**

```
In [7]: print(sales)
sales.index.name = 'MONTHS'
print(sales)
sales.columns.name = 'PRODUCTS'

print(sales)
```

	eggs	salt	spam
JAN	47	12.0	17
FEB	110	50.0	31
MAR	221	89.0	72
APR	77	87.0	20
MAY	132	NaN	52
JUN	205	60.0	55

	eggs	salt	spam
JAN	47	12.0	17
FEB	110	50.0	31
MAR	221	89.0	72
APR	77	87.0	20
MAY	132	NaN	52
JUN	205	60.0	55

	eggs	salt	spam
JAN	47	12.0	17
FEB	110	50.0	31
MAR	221	89.0	72
APR	77	87.0	20
MAY	132	NaN	52
JUN	205	60.0	55

Building an index, then a DataFrame

```
In [9]: import pandas as pd
sales = pd.read_csv("sales.csv", index_col = None)
sales = sales.drop('month',axis = 'columns')
print(sales.head())

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales.index = months
print(sales)
```

	eggs	salt	spam
0	47	12.0	17
1	110	50.0	31
2	221	89.0	72
3	77	87.0	20
4	132	NaN	52

	eggs	salt	spam
Jan	47	12.0	17
Feb	110	50.0	31
Mar	221	89.0	72
Apr	77	87.0	20
May	132	NaN	52
Jun	205	60.0	55

Setting & sorting a MultiIndex

Be familiar with the following ways:

```
sales.index = months
```

```
sales.set_index(['state', 'month'])
```



```
In [10]: import pandas as pd
sales = pd.read_csv("sales_states.csv")
sales = sales.set_index(['state', 'month'])
sales = sales.sort_index()
print(sales)
```

		eggs	salt	spam
state	month			
CA	1	47	12.0	17
	2	110	50.0	31
NY	1	221	89.0	72
	2	77	87.0	20
TX	1	132	NaN	52
	2	205	60.0	55

Extracting data with a MultiIndex

```
In [43]: print(sales.loc[['CA', 'TX']])
print(sales.loc['CA':'TX'])
```

		eggs	salt	spam
state	month			
CA	1	47	12.0	17
	2	110	50.0	31
TX	1	132	NaN	52
	2	205	60.0	55

		eggs	salt	spam
state	month			
CA	1	47	12.0	17
	2	110	50.0	31
NY	1	221	89.0	72
	2	77	87.0	20
TX	1	132	NaN	52
	2	205	60.0	55

Using .loc[] with nonunique indexes

```
In [44]: import pandas as pd
sales = pd.read_csv("sales_states.csv")
sales = sales.set_index(['state'])
print(sales)
print(sales.loc['NY'])
```

	month	eggs	salt	spam
state				
CA	1	47	12.0	17
CA	2	110	50.0	31
NY	1	221	89.0	72
NY	2	77	87.0	20
TX	1	132	NaN	52
TX	2	205	60.0	55

	month	eggs	salt	spam
state				
NY	1	221	89.0	72
NY	2	77	87.0	20

Indexing multiple levels of a MultiIndex

```
In [13]: import pandas as pd
sales = pd.read_csv("sales_states.csv")
sales = sales.set_index(['state', 'month'])
print(sales)

NY_month1 = sales.loc(['NY', 1), :]
print(NY_month1)

CA_TX_month2 = sales.loc(['CA', 'TX'], 2), :]
print(CA_TX_month2)

# Look up data for all states in month 2: all_month2
all_month2 = sales.loc[(slice(None), 2), :]
print(all_month2)
```

		eggs	salt	spam
state	month			
CA	1	47	12.0	17
	2	110	50.0	31
NY	1	221	89.0	72
	2	77	87.0	20
TX	1	132	NaN	52
	2	205	60.0	55
eggs		221.0		
salt		89.0		
spam		72.0		
Name: (NY, 1), dtype: float64				
		eggs	salt	spam
state	month			
CA	2	110	50.0	31
TX	2	205	60.0	55
		eggs	salt	spam
state	month			
CA	2	110	50.0	31
NY	2	77	87.0	20
TX	2	205	60.0	55

