

Pandas 3 (Merging and Reshaping)

Sometimes, the way the data is stored in files or CSVs is not the way you need it for easy data processing. Often, one spends a lot of time "wrangling" the data into shape; the processing itself may even be easy in comparison.

A big plus point of Pandas is that it offers many tools to make the most common data wrangling operations relatively painless.

This lecture will look at some such problems:

- Merging multiple DataFrames,
- Transforming data using ranks and quantiles, and
- Reshaping the data via `pivot_table()`, `stack()` and `unstack()`

Merging DataFrames

Here's the setup: we have a bunch of DataFrames, each of which contains bits of the information we need. Ideally we want to combine them together into one DataFrame that we can easily work with.

We will use a dataset of a Question-Answering service where users ask questions about Pandas, which are answered by other users (data [here](#)).

```
In [1]: from pandas import Series, DataFrame
import pandas as pd
%pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```
In [2]: askers = pd.read_csv('Pandas_3_data/PandasUsers.csv')
questions = pd.read_csv('Pandas_3_data/PandasPosts.csv')
helpers = pd.read_csv('Pandas_3_data/PandasAnswerers.csv')
answers = pd.read_csv('Pandas_3_data/PandasAnswers.csv')
```

```
In [3]: askers.columns.values
```

```
Out[3]: array(['Id', 'Reputation', 'CreationDate', 'DisplayName',
       'LastAccessDate', 'WebsiteUrl', 'Location', 'AboutMe', 'Views',
       'UpVotes', 'DownVotes', 'EmailHash', 'Age'], dtype=object)
```

```
In [4]: questions.columns.values
```

```
Out[4]: array(['Id', 'PostTypeId', 'AcceptedAnswerId', 'CreationDate', 'Score',
       'ViewCount', 'Body', 'OwnerUserId', 'OwnerDisplayName',
       'LastEditorUserId', 'LastEditDate', 'LastActivityDate', 'Title',
       'Tags', 'AnswerCount', 'CommentCount', 'FavoriteCount',
       'ClosedDate'], dtype=object)
```

```
In [5]: helpers.columns.values
```

```
Out[5]: array(['Id', 'Reputation', 'CreationDate', 'DisplayName',
       'LastAccessDate', 'WebsiteUrl', 'Location', 'AboutMe', 'Views',
```

```
'UpVotes', 'DownVotes', 'EmailHash', 'Age'], dtype=object)
```

In [6]: `answers.columns.values`

Out[6]: `array(['Id', 'PostTypeId', 'ParentId', 'CreationDate', 'Score', 'Body', 'OwnerUserId', 'OwnerDisplayName', 'LastEditorUserId', 'LastEditorDisplayName', 'LastEditDate', 'LastActivityDate', 'CommentCount'], dtype=object)`

In [7]: `askers.iloc[0]`

Out[7]:

<code>Id</code>	<code>1843099</code>
<code>Reputation</code>	<code>100</code>
<code>CreationDate</code>	<code>2012-11-21 19:25:04</code>
<code>DisplayName</code>	<code>user1843099</code>
<code>LastAccessDate</code>	<code>2013-02-03 03:46:50</code>
<code>WebsiteUrl</code>	<code>NaN</code>
<code>Location</code>	<code>NaN</code>
<code>AboutMe</code>	<code>NaN</code>
<code>Views</code>	<code>0</code>
<code>UpVotes</code>	<code>16</code>
<code>DownVotes</code>	<code>0</code>
<code>EmailHash</code>	<code>17a9c54b937f8b505dcb3b0b5b34cbb6</code>
<code>Age</code>	<code>NaN</code>
<code>Name:</code>	<code>0</code>	<code>, dtype: object</code>

In [8]: `questions.iloc[0]`

Out[8]:

