Ch. 8 Data Wrangling: Joining, Combine, and Reshape pg. 221 - 251

This chapted focuses on tools to help combine, join, and rearrange data.

In [32]:
```
import pandas as pd
import numpy as np
```

8.1 Hierarchical Indexing

This is an important feature of pandas that allows you to have multiple index *levels* on an axis. It allows for you to work with higher dimensional data in a lower dimensional form.
Starting with an example, first by creating a Series with lists of lists as the index:

In [45]:
```
data = pd.Series(np.random.randn(9),
                 index = [['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
                          [1, 2, 3, 1, 3, 1, 2, 2, 3]])
data

# This shows a Series with a MultiIndex as its index.
# The gaps in the index display mean "use the label directly above"
```

Out[45]:
```
a  1    0.180205
   2    0.414830
   3    0.804819
b  1   -0.693624
   3   -0.232748
c  1    0.224456
   2   -0.776500
d  2   -0.625619
   3   -0.748366
dtype: float64
```

[?] Why does my output look different than whats on pg. 222?
https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html (https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html)

In [46]:
```
data.index
```

Out[46]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

In [47]:
```
# You can also 'partically index' with hieratchicall indexing, to select subse
ts of data.
data['b']
```

Out[47]:
```
1   -0.693624
3   -0.232748
dtype: float64
```

In [48]:
```
data['b':'c']
```

Out[48]:
```
b  1   -0.693624
   3   -0.232748
c  1    0.224456
   2   -0.776500
dtype: float64
```

In [54]:
```
# Can also select from an "inner level" aka within the hierarchical index
data.loc[:, 2]  # Give me all the values within any index sith the index of 2
```

Out[54]:
```
a    0.414830
c   -0.776500
d   -0.625619
dtype: float64
```

Hieratchical indexing is important for reshaping data and group-based observation

You can also, e.g., rearrange data into a DF using its 'unstack' method.

In [55]:
```
data.unstack()
```

Out[55]:

	1	2	3
a	0.180205	0.414830	0.804819
b	-0.693624	NaN	-0.232748
c	0.224456	-0.776500	NaN
d	NaN	-0.625619	-0.748366

In [56]:
```
# The inverse operation of the aboce is 'stack':
data.unstack().stack()
```

Out[56]:
```
a  1    0.180205
   2    0.414830
   3    0.804819
b  1   -0.693624
   3   -0.232748
c  1    0.224456
   2   -0.776500
d  2   -0.625619
   3   -0.748366
dtype: float64
```

In [60]:
```
# With a Datafram, either axis can have a hierarchical index:
frame = pd.DataFrame(np.arange(12).reshape((4,3)),
                     index = [['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                     columns = [['Ohio', 'Ohio', 'Colorado'],
                                ['Green', 'Red', 'Green']])
frame
```

Out[60]:

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [61]:
```
# The hierarchical levels can have names. If so, they will show up in the cons
ole output
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']

frame
#  three layers of indexes

## NOTE: Be careful to distinguish the index names 'state' and 'color' from th
e rows!
```

Out[61]:

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [62]:
```
# With partial column index, you can select groups of columns:
frame['Ohio']
```

Out[62]:

	color	Green	Red
key1	key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

In [63]:
```
# A 'MultiIndex' can be created by itself and then reused.
# The columns in the preceding DF w. level names could be created like this as
well:

MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green'
]],
                       names = ['state', 'color'])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-63-a1fa2e1c1fec> in <module>
      2 # The columns in the preceding DF w. level names could be created lik
e this as well:
      3
----> 4 MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Re
d', 'Green']],
      5                       names = ['state', 'color'])

NameError: name 'MultiIndex' is not defined
```

Recording and Sorting Levels

You will at times need to rearrange the order of the levels on a axis or sort the data by values of just 1 specific level. The **'swaplevel'**</u> takes two level numbers or names and returns a new object with the levels interchanged (but the data itself is otherwise unaltered).

In [ ]:
```
frame.swaplevel('key1', 'key2')
```

'sort_index' on the other hand, sorts the data using only the values in a single level.

When swapping levels, it is not uncommon to also use sort_index so that the result is lexicopgraphically sorted by the indicated lvl.

In [64]:
```
frame.sort_index(level = 1) # sort by 2nd lvl of indexes, aka the 1, 1, 2,2

#To remember the order of the indexes, refer to previous chunk with Frame, whe
n it was first defined
```

Out[64]:

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

In [65]:
```
frame.sort_index(level = 0) # sort by le column values in the first level of i
ndex, aka the a & b
```

Out[65]:

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [66]:
```
frame.swaplevel(0, 1).sort_index(level = 0) # Switch Key 1 and 2, in terms of
 how they appear in the DataFrame
```

Out[66]:

	state	Ohio		Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

Note: Data select performance goes much quicker/efficiently on hierarchically indexed objects if the index is lexicopgrahically sorted starting with the outermost level-

Aka, the output of calling ' sort_index(level = 0) OR sort_index()

Summary Statistics by Level

Most of the summary stats on DFs and series have a 'level' option, where you specify the level of the index you want to aggregate by on a particular axis. See below for example:

In [ ]:
```
# With the Frame, we can aggregate by lvl on either the rows or columns:
# row-wise sum

frame.sum(level = 'key2')
```

In [ ]:
```
# Column-wise sum
frame.sum(level = 'color', axis = 1)


# Both of these functions/methods are actually using panda's groupby machinery
```

Indexing with a DataFrame's columns

> It's not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame's columns. Here's an example DataFrame: Usually used for time series, makes it faster to use this col of the date as an index

In [71]:
```
frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
                      'c': ['one', 'one', 'one', 'two', 'two',
                            'two', 'two'],
                      'd': [0, 1, 2, 0, 1, 2, 3]})
frame
```

Out[71]:

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

In [72]:
```
# Using DFs 'set_index' will create a new DF using 1 or more of its cols as the index
frame2 = frame.set_index(['c', 'd'])
frame2
```

Out[72]:

		a	b
c	d		
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

[!] Try to think of an example where you would want to do the above/or below

In [70]:
```
# By default, the cols are removed from the DF, though you CAN leave them in
frame.set_index(['c', 'd'], drop = False)
```

Out[70]:

		a	b	c	d
c	d				
one	0	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	0	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

[?] What does the below mean? WTF does reset_index do and how is it different than the original DateFrame?

In [ ]:
```
# Using 'reset_index' does the OPPOSITE of 'set_index'
# Here, the hierarchical index levels are moved INTO the columns
frame2.reset_index()
```

In [ ]:
```
# Compare the above with 'frame'
frame
```

8.2 Combining and Merging Data Sets

- pandas.merge connects rows in a DF based on 1 or more keys. Similar to *join* in SQL.
- pandas.concat concatenates or "stacks" together objects along an axis.
- combine_first instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

Might also be helpful

Real Python - pandas: merge, join, and concat (https://realpython.com/pandas-merge-join-and-concat/)

Database-Style DataFrame Joins

In [73]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                    'data1': range(7)})
df1
```

Out[73]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [78]:
```
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], #has only unique values, the other does noT. So many to one joi
                    'data2': range(3)})
df2
```

Out[78]:

	key	data2
0	a	0
1	b	1
2	d	2

* Read more about different joints (many to one, or many to many) here:

https://fmhelp.filemaker.com/help/18/fmp/en/index.html#page/FMP_Help/one-to-many-relationships.html

Many-to-one join

In [84]:
```
#Merge is an inner join, and an inner join means only show the common rows
```

In [85]:
```
# An example of many-to-one join: where df1 has multiple rows labeled b and df2 only has 1 row for each value in the key column.
# Calling 'merge' with these objects gives the following:

pd.merge(df1, df2)  # It will add 0s where the table/keys do not align
```

Out[85]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

In [86]:
```
# Note, in the above, which col to join to. If not specified, 'merge' will use the overlapping col names as the keys.
# otherwise, use 'on' to specify

pd.merge(df1, df2, on = 'key') # Output is the same, but it is good habit to specify where to join on.
```

Out[86]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

In [87]:
```
# If the col names are different in each object, you can specify them separately
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                    'data1': range(7)})
df3
```

Out[87]:

	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [88]:
```
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
                    'data2': range(3)})
df4
```

Out[88]:

	rkey	data2
0	a	0
1	b	1
2	d	2

In [89]:
```
pd.merge(df3, df4, left_on = 'lkey', right_on = 'rkey')
```

Out[89]:

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

*In the above merge with df3 and df4, notice that the 'c' and 'd' values are missing. This is because merg does an **inner join; the keys in the result/output are the intersection, or commone set found in both tables.***

Other possible options are 'left', 'right', and 'outer'. The outer join takes the union of the keys, combining the effect of applying both left and right joins.

In [90]:
```
# To do an outer join, and include the keys that are NOT found in both tables.

pd.merge(df1, df2, how = 'outer')
```

Out[90]:

	key	data1	data2
0	b	0.0	1.0
1	b	1.0	1.0
2	b	6.0	1.0
3	a	2.0	0.0
4	a	4.0	0.0
5	a	5.0	0.0
6	c	3.0	NaN
7	d	NaN	2.0

Left join means it will only contain all of the keys from the left data fame. Outer join means it will be a UNION and include ALL keys from both dfs.

See table 8.1 p.229 for a summary of options for 'how' for join types.

Many-to-many join

In [91]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                    'data1': range(6)})
df1
```

Out[91]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [92]:
```
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
                    'data2': range(5)})
df2
```

Out[92]:

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

In [93]:
```
pd.merge(df1, df2, on = 'key', how = 'left')
```

Out[93]:

	key	data1	data2
0	b	0	1.0
1	b	0	3.0
2	b	1	1.0
3	b	1	3.0
4	a	2	0.0
5	a	2	2.0
6	c	3	NaN
7	a	4	0.0
8	a	4	2.0
9	b	5	1.0
10	b	5	3.0

Many-to-many joins form the Cartesian product of the rows. Since there were three 'b' rows in the left DataFrame and two in the right one, there are six 'b' rows in the result. The join method only affects the distinct key values appearing in the result: (see below) pg. 230