Rearranging and reshaping data

About pivot table

- Pivot tables enable a person to arrange and rearrange (or "**pivot**") statistics in order to draw attention to useful information.
- Pivoting is closely related to groupby in sql, it is almost the EXCEL/spreadsheet equivalent of GROUP BY, except some displaying differences.
- If creating a pivot table by just one categorical variable for ROW or COLUMN, then we can obtain almost the same results as group by.
- If creating a pivot table with two categorical variables, then we can still obtain the similar results by 'grouping by two column names'. However, pivot table shows in a more clear way with one variable varying along x-axis and the other variable along y-axis. Usually, for one pair of fixed (x,y) coordinates, we have a group to aggregate upon. So in this case, pivoting and grouping are equivalent except the displaying difference. However, in sql, now they also have similar operation PIVOT, which achieve the similar results as in spreadsheet pivot table.
- In SQL, we can use "GROUP BY column_a, column_b", and then aggregate by, e.g., "count(column_a / column_b)", or "count(other_column)". In the same way, we can also do so with pivot. The VALUE we aggregated upon in pivot table can be the categorical variables themselves, or the values corresponding to other categorical variables.
- In SQL, we can GROUP BY many columns (beyond two). We also can create pivot table with multiple ROW variables, and multiple COLUMN variables, though it might not so useful. **It is not true that for ROW (or COLUMN) we can only choose one variable.**
- With pivot table and aggregating upon the categorical variables chosen for creating pivot table, we essentially obtain the so-called contingency table for χ^2 testing for categorical variable selection in machine learning. Below is an example:

A flat table made from two categorical variables sex (F/M) and interest (Art, math, science) for χ^2 testing.

```
Sex,      Interest
Male,     Art
Female,    Math
Male,      Science
Male,      Math
.....
```

We can summarize the collected observations in a table with one variable corresponding to columns and another variable corresponding to rows. Each cell in the table corresponds to the count or frequency of observations that correspond to the row and column categories.

Historically, a table summarization of two categorical variables in this form is called a contingency table.

	Science,	Math,	Art
Male	20,	30,	15
Female	20,	15,	30

From the description earlier, this is just a pivot table in which we aggregate upon the categorical variables, with which the pivot table is created. So contingency table is actually a special case of pivot table, or special case of group by in sql.

- As in group by of sql where we can use multiple column names for grouping, in pivot table, we can also use multiple categorical names/variables as ROW or COLUMN. In other words, the ROW or COLUMN name to create pivot table is not necessarily a single categorical variable.
- pivot table is only useful for complicated tables with e.g., many categorical variables and large number of records. Otherwise, sorting, filtering, subtotal is more than enough.

Pivoting vs pivot table

In Pandas, the function `.pivot()` is a special case of `.pivot_table`. In `.pivot`, the categorical variables to create a new table or DataFrame is unique index. In other words, we cannot have duplicate entries for the same index values. See detailed examples later. The first a few sections below are about `.pivot`, and then `.pivot_table`. The later is similar to the pivot table in EXCEL/spreadsheet.

Pivoting a single variable

In the terms of spread-sheet, this is actually pivoting with one VALUE variable, which is 'visitors' in the below example. **Here 'a single variable' is not the ROW or COLUMN variable for creating pivot table, but refers to the VALUE variable. In the example below, ROW variable is just the index = 'weekday', and COLUMN variable is just 'city'.**

Note here we just display nicely a group by results. For example, for each 'coordinate' in pivot table such as (Mon, Dallas), corresponds to a group. The value in this coordinate is just the aggregating result for this group. In `.pivot()` case, no aggregation because is only one entry. In `.pivot_table`, then we have real aggregation.

```
In [1]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)

visitors_pivot = users.pivot(index='weekday', columns='city', values='visitors')
print(visitors_pivot)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

	city	Austin	Dallas
weekday			
Mon		326	456
Sun		139	237

Pivoting all variables

If you do not select any particular variables, all of them will be pivoted. In this case - with the users DataFrame - both 'visitors' and 'signups' will be pivoted, creating hierarchical column labels. **Here pivoting all variables** means in the same pivot table we use all other column names (except the column names for creating the pivot table) as the VALUE columns to aggregate upon. **'Pivoting all variables' does not mean we use all variables as ROW and COLUMN variables to create pivot table.**