<code>Id</code>	<code>13446791</code>	
<code>PostTypeId</code>	<code>1</code>	
<code>AcceptedAnswerId</code>	<code>NaN</code>	
<code>CreationDate</code>	<code>2012-11-19 02:12:10</code>	
<code>Score</code>	<code>1</code>	
<code>ViewCount</code>	<code>74</code>	
<code>Body</code>	<code>I have a pandas dataframe with 2 columns (snip...</code>	
<code>OwnerUserId</code>	<code>1438637</code>	
<code>OwnerDisplayName</code>	<code>NaN</code>	
<code>LastEditorUserId</code>	<code>1438637.0</code>	
<code>LastEditDate</code>	<code>2012-11-19 21:41:54</code>	
<code>LastActivityDate</code>	<code>2012-11-19 21:41:54</code>	
<code>Title</code>	<code>Group Pandas dataframe based on highest occurr...</code>	
<code>Tags</code>	<code><python><pandas></code>	
<code>AnswerCount</code>	<code>1.0</code>	
<code>CommentCount</code>	<code>2.0</code>	
<code>FavoriteCount</code>	<code>NaN</code>	
<code>ClosedDate</code>	<code>NaN</code>	
<code>Name:</code>	<code>0</code>	<code>, dtype: object</code>	

Example: What questions did the first asker ask?

This is an example of combining data across multiple DataFrames. We must

- pick the asker ID from the askers DataFrame, and
- query for it in the questions DataFrame.

In [9]: `asker_id = askers.iloc[0]['Id']
asker_id`

Out[9]: `1843099`

In [10]:

```
questions_by_this_person = questions[questions['OwnerUserId'] == asker_id]
questions_by_this_person[['CreationDate', 'Body']]
```

	CreationDate	Body
65	2012-12-03 15:51:59	I have the following data:\n\nStart Time=2012...
70	2012-12-06 04:29:37	I try to read-in a file by pandas like this:\n...
79	2012-12-12 04:44:03	I read-in a file and plot it with pandas DataF...
96	2012-12-14 20:47:09	after plotting a figure I get a figure legend ...
106	2012-12-14 04:13:35	I have a temperature file with many years temp...
294	2013-01-24 23:51:14	I have a data file that includes several years...
332	2013-01-28 20:14:49	I have a DataFrame 'tso':\n\nntso=DataFrame(tem...

How many questions are asked on average by people with reputation >= 1000?

We have a similar problem:

- The 'reputation' exists in the askers DataFrame, while
- the number of questions can only be found in the questions DataFrame.

In [11]:

```
def is_reputed_asker(x):
    """Find all reputed asker IDs, and then check if x is among those IDs"""
    reputed_askers = askers[askers['Reputation'] >= 1000]
    ids_of_all_reputed_askers = reputed_askers['Id'].values
    return (x in ids_of_all_reputed_askers)

mask_questions_by_reputed_askers = questions['OwnerUserId'].map(is_reputed_asker)
questions[mask_questions_by_reputed_askers][:3]
```

Out[11]:

	Id	PostTypeId	AcceptedAnswerId	CreationDate	Score	ViewCount	Body	OwnerUserId	OwnerDisplayName	LastEditorUserId
30	13457335	1	NaN	2012-11-19 15:50:40	0	58	After seeing this question about replicating S...	567620	NaN	NaN
42	13675749	1	13708892.0	2012-12-03 00:58:53	2	147	I have inhomogeneous ~secondly data with a tim...	128508	NaN	1452002.0
51	13740672	1	13741439.0	2012-12-06 09:36:35	0	64	I'd like to filter out weekend data and only l...	24718	NaN	NaN

- Super messy.
- Hard to see the question and the asker's reputation all in one table

We need to **merge** the two DataFrames.

In [12]:

```
questions_askers = questions.merge(askers,
                                   left_on='OwnerUserId',
                                   right_on='Id',
                                   suffixes=['_question', '_asker'])
```

Let's analyze the previous statement closely. There are several arguments to the DataFrame.merge() function:

Which DataFrames?

We must first specify the DataFrames to be merged. The "left" DataFrame is questions, and the "right" DataFrame is askers.

```
questions_askers = questions.merge(askers,
                                   left_on='OwnerUserId',
                                   right_on='Id',
                                   suffixes=['_question', '_asker'])
```

How to merge?

In this case, we want the question owner (i.e., the person who asked the question) to be the same as the user in the askers DataFrame:

```
> question['OwnerUserId'] = askers['Id'].
```

That is, we want

- the 'OwnerUserId' field from the "left" DataFrame to match (left_on)
- the 'Id' column of the "right" DataFrame (right_on).

```
questions_askers = questions.merge(askers,
                                   left_on='OwnerUserId',
                                   right_on='Id',
                                   suffixes=['_question', '_asker'])
```

What about columns with the same names?