In [94]:
```
pd.merge(df1, df2, how='inner') #Remember, inner joins, the standard, do not include any keys that are not found in BOTH
```

Out[94]:

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	b	5	1
5	b	5	3
6	a	2	0
7	a	2	2
8	a	4	0
9	a	4	2

To merge with multiple keys, pass a list of column names

In [95]:
```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
                     'key2': ['one', 'two', 'one'],
                     'lval': [1, 2, 3]})
```

In [96]:
```
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                      'key2': ['one', 'one', 'one', 'two'],
                      'rval': [4, 5, 6, 7]})
```

In [97]:
```
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

Out[97]:

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the earlier section on renaming axis labels), merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects: pg. 231

In [ ]:
```
pd.merge(left, right, on='key1')
```

In [ ]:
```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Merging on Index

> In some cases, the merge key(s) in a DataFrame will be found in its index. In this case, you can pass left_index=True or right_index=True (or both) to indicate that the index should be used as the merge key:

In [98]:
```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
                      'value': range(6)})
left1
```

Out[98]:

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

In [99]:
```
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
right1
```

Out[99]:

	group_val
a	3.5
b	7.0

In [100]:
```
pd.merge(left1, right1, left_on='key', right_index=True)
```

Out[100]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

In [101]:
```
# With hierarchically indexed data, things are more complicated, as joining on
index is implicitly a multiple-key merge:
lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
                               'Nevada', 'Nevada'],
                      'key2': [2000, 2001, 2002, 2001, 2002],
                      'data': np.arange(5.)})
lefth
```

Out[101]:

	key1	key2	data
0	Ohio	2000	0.0
1	Ohio	2001	1.0
2	Ohio	2002	2.0
3	Nevada	2001	3.0
4	Nevada	2002	4.0

In [102]:
```
righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
                      index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
                              'Ohio', 'Ohio'],
                             [2001, 2000, 2000, 2000, 2001, 2002]],
                      columns=['event1', 'event2'])
righth
```

Out[102]:

		event1	event2
Nevada	2001	0	1
	2000	2	3
Ohio	2000	4	5
	2000	6	7
	2001	8	9
	2002	10	11

In [103]:
```
# With hierarch. indexing, , you have to indicate multiple columns to merge on
as a list
pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)

# This one, aka joining INNER, WILL NOT print rows with any missing data
```

Out[103]:

	key1	key2	data	event1	event2
0	Ohio	2000	0.0	4	5
0	Ohio	2000	0.0	6	7
1	Ohio	2001	1.0	8	9
2	Ohio	2002	2.0	10	11
3	Nevada	2001	3.0	0	1

In [104]:
```
# (note the handling of duplicate index values with how='outer'):
pd.merge(lefth, righth, left_on=['key1', 'key2'],
         right_index=True, how='outer') # This DOES show rows with missing dat
a
```

Out[104]:

	key1	key2	data	event1	event2
0	Ohio	2000	0.0	4.0	5.0
0	Ohio	2000	0.0	6.0	7.0
1	Ohio	2001	1.0	8.0	9.0
2	Ohio	2002	2.0	10.0	11.0
3	Nevada	2001	3.0	0.0	1.0
4	Nevada	2002	4.0	NaN	NaN
4	Nevada	2000	NaN	2.0	3.0

Using the indexes of both sides of the merge is also possible:

In [105]:
```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                     index=['a', 'c', 'e'],
                     columns=['Ohio', 'Nevada'])
left2
```

Out[105]:

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

In [106]:
```
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                      index=['b', 'c', 'd', 'e'],
                      columns=['Missouri', 'Alabama'])
right2
```

Out[106]:

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

In [107]:
```
pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

Out[107]:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

In [108]:
```
# Can also use 'join' for mergin by index, but not for overlapping columns.
left2.join(right2, how = 'outer') # Is the same as the previous line, with pd
 Merge.
```

Out[108]:

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

join() does a left join by default

In [ ]:
```
left1.join(right1, on = 'key')
```

In [ ]:
```
# Simple index-on-index merger, can pass a list of DF to 'join' as an alternat
ive to use the more general  'concat' function as shown in the next section
another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
                       index=['a', 'c', 'e', 'f'],
                       columns=['New York', 'Oregon'])
another
```

In [ ]:
```
left2.join([right2, another])
```

In [44]:
```
left2.join([right2, another], how = 'outer')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-44-9d7e8ff68cce> in <module>
----> 1 left2.join([right2, another], how = 'outer')

NameError: name 'left2' is not defined
```

Concatenating Along an Axis

Aka adding ROWS together by default, and you will get a series.
If you concatnate by columns you will get a DF.

In [ ]: _

Combining Data with Overlap

In [ ]: _

8.3 Reshaping and Pivoting

Reshaping with Hierarchical Indexing

In [ ]: _

Pivoting "Long" to "Wide" Format

In [ ]: _

Pivoting "Wide" to "Long" Format

In [ ]: _

In [ ]: _

8.4 Conclusion

In [ ]: _