```
In [16]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)

signups_pivot = users.pivot(index='weekday', columns='city', values='signups')

print(signups_pivot)

pivot = users.pivot(index='weekday', columns='city')

print(pivot)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

	city	Austin	Dallas
weekday			
Mon		3	5
Sun		7	12

		visitors		signups	
	city	Austin	Dallas	Austin	Dallas
weekday					
Mon		326	456	3	5
Sun		139	237	7	12

Stacking & unstacking I

In pivoting, we would like to display the data in a crossing view with ROW and COLUMN categorical variables set up respectively. However, if the two variables are already defined in a multi-index, then we need unstack to do so. (Note later in group by, it seems we can group by part of multi-index without unstacking it).

We pivot using a column name (e.g. city) earlier. **However, we cannot pivot using a column name which has been used in the multi-index, such as the 'weekday' in the following example.** In this case, we need move the multi-index column name ('weekday') to the normal column name. This is called unstacking. The opposite is called stacking. **So stacking and unstacking is usually applied for Dataframe index, particularly multi-Index?**

```
In [5]: #The above is for preparing data
import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
users = users.set_index(['city', 'weekday'])
print(users)
users = users.sort_index() #Only after sort index, then users will be same as DataCamp.
print(users)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

		visitors	signups
city	weekday		
Austin	Sun	139	7
Dallas	Sun	237	12
Austin	Mon	326	3
Dallas	Mon	456	5

		visitors	signups
city	weekday		
Austin	Mon	326	3
	Sun	139	7
Dallas	Mon	456	5
	Sun	237	12


```
In [6]: byweekday = users.unstack(level='weekday')
print(byweekday)
print(byweekday.stack(level='weekday'))
```

	visitors		signups	
weekday	Mon	Sun	Mon	Sun
city				
Austin	326	139	3	7
Dallas	456	237	5	12

	visitors		signups	
city	weekday			
Austin	Mon	326		3
	Sun	139		7
Dallas	Mon	456		5
	Sun	237		12

Stacking & unstacking II

Unstack and then stack the 'city' level, as did previously for 'weekday'. **Note that we cannot get the same DataFrame. The multi-index order is changed**

```
In [9]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
users = users.set_index(['city', 'weekday'])
users = users.sort_index()
#Only after sort index, then users will be same as DataCamp.
print(users)

bycity = users.unstack(level='city')
print(bycity)
print(bycity.stack(level='city'))
```

		visitors	signups
city	weekday		
Austin	Mon	326	3
	Sun	139	7
Dallas	Mon	456	5
	Sun	237	12

		visitors		signups	
city		Austin	Dallas	Austin	Dallas
weekday					
Mon		326	456	3	5
Sun		139	237	7	12

		visitors	signups
weekday	city		
Mon	Austin	326	3
	Dallas	456	5
Sun	Austin	139	7
	Dallas	237	12

Restoring the index order

use `swaplevel(0, 1)` to flip the index levels, `sort_index()`, and then obtain the original DataFrame.

```
In [10]: newusers = bycity.stack(level='city')
print(newusers)
newusers = newusers.swaplevel(0, 1)
print(newusers)
newusers = newusers.sort_index()
print(newusers)
print(newusers.equals(users))
```

		visitors	signups
weekday	city		
Mon	Austin	326	3
	Dallas	456	5
Sun	Austin	139	7
	Dallas	237	12

		visitors	signups
city	weekday		
Austin	Mon	326	3
Dallas	Mon	456	5
Austin	Sun	139	7
Dallas	Sun	237	12

		visitors	signups
city	weekday		
Austin	Mon	326	3
	Sun	139	7
Dallas	Mon	456	5
	Sun	237	12

True

Adding names for readability and DataFrame melting

melting DataFrames. melting is to restore a pivoted DataFrame to its original form, or to change it from a wide shape to a long shape.