The questions DataFrame has an 'Id' column, as does the askers DataFrame. Which field do we keep in the merged DataFrame?

The suffixes argument says that in the merged DataFrame:

- the 'Id' from questions will be called 'Id_question'
- the 'Id' from askers will be called 'Id_asker'

In [13]: `questions_askers.columns.values`

Out[13]: `array(['Id_question', 'PostTypeId', 'AcceptedAnswerId',
 'CreationDate_question', 'Score', 'ViewCount', 'Body',
 'OwnerUserId', 'OwnerDisplayName', 'LastEditorUserId',
 'LastEditDate', 'LastActivityDate', 'Title', 'Tags', 'AnswerCount',
 'CommentCount', 'FavoriteCount', 'ClosedDate', 'Id_asker',
 'Reputation', 'CreationDate_asker', 'DisplayName',
 'LastAccessDate', 'WebsiteUrl', 'Location', 'AboutMe', 'Views',
 'UpVotes', 'DownVotes', 'EmailHash', 'Age'], dtype=object)`

What does the table look like?

In [14]: `mask_one_asker = (questions_askers['Id_asker'] == askers.iloc[0]['Id'])
questions_askers[mask_one_asker][['Id_asker', 'OwnerUserId', 'Body']]`

	<code>Id_asker</code>	<code>OwnerUserId</code>	<code>Body</code>
182	1843099	1843099	I have the following data:\n\nStart Time=2012...
183	1843099	1843099	I try to read-in a file by pandas like this:\n...
184	1843099	1843099	I read-in a file and plot it with pandas DataF...

ID_asker	OwnerUserId	Body
185	1843099	after plotting a figure I get a figure legend ...
186	1843099	I have a temperature file with many years temp...
187	1843099	I have a data file that includes several years...
188	1843099	I have a DataFrame 'tso':\nntso=DataFrame(tem...

- ID_asker is the same as OwnerUserId; that is because we *merged* on these two fields
- There was one ID_asker but 7 questions corresponding to him
 - In the merged table, there are 7 rows.

Example 1: Merge askers and questions based on ID_asker and QuestionOwnerId.

askers Table

ID_asker	Asker_city	Asker_name
1	NYC	John
2	SFO	John

questions Table

QuestionOwnerId	QuestionOwnerName	Question_data
1	John	Pandas problem
1	John	R question
2	John	what is the meaning of life

Answer: Merged Table has 3 rows

ID_asker	QuestionOwnerId	Asker_city	Asker_name	QuestionOwnerName	Question_data
1	1	NYC	John	John	Pandas problem
1	1	NYC	John	John	R question
2	2	SFO	John	John	what is the meaning of life

Example 2: Merge based on Asker_name and QuestionOwnerName.

askers Table

ID_asker	Asker_city	Asker_name
1	NYC	John
2	SFO	John

questions Table

QuestionOwnerId	QuestionOwnerName	Question_data
1	John	Pandas problem
1	John	R question
2	John	what is the meaning of life

Answer: Merged Table has 6 rows

ID_asker	QuestionOwnerId	Asker_city	Asker_name	QuestionOwnerName	Question_data
1	1	NYC	John	John	Pandas problem

ID_asker	QuestionOwnerID	Asker_city	Asker_name	QuestionOwnerName	Question_data
1	1	NYC	John	John	R question
1	2	NYC	John	John	what is the meaning of life
2	1	SFO	John	John	Pandas problem
2	1	SFO	John	John	R question
2	2	SFO	John	John	what is the meaning of life

How many questions are asked by people with reputation ≥ 1000 ?

How do we answer this?

- Mask by question asker's reputation
- Each row remaining corresponds to one question asked by a reputed asker
 - Count the number of such rows

In [15]:

```
# How many questions are asked by reputed askers?
mask = (questions_askers['Reputation'] >= 1000)
num_questions_reputed_askers = len(questions_askers[mask])
print('Number of questions by reputed askers =', num_questions_reputed_askers)
```

Number of questions by reputed askers = 119

How many askers have reputation ≥ 1000 ?

