**Pivoting a single variable**

Suppose you started a blog for a band, and you would like to log how many visitors you have had, and how many signed-up for your newsletter. To help design the tours later, you track where the visitors are. Create a DataFrame called users from users.csv file.

Inspect users in the IPython Shell and make a note of which variable you want to use to index the rows ('weekday'), which variable you want to use to index the columns ('city'), and which variable will populate the values in the cells ('visitors'). Try to visualize what the result should be.

For example, I have used 'treatment' to index the rows, 'gender' to index the columns, and 'response' to populate the cells. Prior to pivoting, the DataFrame looked like this:

id treatment gender response

0 1 A F 5

1 2 A M 3

2 3 B F 8

3 4 B M 9

After pivoting:

gender F M

treatment

A 5 3

B 8 9

In this exercise, your job is to pivot users so that the focus is on 'visitors', with the columns indexed by 'city' and the rows indexed by 'weekday'.

##### INSTRUCTIONS

* Pivot the users DataFrame with the rows indexed by 'weekday', the columns indexed by 'city', and the values populated with 'visitors'.
* Print the pivoted DataFrame.

# 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.

You will explore this for yourself now in this exercise.

##### INSTRUCTIONS

* Pivot the users DataFrame with the 'signups' indexed by 'weekday' in the rows and 'city' in the columns.
* Print the new DataFrame.
* Pivot the users DataFrame with both 'signups' and 'visitors' pivoted - that is, all the variables. This will happen automatically if you do not specify an argument for the values parameter of .pivot().
* Print the pivoted DataFrame.

# Stacking & unstacking I

You are now going to practice stacking and unstacking DataFrames. Use yhe users DataFrame you have been working this time with a MultiIndex. Explore it in the IPython Shell to see the data layout. Pay attention to the index, and notice that the index levels are ['city', 'weekday']. So 'weekday' - the second entry - has position 1. This position is what corresponds to the level parameter in .stack() and .unstack() calls. Alternatively, you can specify 'weekday' as the level instead of its position.

Your job in this exercise is to unstack users by 'weekday'. You will then use .stack() on the unstacked DataFrame to see if you get back the original layout of users.

##### INSTRUCTIONS

* Define a DataFrame byweekday with the 'weekday' level of users unstacked.
* Print the byweekday DataFrame to see the new data layout.
* Stack byweekday by 'weekday' and print it to check if you get the same layout as the original users DataFrame.

# Stacking & unstacking II

You are now going to continue working with the users DataFrame. As always, first explore it in the IPython Shell to see the layout and note the index.

Your job in this exercise is to unstack and then stack the 'city'level, as you did previously for 'weekday'. Note that you won't get the same DataFrame.

##### INSTRUCTIONS

* Define a DataFrame bycity with the 'city' level of users unstacked.
* Print the bycity DataFrame to see the new data layout..
* Stack bycity by 'city' and print it to check if you get the same layout as the original users DataFrame.

# Restoring the index order

Continuing from the previous exercise, you will now use .swaplevel(0, 1) to flip the index levels. Note they won't be sorted. To sort them, you will have to follow up with a .sort\_index(). You will then obtain the original DataFrame. Note that an unsorted index leads to slicing failures.

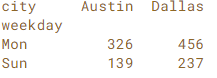
To begin, print both users and bycity in the IPython Shell. The goal here is to convert bycity back to something that looks like users.

##### INSTRUCTIONS

* Define a DataFrame newusers with the 'city' level stacked back into the index of bycity.
* Swap the levels of the index of newusers.
* Print newusers and verify that the index is not sorted.
* Sort the index of newusers.
* Print newusers and verify that the index is now sorted. This has been done for you.
* Assert that newusers equals users.

# Adding names for readability

You are now going to practice melting DataFrames. Create a DataFrame called visitors\_by\_city\_weekday referring the below snapshot



Explore it in the IPython Shell and see that it is the usersDataFrame from previous exercises with the rows indexed by 'weekday', columns indexed by 'city', and values populated with 'visitors'.

The goal of melting is to restore a pivoted DataFrame to its original form, or to change it from a wide shape to a long shape. You can explicitly specify the columns that should remain in the reshaped DataFrame with id\_vars, and list which columns to convert into values with value\_vars.. If you don't pass a name to the values in pd.melt(), you will lose the name of your variable. You can fix this by using the value\_name keyword argument.

Your job in this exercise is to melt visitors\_by\_city\_weekdayto move the city names from the column labels to values in a single column called 'city'. If you were to use just pd.melt(visitors\_by\_city\_weekday), you would obtain the following result:

city value

0 weekday Mon

1 weekday Sun

2 Austin 326

3 Austin 139

4 Dallas 456

5 Dallas 237

Therefore, you have to specify the id\_vars keyword argument to ensure that 'weekday' is retained in the reshaped DataFrame, and the value\_name keyword argument to change the name of value to visitors.

##### INSTRUCTIONS

* Reset the index of visitors\_by\_city\_weekday with .reset\_index().
* Print visitors\_by\_city\_weekday and verify that you have just a range index, 0, 1, 2, 3..
* Melt visitors\_by\_city\_weekday to move the city names from the column labels to values in a single column called city.
* Print visitors to check that the city values are in a single column now and that the dataframe is longer and skinnier.

# Going from wide to long

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

Use the users DataFrame

##### INSTRUCTIONS

* Define a DataFrame skinny where you melt the 'visitors'and 'signups' columns of users into a single column.
* Print skinny to verify the results. Note the value column that had the cell values in users.

# 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.

##### INSTRUCTIONS

* Set the index of users to ['city', 'weekday'].
* Print the DataFrame users\_idx to see the new index.
* Obtain the key-value pairs corresponding to visitors and signups by melting users\_idx with the keyword argument col\_level=0

# Setting up a pivot table

A pivot table allows you to see all of your variables as a function of two other variables. In this exercise, you will use the .pivot\_table() method to see how the users DataFrame entries appear when presented as functions of the 'weekday' and 'city' columns. That is, with the rows indexed by 'weekday' and the columns indexed by 'city'.

Before using the pivot table, print the users DataFrame in the IPython Shell and observe the layout.

##### INSTRUCTIONS

* Use a pivot table to index the rows of users by 'weekday'and the columns of users by 'city'. These correspond to the index and columns parameters of .pivot\_table().
* Print by\_city\_day

# Using other aggregations in pivot tables

You can also use aggregation functions with in a pivot table by specifying the aggfunc parameter. In this exercise, you will practice using the 'count' and len aggregation functions - which produce the same result - on the users DataFrame.

##### INSTRUCTIONS

* Define a DataFrame count\_by\_weekday1 that shows the count of each column with the parameter aggfunc='count'. The index here is 'weekday'.
* Print count\_by\_weekday1. This has been done for you.
* Replace aggfunc='count' with aggfunc=len and verify you obtain the same result.

# Using margins in pivot tables

Sometimes it's useful to add totals in the margins of a pivot table. You can do this with the argument margins=True. In this exercise, you will practice using margins in a pivot table along with a new aggregation function: sum.

Use the users DataFrame

##### INSTRUCTIONS

* Define a DataFrame signups\_and\_visitors that shows the breakdown of signups and visitors by day, as well as the totals.
  + You will need to use aggfunc=sum to do this.
* Print signups\_and\_visitors
* Now pass the additional argument margins=True to the .pivot\_table() method to obtain the totals.
* Print signups\_and\_visitors\_total.