```
In [17]: import pandas as pd
visitors_by_city_weekday = pd.read_csv("users_special.csv", index_col = 0) #This set the first column as index.
visitors_by_city_weekday.index.name = 'weekday'
#Without this is also fine. In the end, we can use id_vars = ['index'], the default name.
visitors_by_city_weekday.columns.name = 'city'
#Without this is also fine. In the end, the column name will be 'variable' after melting.
print(visitors_by_city_weekday)
print('-----')

visitors_by_city_weekday = visitors_by_city_weekday.reset_index()
#This make the original index column named 'weekday' become an ordinary column (no longer index)

print(visitors_by_city_weekday)
print('-----')

visitors = pd.melt(visitors_by_city_weekday, id_vars=['weekday'], value_name='visitors')

print(visitors)
```

```
city    Austin  Dallas
weekday
Mon         326     456
Sun         139     237
-----
None
city weekday  Austin  Dallas
0      Mon     326     456
1      Sun     139     237
-----
   weekday    city  visitors
0     Mon  Austin     326
1     Sun  Austin     139
2     Mon  Dallas     456
3     Sun  Dallas     237
```

Comments:

When pivoting the above table, we do the following:

```
visitors_pivot = users.pivot(index='weekday', columns='city', values='visitors')
```

index = 'weekday' is equivalent to setting ROW variable and columns = 'city' is equivalent to setting COLUMN variable in EXCEL.

The melting above is just doing the opposite.

Going from wide to long

move multiple columns into a single column (making the data long and skinny) by "melting" multiple columns.

Comments: pivoting goes from long to wide, and melting goes from wide to long. However, the number of cells in .pivoting is same due to unique index, while the number of cells in .pivot_table could be significantly reduced from an aggregate function.

```
In [27]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
print('-----')
# Melt users: skinny
skinny = pd.melt(users, id_vars=['weekday' , 'city'])

# Print skinny
print(skinny)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

```
-----
```

	weekday	city	variable	value
0	Sun	Austin	visitors	139
1	Sun	Dallas	visitors	237
2	Mon	Austin	visitors	326
3	Mon	Dallas	visitors	456
4	Sun	Austin	signups	7
5	Sun	Dallas	signups	12
6	Mon	Austin	signups	3
7	Mon	Dallas	signups	5

Obtaining key-value pairs with melt()

Sometimes, all you need is some key-value pairs, and the context does not matter. If said context is in the index, you can easily obtain what you want. For example, in the users DataFrame, the visitors and signups columns lend themselves well to being represented as key-value pairs. So if you created a hierarchical index with 'city' and 'weekday' columns as the index, you can easily extract key-value pairs for the 'visitors' and 'signups' columns by melting users and specifying col_level=0.

```
In [21]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
print('-----')

users_idx = users.set_index(['city', 'weekday'])
print(users_idx)
print('-----')

kv_pairs = pd.melt(users_idx, col_level=0)
print(kv_pairs)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

```
-----
```

		visitors	signups
city	weekday		
Austin	Sun	139	7
Dallas	Sun	237	12
Austin	Mon	326	3
Dallas	Mon	456	5

```
-----
```

	variable	value
0	visitors	139
1	visitors	237
2	visitors	326
3	visitors	456
4	signups	7
5	signups	12
6	signups	3
7	signups	5

Did not quite understand the above mechanism. Why col_level = 0

col_level : int or string, optional. If columns are a MultiIndex then use this level to melt.

Setting up a pivot table

As introduced earlier, `.pivot()` need **unique index column pairs** to identify values in the new table. If we don't have unique index columns, as in the example below, the 2nd and third row will give duplicate index when using response-gender multiIndex, then `.pivot` cannot identify which value (response) should have.

	id	response	gender	response
0	1	A	F	5
1	2	A	M	3
2	3	A	M	8
.....				

In this case, `.pivot` method will not work. `.pivot_table()` will be used instead, where we use aggregate function to reduct. **Comments: it seems `.pivot_table()` is more useful. Without huge amount of reduction with aggregation, `.pivot()` only change the shape of a table, but the size is still large, right?**

`Pivot_table` always need an aggregate function. **When it is not explicitly specified, its default mean or average is used.** In the following example, there is no duplicate entries, so `pivot_table` give the same results as `pivot`. This is when we use the default average (mean) aggregate function. If using other aggregate function, then they will not be same.