How do we answer this?

- By a mask, but on which table?
 - The questions_askers merged table?
 - The askers table?

In [16]:

```
# How many reputed askers are there overall?
mask = (askers['Reputation'] >= 1000)
num_reputed_askers = len(askers[mask])
print('Number of reputed askers =', num_reputed_askers)
```

Number of reputed askers = 59

Now we can answer our original question.

How many questions are asked on average by people with reputation ≥ 1000 ?

In [17]:

```
print('Average questions per reputed asker =', float(num_questions_reputed_askers) / num_reputed_askers)
```

Average questions per reputed asker = 2.016949152542373

We answered the question, but is 2.017 questions per asker high or low?

How many questions are asked on average by askers?

In [18]:

```
float(len(questions_askers)) / len(askers)
```

Out[18]: 1.9428571428571428

- Doesn't seem to be that much of a difference
- But then, we picked a reputation threshold of 1000 randomly
 - Later on, we'll ask: What is the right way?

Reshaping the data

- Often the data we get will be in *long form*, where each row represents one item
 - In the New York complaints dataset seen earlier, each row was one individual complaint.
- However, for analysis, the data is best represented in other ways.

Pandas makes this *reshaping* easy.

Let us revisit the NY complaints dataset. I have created a small CSV that contains only noise-related complaints.

```
In [19]: complaints = pd.read_csv('Pandas_3_data/311-requests-pivot.csv')
complaints.columns.values
```

```
Out[19]: array(['Complaint Type', 'Borough', 'Status', 'Agency', 'count'],
              dtype=object)
```

```
In [20]: complaints[:10]
```

```
Out[20]:
```

	Complaint Type	Borough	Status	Agency	count
0	Collection Truck Noise	BROOKLYN	Closed	DSNY	6
1	Collection Truck Noise	MANHATTAN	Closed	DSNY	5
2	Collection Truck Noise	QUEENS	Closed	DSNY	4
3	Collection Truck Noise	STATEN ISLAND	Closed	DSNY	1
4	Noise	BRONX	Closed	DEP	40
5	Noise	BRONX	Open	DEP	75
6	Noise	BROOKLYN	Closed	DEP	85
7	Noise	BROOKLYN	Open	DEP	442
8	Noise	BROOKLYN	Started	DEP	9
9	Noise	MANHATTAN	Closed	DEP	131

This is typically how a database will store the data.

How many complaints of each type occur in each Borough?

In Excel, we have pivot tables to do this. Pandas gives us the same power.

```
In [21]: pivoted = pd.pivot_table(complaints,
                             index='Complaint Type',
                             columns='Borough',
                             values='count',
                             aggfunc=sum)
pivoted
```

```
Out[21]:
```

Complaint Type	Borough	BRONX	BROOKLYN	MANHATTAN	QUEENS	STATEN ISLAND
Collection Truck Noise		NaN	6.0	5.0	4.0	1.0

	Borough	BRONX	BROOKLYN	MANHATTAN	QUEENS	STATEN ISLAND
Complaint Type						
Noise		115.0	536.0	1061.0	400.0	72.0
Noise - Commercial		163.0	698.0	1223.0	406.0	42.0
Noise - Helicopter		NaN	21.0	63.0	9.0	NaN
Noise - House of Worship		1.0	24.0	3.0	7.0	NaN
Noise - Park		2.0	NaN	17.0	11.0	NaN
Noise - Street/Sidewalk		109.0	208.0	609.0	100.0	19.0
Noise - Vehicle		53.0	183.0	228.0	75.0	18.0

Let us look at the pivot_table() function in more detail.

```
pivoted = pd.pivot_table(complaints,
                         index='Complaint Type',
                         columns='Borough',
                         values='count',
                         aggfunc=sum)
```

This creates a pivot table where

- the rows are all the possible complaint types,
- the columns are all possible values of the Boroughs, and
- the values are 'counts'.

But there is a problem.

Complaint Type	Borough	Agency	Status	count
Noise	BRONX	Closed	DEP	40
Noise	BRONX	Open	DEP	75

Since we are only looking at 'Complaint Type' and 'Borough', both count=40 and count=50 go to the same cell of the pivot table.

- the aggfunc (short for aggregator function) says hows to combine these values. aggfunc=sum says take the total number of noise complaints.

More complicated pivot tables are also possible.

```
In [22]: pivot_all = pd.pivot_table(complaints,
                               index=['Complaint Type', 'Borough'],
                               columns=['Agency', 'Status'],
                               values='count',
                               fill_value=0) # fill in missing values
pivot_all[:7]
```

Complaint Type	Borough	Agency		DEP		DSNY		EDC		NYPD	
		Status	Closed	Open	Started	Closed	Open	Assigned	Closed	Open	
Collection Truck Noise	BROOKLYN	0	0	0	6	0	0	0	0	0	
	MANHATTAN	0	0	0	5	0	0	0	0	0	
	QUEENS	0	0	0	4	0	0	0	0	0	
	STATEN ISLAND	0	0	0	1	0	0	0	0	0	
Noise	BRONX	40	75	0	0	0	0	0	0	0	

Complaint Type	Borough	Agency		DEP		DSNY		EDC		NYPD	
		Status	Closed	Open	Started	Closed	Open	Assigned	Closed	Open	
BROOKLYN		85	442	9	0	0	0	0	0	0	
MANHATTAN		131	928	2	0	0	0	0	0	0	

This yields a *hierarchical* index on both rows and columns.

Which agency is most responsible for each type of complaint?

In [23]:

```
complaint_agency = pd.pivot_table(complaints,
                                   index='Complaint Type',
                                   columns='Agency',
                                   values='count',
                                   aggfunc=sum,
                                   fill_value=0)

complaint_agency
```

Out[23]:

Complaint Type	Agency	DEP	DSNY	EDC	NYPD
Collection Truck Noise	0	16	0	0	0
Noise	2184	0	0	0	0
Noise - Commercial	0	0	0	2532	0
Noise - Helicopter	0	0	93	0	0
Noise - House of Worship	0	0	0	35	0
Noise - Park	0	0	0	30	0
Noise - Street/Sidewalk	0	0	0	1045	0
Noise - Vehicle	0	0	0	557	0

We now have a pivot table of agency versus noise complaint type. What do we do now?

The `idxmax()` method

In [24]:

```
complaint_agency.idxmax() # Oops
```

Out[24]:

```
Agency
DEP           Noise
DSNY      Collection Truck Noise
EDC        Noise - Helicopter
NYPD      Noise - Commercial
dtype: object
```

Oops! This gave the most common complaint for each agency; we want the most common agency for each complaint.

How do we get this?

We want to convert this to a matrix with the rows and columns *transposed*

```
complaint_agency.T
```

In [25]:

```
complaint_agency.T.idxmax()
```

Out[25]:

```
Complaint Type
Collection Truck Noise      DSNY
Noise                      DEP
```

```
Noise - Commercial ..... NYPD
Noise - Helicopter ..... EDC
Noise - House of Worship ..... NYPD
Noise - Park ..... NYPD
Noise - Street/Sidewalk ..... NYPD
Noise - Vehicle ..... NYPD
dtype: object
```



Summary so far

- pivot_table(index='X', columns='Y')
 - gives us a table where each row is an individual value from column X,
 - and each column is an individual value from column Y
- idxmax() picks the *index* corresponding to the maximum value for each column
 - which *value* of X is greatest for each *value* of Y

Ranks and Quantiles

Let's work with a concrete question.

Do those with greater reputation ask more questions, or less?

We have to do this in three steps:

1. Split askers by the percentile of their reputation, that is,
 - tag each asker by whether he/she is in the top-10%, top-20%, ... of all askers by reputation.
2. Merge this percentile information against questions
3. Count questions per asker for each percentile.

Step 1: Split askers by the percentile of their reputation

- If there are 1000 askers and we want 10 buckets
 - the top-100 ranked askers go into the top bucket
 - the next 100 ranked askers into the second bucket
 - and so on.