```
In [2]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
print('-----')
by_city_day = users.pivot_table(index='weekday', columns='city')

print(by_city_day)
print('-----')
print(by_city_day.index)
print('-----')
print(by_city_day.columns)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

```
-----
```

		signups		visitors	
	city	Austin	Dallas	Austin	Dallas
weekday					
	Mon	3	5	326	456
	Sun	7	12	139	237

```
-----
```

Index(['Mon', 'Sun'], dtype='object', name='weekday')

```
-----
```

MultiIndex(levels=[['signups', 'visitors'], ['Austin', 'Dallas']],
 codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
 names=[None, 'city'])

Using other aggregations in pivot tables

See pivot tables in EXCEL/spreadsheets, where aggregating function can count in many different ways.


```
In [3]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
print('-----')

count_by_weekday1 = users.pivot_table(index='weekday', aggfunc='count')
print(count_by_weekday1)
count_by_weekday2 = users.pivot_table(index='weekday', aggfunc=len)
print(count_by_weekday2)
print('=====')
print(count_by_weekday1.equals(count_by_weekday2))
```

```
   weekday  city  visitors  signups
0    Sun  Austin     139         7
1    Sun  Dallas     237        12
2    Mon  Austin     326         3
3    Mon  Dallas     456         5
-----
      city  signups  visitors
weekday
Mon         2         2         2
Sun         2         2         2
      city  signups  visitors
weekday
Mon         2         2         2
Sun         2         2         2
=====
True
```

Using margins in pivot tables

```
In [2]: import pandas as pd
users = pd.read_csv("users.csv", index_col = 0)
print(users)
print('-----')

signups_and_visitors = users.pivot_table(index='weekday', aggfunc=sum)
print(signups_and_visitors)
print('-----')
signups_and_visitors_total = users.pivot_table(index='weekday', aggfunc=sum, margins=True)
print(signups_and_visitors_total)
```

	weekday	city	visitors	signups
0	Sun	Austin	139	7
1	Sun	Dallas	237	12
2	Mon	Austin	326	3
3	Mon	Dallas	456	5

		signups	visitors
weekday			
Mon		8	782
Sun		19	376

		signups	visitors
weekday			
Mon		8	782
Sun		19	376
All		27	1158

Grouping data

- Comments: In fact the `pivot_table`, as detailed earlier, is also a GROUPING technique. See comments elsewhere.
- Here we still handle one of the fundamentals of data manipulation: grouping, and aggregating with built-in or self-defined functions.

Grouping by multiple columns

```
In [5]: import pandas as pd
titanic = pd.read_csv("titanic.csv")
# print(titanic.head())
# print('-----')

by_class = titanic.groupby('pclass')
# print(by_class.head(50))
print('-----')
#This is not the output like in sql. It is the df behind sql.

count_by_class = by_class['survived'].count()

print(count_by_class)
print('-----')

by_mult = titanic.groupby(['embarked', 'pclass'])

# print(by_mult.head(50))

count_mult = by_mult['survived'].count()

print(count_mult)
```

```
-----
pclass
1      323
2      277
3      709
Name: survived, dtype: int64
-----
embarked  pclass
C          1      141
           2       28
           3      101
Q          1        3
           2        7
           3      113
S          1      177
           2      242
           3      495
Name: survived, dtype: int64
```

Grouping by another series

Normally group by is followed by a column name (equivalently a categorical variable). Here by another series should be similar. In other words, this should be group by imagined categorical variable, e.g. region, with its unique values such as America, East Asia & Pacific, ...

```
In [ ]: life = pd.read_csv(life_fname, index_col='Country') #Two tables are set with the same index.
regions = pd.read_csv(regions_fname, index_col='Country')

life_by_region = life.groupby(regions['region'])
#No data available. Is it possible to group by the distinct names in column regions['region']?

print(life_by_region['2010'].mean())

# region
# America                74.037350
# East Asia & Pacific     73.405750
# Europe & Central Asia   75.656387
# Middle East & North Africa 72.805333
# South Asia             68.189750
# Sub-Saharan Africa     57.575080
# Name: 2010, dtype: float64
```

Computing multiple aggregates of multiple columns

- **Figure out the following that is inconsistent with SQL case: column names after select (here it is slicing in pandas) can be out of the column names after group by.** In SQL, we can aggregate upon different columns that is not after group by statement. But we cannot select these different columns in a SELECT statement. However, in Pandas, we can do this, though in a slightly different way.
- Like in pivot table, for the same subgroup, we can provide multiple aggregates.

```

In [8]: import pandas as pd
titanic = pd.read_csv("titanic.csv")
# print(len(titanic)) # 1309

df = titanic['pclass']==1
# print(len(titanic[df]))

by_class = titanic.groupby('pclass')

# print(by_class.head())

# Select 'age' and 'fare'
by_class_sub = by_class[['pclass', 'age', 'fare']] # IF in sql, we cannot groupby by 'pclass' but select 'age' and
print(by_class_sub.head())

# Aggregate by_class_sub by 'max' and 'median': aggregated
aggregated = by_class_sub.agg(['max', 'median'])

print(aggregated.head()) #I added this line and it shows a lot of information.
print('-----')
print(aggregated.loc[:, ('age', 'max')])
print('-----')
print(aggregated.loc[:, ('fare', 'median')])

```

	pclass	age	fare
0	1	29.00	211.3375
1	1	0.92	151.5500
2	1	2.00	151.5500
3	1	30.00	151.5500
4	1	25.00	151.5500
323	2	30.00	24.0000
324	2	28.00	24.0000
325	2	30.00	13.0000
326	2	18.00	11.5000
327	2	25.00	10.5000
600	3	42.00	7.5500
601	3	13.00	20.2500
602	3	16.00	20.2500
603	3	35.00	20.2500
604	3	16.00	7.6500

	pclass	age	fare
--	--------	-----	------

	max	median	max	median	max	median
pclass						
1	1	1	80.0	39.0	512.3292	60.0000
2	2	2	70.0	29.0	73.5000	15.0458
3	3	3	74.0	24.0	69.5500	8.0500

pclass

1 80.0

2 70.0

3 74.0

Name: (age, max), dtype: float64

pclass

1 60.0000

2 15.0458

3 8.0500

Name: (fare, median), dtype: float64

Aggregating on index levels/fields

- Creating multi-column index directly in the read_csv.
- Grouping by index levels and aggregating should be more efficient, as indexing is usually implemented by balanced trees or hash?

```

In [69]: gapminder = pd.read_csv('gapminder_tidy.csv', index_col=['Year', 'region', 'Country']).sort_index()

by_year_region = gapminder.groupby(level=['Year', 'region'])
# It seems we cannot pivoting with part of a multi-level index. But it is OK group by, right?

#print(by_year_region.head(n=50)) #This is very informative.

# Define the function to compute spread: spread
def spread(series):
    return series.max() - series.min()

# Create the dictionary: aggregator
aggregator = {'population': 'sum', 'child_mortality': 'mean', 'gdp': spread}
# Check earlier cells for providing multiple aggregate functions.
##Note the spread is a function.

aggregated = by_year_region.agg(aggregator)
print(aggregated.tail(6))
#This is like providing multiple aggregate functions in SQL statements.

```

Year	region	population	child_mortality	gdp
2013	America	9.629087e+08	17.745833	49634.0
	East Asia & Pacific	2.244209e+09	22.285714	134744.0
	Europe & Central Asia	8.968788e+08	9.831875	86418.0
	Middle East & North Africa	4.030504e+08	20.221500	128676.0
	South Asia	1.701241e+09	46.287500	11469.0
	Sub-Saharan Africa	9.205996e+08	76.944490	32035.0

Grouping on a function of the index

Groupby operations can also be performed on transformations of the index values.

Is there a day of the week that is more popular for customers? To find out, you're going to use `.strftime('%a')` to transform the index datetime values to abbreviated days of the week.

```
In [9]: sales = pd.read_csv('sales-feb-2015.csv', index_col='Date', parse_dates=True)

print(sales.head())
print('-----')

# Create a groupby object: by_day
print(sales.index.strftime('%a'))

print('-----')
by_day = sales.groupby(sales.index.strftime('%a'))
print(by_day.head())
# Here we cannot see the difference after printing.
# after transforming to week days, there must be repetitive ones. However, it seems groupby calculate the unique values
# and then group by, right?

# Here we group by day_of_the_week but not group by Monday, Tuesday....
print('-----')

# Create sum: units_sum
units_sum = by_day['Units'].sum()

# Print units_sum
print(units_sum)
```