In [28]:

```
# Get percentiles via qcut
reputation_percentile = pd.qcut(askers['Reputation'],
                                10,
                                labels=['0-10', '10-20', '20-30',
                                         '30-40', '40-50', '50-60',
                                         '60-70', '70-80', '80-90', '90-100'])
reputation_percentile[:5]
```

Out[28]:

0	60-70
1	80-90
2	70-80
3	60-70
4	10-20

Name: Reputation, dtype: category

```
Categories (10, object): ['0-10' < '10-20' < '20-30' < '30-40' ... '60-70' < '70-80' < '80-90' < '90-100']
```

In [29]:

```
# Add this as a new column to askers
askers['Reputation Percentile'] = reputation_percentile
askers[['Reputation', 'Reputation Percentile']][:5]
```

Out[29]:

	Reputation	Reputation Percentile
0	100	60-70
1	766	80-90
2	472	70-80
3	188	60-70
4	13	10-20

We can check that each percentile has approximately the same number of askers.

In [30]:

```
askers_by_percentile = askers['Reputation Percentile'].value_counts()
askers_by_percentile
```

Out[30]:

0-10	55
20-30	53
60-70	49
70-80	49
80-90	49
90-100	49
40-50	48
50-60	48
30-40	47
10-20	43

Name: Reputation Percentile, dtype: int64

Step 2: Merge this percentile information against questions.

In [31]:

```
# Step 2: Merge this with questions
questions_askers = questions.merge(askers,
                                   left_on='OwnerUserId',
                                   right_on='Id',
                                   suffixes=['_question', '_asker'])
```

Step 3: Count questions per asker for each percentile.

In [32]:

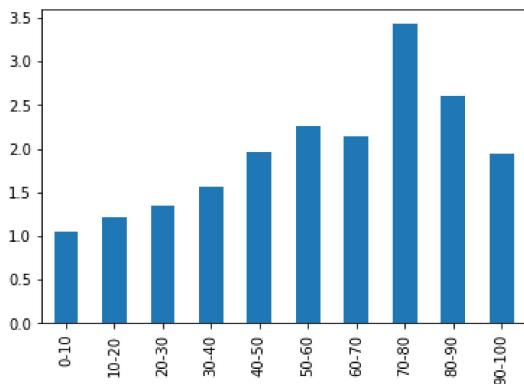
```
# Step 3: Get number of questions by Reputation Percentile.
questions_by_percentile = questions_askers['Reputation Percentile'].value_counts()
questions_per_asker_percentiles = questions_by_percentile / askers_by_percentile
```

In [33]:

```
questions_per_asker_percentiles.plot(kind='bar')
```

Out[33]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x134aa5b62c8>
```



Answer: Questions per asker is highest for the top 20-30% of askers by reputation.

- Why is it less for the less reputed? Maybe because they are less engaged.
- Why is it less for the more reputed? Maybe they are the experts, who answer more questions than they ask.

In [34]:

```
print(len(askers), 'askers')
print(len(questions), 'questions')
```

```
490 askers
952 questions
```

Caveat: It's a pretty small sample. Don't believe it too much...

Other reshaping methods

Apart from pivot_table(), there are two main ways to reshape the data:

- **stack**, which "rotates" or pivots from columns to index, and
- **unstack**, which does the opposite.

It's easier with examples.

In [37]:

```
pivoted
```

Out[37]:

	Borough	BRONX	BROOKLYN	MANHATTAN	QUEENS	STATEN ISLAND
Complaint Type						
Collection Truck Noise	NaN	6.0	5.0	4.0	1.0	
Noise	115.0	536.0	1061.0	400.0	72.0	
Noise - Commercial	163.0	698.0	1223.0	406.0	42.0	
Noise - Helicopter	NaN	21.0	63.0	9.0	NaN	
Noise - House of Worship	1.0	24.0	3.0	7.0	NaN	
Noise - Park	2.0	NaN	17.0	11.0	NaN	
Noise - Street/Sidewalk	109.0	208.0	609.0	100.0	19.0	
Noise - Vehicle	53.0	183.0	228.0	75.0	18.0	

Stack

In [38]:

```
stacked = pivoted.stack()
```

stacked[:10]

```
Out[38]: Complaint Type      Borough
Collection Truck Noise    BROOKLYN      6.0
                           MANHATTAN     5.0
                           QUEENS       4.0
                           STATEN ISLAND 1.0
Noise                   BRONX        115.0
                           BROOKLYN     536.0
                           MANHATTAN   1061.0
                           QUEENS      400.0
                           STATEN ISLAND 72.0
Noise - Commercial     BRONX       163.0
dtype: float64
```

We originally had:

- index = noise complaint type
- columns = borough

After stacking, the 'Borough' column becomes an extra level of the index; we get a *hierarchical* index.

- first level of index = noise complaint type
- second level of index = borough

Notice that the NaN values are smoothly hidden in the stacked view.

Unstack

Unstacking is the opposite of stacking; we move something from index to column. In a hierarchical index, we can choose what to send from index to column.