Date	Company	Product	Units
2015-02-02 08:30:00	Hooli	Software	3
2015-02-02 21:00:00	Mediacore	Hardware	9
2015-02-03 14:00:00	Initech	Software	13
2015-02-04 15:30:00	Streeplex	Software	13
2015-02-04 22:00:00	Acme Coporation	Hardware	14

```
Index(['Mon', 'Mon', 'Tue', 'Wed', 'Wed', 'Thu', 'Thu', 'Sat', 'Mon', 'Mon',
      'Wed', 'Wed', 'Mon', 'Thu', 'Thu', 'Sat', 'Sat', 'Wed', 'Thu'],
      dtype='object')
```

Date	Company	Product	Units
2015-02-02 08:30:00	Hooli	Software	3
2015-02-02 21:00:00	Mediacore	Hardware	9
2015-02-03 14:00:00	Initech	Software	13

2015-02-04 15:30:00	Streeplex	Software	13
2015-02-04 22:00:00	Acme Coporation	Hardware	14
2015-02-05 02:00:00	Acme Coporation	Software	19
2015-02-05 22:00:00	Hooli	Service	10
2015-02-07 23:00:00	Acme Coporation	Hardware	1
2015-02-09 09:00:00	Streeplex	Service	19
2015-02-09 13:00:00	Mediacore	Software	7
2015-02-11 20:00:00	Initech	Software	7
2015-02-11 23:00:00	Hooli	Software	4
2015-02-16 12:00:00	Hooli	Software	10
2015-02-19 11:00:00	Mediacore	Hardware	16
2015-02-19 16:00:00	Mediacore	Service	10
2015-02-21 05:00:00	Mediacore	Software	3
2015-02-21 20:30:00	Hooli	Hardware	3
2015-02-25 00:30:00	Initech	Service	10
2015-02-26 09:00:00	Streeplex	Service	4

Mon 48

Sat 7

Thu 59

Tue 13

Wed 48

Name: Units, dtype: int64

Detecting outliers with Z-Scores

```
In [89]: import pandas as pd

gapminder = pd.read_csv("gapminder_tidy.csv", index_col = 'Country')

gapminder_2010 = gapminder[gapminder['Year'] == 2010]

gapminder_2010 = gapminder_2010.drop('Year', axis = 'columns') #axis = 'columns' is not axis = 'column'
#print(gapminder_2010.head())

from scipy.stats import zscore

standardized = gapminder_2010.groupby('region')['life', 'fertility'].transform(zscore)

outliers = (standardized['life'] < -3) | (standardized['fertility'] > 3)

gm_outliers = gapminder_2010.loc[outliers]

print(gm_outliers)
```

	fertility	life	population	child_mortality	gdp \
Country					
Guatemala	3.974	71.100	14388929.0	34.5	6849.0
Haiti	3.350	45.000	9993247.0	208.8	1518.0
Tajikistan	3.780	66.830	6878637.0	52.6	2110.0
Timor-Leste	6.237	65.952	1124355.0	63.8	1777.0

	region
Country	
Guatemala	America
Haiti	America
Tajikistan	Europe & Central Asia
Timor-Leste	East Asia & Pacific

Using z-scores like this is a great way to identify outliers in your data.

Filling missing data (imputation) by group

```
In [38]: import pandas as pd
titanic = pd.read_csv("titanic.csv")
by_sex_class = titanic.groupby(['sex', 'pclass'])

def impute_median(series):
    return series.fillna(series.median())
titanic.age = by_sex_class.age.transform(impute_median)
print(titanic.tail(5))
```

	pclass	survived	name	sex	age	sibsp	parch	\
1304	3	0	Zabour, Miss. Hileni	female	14.5	1	0	
1305	3	0	Zabour, Miss. Thamine	female	22.0	1	0	
1306	3	0	Zakarian, Mr. Mapriededer	male	26.5	0	0	
1307	3	0	Zakarian, Mr. Ortin	male	27.0	0	0	
1308	3	0	Zimmerman, Mr. Leo	male	29.0	0	0	

	ticket	fare	cabin	embarked	boat	body	home.dest
1304	2665	14.4542	NaN	C	NaN	328.0	NaN
1305	2665	14.4542	NaN	C	NaN	NaN	NaN
1306	2656	7.2250	NaN	C	NaN	304.0	NaN
1307	2670	7.2250	NaN	C	NaN	NaN	NaN
1308	315082	7.8750	NaN	S	NaN	NaN	NaN

Other transformations with .apply

.apply() is used to, for example, apply many functions to subgroups obtained from group by. See earlier cells about apply many aggregate functions with dictionary. Here .apply is more flexible.

```
In [39]: import pandas as pd
gapminder = pd.read_csv("gapminder_tidy.csv", index_col = 'Country')
gapminder_2010 = gapminder[gapminder['Year'] == 2010]
gapminder_2010 = gapminder_2010.drop('Year', axis = 'columns') #axis = 'columns' is not axis = 'column'
#print(gapminder_2010.head())

def disparity(gr):
    s = gr['gdp'].max() - gr['gdp'].min()
    # Compute the z-score
    z = (gr['gdp'] - gr['gdp'].mean())/gr['gdp'].std()
    return pd.DataFrame({'z(gdp)':z , 'regional spread(gdp)':s})

regional = gapminder_2010.groupby('region')
reg_disp = regional.apply(disparity)

print(reg_disp.loc[['United States', 'United Kingdom', 'China']])
```

	regional spread(gdp)	z(gdp)
Country		
United States	47855.0	3.013374
United Kingdom	89037.0	0.572873
China	96993.0	-0.432756

Grouping and filtering with .apply()

```
In [42]: import pandas as pd
titanic = pd.read_csv("titanic.csv")
print(len(titanic))

def c_deck_survival(gr):

    c_passengers = gr['cabin'].str.startswith('C').fillna(False)
    #After filling NaN with False, then the row will not be returned

    return gr.loc[c_passengers, 'survived'].mean()

by_sex = titanic.groupby('sex')

c_surv_by_sex = by_sex.apply(c_deck_survival)

print(c_surv_by_sex)
```

```
1309
sex
female    0.913043
male      0.312500
dtype: float64
```

Grouping and filtering with .filter()

Note the built-in python function filter() has two arguments. Here the .filter() defined on an object needs only one parameter.

```
In [43]: import pandas as pd
sales = pd.read_csv("sales-feb-2015.csv", index_col='Date', parse_dates=True)

by_company = sales.groupby('Company')
by_com_sum = by_company['Units'].sum()
print(by_com_sum)

# Filter 'Units' where the sum is > 35: by_com_filt
by_com_filt = by_company.filter(lambda g:g['Units'].sum() > 35)
print(by_com_filt)
```

```
Company
Acme Coporation    34
Hooli              30
Initech            30
Mediacore          45
Streeplex          36
Name: Units, dtype: int64
```

	Company	Product	Units
Date			
2015-02-02 21:00:00	Mediacore	Hardware	9
2015-02-04 15:30:00	Streeplex	Software	13
2015-02-09 09:00:00	Streeplex	Service	19
2015-02-09 13:00:00	Mediacore	Software	7
2015-02-19 11:00:00	Mediacore	Hardware	16
2015-02-19 16:00:00	Mediacore	Service	10
2015-02-21 05:00:00	Mediacore	Software	3
2015-02-26 09:00:00	Streeplex	Service	4

Filtering and grouping with .map()

```
In [40]: import pandas as pd
titanic = pd.read_csv("titanic.csv")

under10 = (titanic['age'] < 10).map({True:'under 10', False:'over 10'})
print(type(under10 ))
print('-----')
print(under10.head())
print('-----')

# See earlier cells about 'group by another series'
survived_mean_1 = titanic.groupby(under10)['survived'].mean()
print(survived_mean_1)

survived_mean_2 = titanic.groupby([under10, 'pclass'])['survived'].mean()
print(survived_mean_2)
```

```
<class 'pandas.core.series.Series'>
```

```
-----
0    over 10
1    under 10
2    under 10
3    over 10
4    over 10
Name: age, dtype: object
```

```
-----
age
over 10    0.366748
under 10    0.609756
Name: survived, dtype: float64
age      pclass
over 10  1      0.617555
          2      0.380392
          3      0.238897
under 10  1      0.750000
          2      1.000000
          3      0.446429
Name: survived, dtype: float64
```