```
In [39]: stacked.unstack('Borough')[:3] # we get back the pivoted DataFrame
```

```
Out[39]:      Borough  BRONX  BROOKLYN  MANHATTAN  QUEENS  STATEN ISLAND
Complaint Type
Collection Truck Noise    NaN      6.0       5.0      4.0      1.0
Noise             115.0    536.0    1061.0    400.0     72.0
Noise - Commercial  163.0    698.0    1223.0    406.0     42.0
```

```
In [40]: stacked.unstack('Complaint Type')[:3] # we can also get it the other way
```

```
Out[40]:      Complaint Type  Collection Truck Noise  Noise - Commercial  Noise - Helicopter  Noise - House of Worship  Noise - Park  Noise - Street/Sidewalk  Noise - Vehicle
Borough
BRONX            NaN          115.0              163.0           NaN                 1.0            2.0            109.0          53.0
BROOKLYN         6.0          536.0              698.0            21.0               24.0           NaN            208.0          183.0
MANHATTAN        5.0          1061.0             1223.0            63.0                3.0            17.0            609.0          228.0
```

Which type of noise complaint is most prevalent in each Borough?

```
In [41]: pivoted[:5]
```

Out[41]:

	Borough	BRONX	BROOKLYN	MANHATTAN	QUEENS	STATEN ISLAND
	Complaint Type					
Collection Truck Noise		NaN	6.0	5.0	4.0	1.0
Noise		115.0	536.0	1061.0	400.0	72.0
Noise - Commercial		163.0	698.0	1223.0	406.0	42.0
Noise - Helicopter		NaN	21.0	63.0	9.0	NaN
Noise - House of Worship		1.0	24.0	3.0	7.0	NaN

If we only wanted the most prevalent complaint for one Borough (say, Bronx), what would we do?

In [42]:

```
# Getting the max value or index for a Series is easy.
print('Most prevalent complaint for BRONX is', pivoted['BRONX'].idxmax())
print('which occurred', pivoted['BRONX'].max(), 'times.')
```

Most prevalent complaint for BRONX is Noise - Commercial
which occurred 163.0 times.

The same *idxmax()* function works for DataFrames as well

In [43]:

```
# For a DataFrame, the same thing works.
pivoted.idxmax()
```

Out[43]:

```
Borough
BRONX           Noise - Commercial
BROOKLYN        Noise - Commercial
MANHATTAN        Noise - Commercial
QUEENS          Noise - Commercial
STATEN ISLAND    Noise
dtype: object
```

In [44]:

```
# Similarly, getting the max value is easy.
pivoted.max()
```

Out[44]:

```
Borough
BRONX           163.0
BROOKLYN        698.0
MANHATTAN        1223.0
QUEENS          406.0
STATEN ISLAND    72.0
dtype: float64
```

In [45]:

```
# How do we get both together?
most_common = DataFrame({'Most common complaint':pivoted.idxmax(), 'Num complaints':pivoted.max()})
most_common
```

Out[45]:

Borough	Most common complaint	Num complaints
BRONX	Noise - Commercial	163.0
BROOKLYN	Noise - Commercial	698.0
MANHATTAN	Noise - Commercial	1223.0
QUEENS	Noise - Commercial	406.0
STATEN ISLAND	Noise	72.0

Summary

- **merge** combines multiple DataFrames into one

- Useful when you want to ask questions whose answers lie in different DataFrames
- **rank**, which ranks the values in a Series
- **qcut**, which splits the values of a Series into percentiles
- **pivot_table**, which gives a break-up according to *values* of columns in a DataFrame