



DAAN862: ANALYTICS PROGRAMMING IN PYTHON

Lesson 5: Data Wrangling with Pandas

Lesson 5: Objectives and Overview (1 of 5)

Lesson 5: Data Wrangling with Pandas

In many cases, data may not be arranged in a clean DataFrame like format or stored in different files. pandas provides various functions to solve these tasks.

At the end of this lesson, using pandas, the student will be able to:

Construct and modify hierarchical indexes of Series and DataFrame

Combine and join two datasets

Perform data reshaping and pivoting

By the end of this lesson, please complete all readings and assignments found in the [Lesson 5 Course Schedule](#).

Lesson 5.1: Hierarchical Indexing (2 of 5)

Lesson 5.1: Hierarchical Indexing

Hierarchical indexing is a useful feature in pandas which enables multiple index levels on both axes (column and row). It provides a way to handle higher dimensional data with a lower dimensional format.

Please refer to Figure 5.1 for an example of a *Series* with a two lists as the index:



Fig
5.1

The output is a nice view of a Series with a two-level index. The outer and the inner levels are indicated in the Figure 5.1. The “gaps” in the outer-level index mean “use the index label above”.

To access the *index attribute* refer to Figure 5.2:

```
In [4]: data.index
Out[4]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
            labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

Fig 5.2 (click to enlarge)

Figure 5.2 shows how to return Multiindex with both levels and labels the information.

If you are using hierarchical indexing, you can easily use an outer-level index to select subsets of the data as shown in Figure 5.3:

```
In [5]: data['b']
Out[5]:
1    -0.562908
3    -1.064038
dtype: float64

In [6]: data['b' : 'c']
Out[6]:
b 1    -0.562908
   3    -1.064038
c 1    -0.464033
   2     0.532145
dtype: float64

In [7]: data.loc[['b', 'd']]
Out[7]:
b 1    -0.562908
   3    -1.064038
d 2     0.962120
   3    -1.193371
dtype: float64
```

Fig 5.3 (click to enlarge)

To select subsets from an "inner" level refer to Figure 5.4:

```
In [8]: data.loc[:, 2]
Out[8]:
a    -0.910733
c     0.532145
d     0.962120
dtype: float64
```

Fig 5.4 (click to enlarge)

pandas provides a very convenient method `unstack` to reshape data into a pivot table. For example, you can rearrange the data created in Figure 5.5 into a DataFrame:

```
In [9]: data.unstack()
Out[9]:
```

	1	2	3
a	0.747121	-0.910733	-0.610212
b	-0.562908	NaN	-1.064038
c	-0.464033	0.532145	NaN
d	NaN	0.962120	-1.193371

Fig 5.5 (click to enlarge)

The DataFrame created will use the out-level index as row labels and the inner-level index as column labels. And it will fill NaNs to elements whose row-column label combinations don't exist in the original data.

stack is the inverse operation of `unstack`:

```
In [10]: data.unstack().stack()
Out[10]:
a 1    0.747121
   2   -0.910733
   3   -0.610212
b 1   -0.562908
   3   -1.064038
c 1   -0.464033
   2    0.532145
d 2    0.962120
   3   -1.193371
dtype: float64
```

Fig 5.6 (click to enlarge)

The two axes in the DataFrame can have a hierarchical index as shown in Figure 5.7:



Fig

5.7

The highlighted regions in Figure 5. show the hierarchical indexes.

You can also give names to the hierarchical levels as shown in Figure 5.8:



Fig

5.8

As indicated by the arrow, the *frame* has level names now.

Similarly, you can select subsets of data by using column index:

```
In [16]: frame['Ohio']
Out[16]:
color      Green  Red
key1 key2
a      1      0    1
      2      3    4
b      1      6    7
      2      9   10
```

Fig 5.9 (click to enlarge)

Reordering and Sorting Levels, Summary Statistics by Level, and Indexing with a DataFrames Columns

(click the tabs to learn more)

Reordering and Sorting Levels

Sometimes, you have to rearrange the order of axis levels or sort the data according to the values of a level.

The method ***swaplevel*** as show in Figure 5.10 can switch two levels:

```
In [17]: frame.swaplevel('key1', 'key2')
Out[17]:
```

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

Fig 5.10 (click to enlarge)

sort index can sort the data for a level:



Fig

5.11

Summary Statistics by Level

Most statistic methods of Series and DataFrame have an option to specify the level you would like to aggregate.

Consider the data *frame* in Figure 5.12, you can sum rows or columns by level:

```
In [20]: frame.sum(level = 'key2')           # Column sums for level = key2
Out[20]:
```

	state	Ohio	Colorado
color		Green	Red
key2			
1		6	8
2		12	14

```
In [21]: frame.sum(level = 'color', axis = 1)   # Row sums for level = color
Out[21]:
```

	color	Green	Red
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

Fig 5.12 (click to enlarge)

Indexing with a Data Frame's Columns

Sometimes you may want to use some columns of the data as the index. A new DataFrame *frame 2* has been created as example in Figure 5.13:

```
In [22]: frame2 = 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]})

In [23]: frame2
Out[23]:
```

	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

Fig 5.13 (click to enlarge)

You can use **set_index** method which will create a new DataFrame using 'c' and 'd' columns as the index that is shown in Figure 5.14:

```
In [24]: frame2_indexed = frame2.set_index(['c', 'd'])

In [25]: frame2_indexed
Out[25]:
```

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

Fig 5.14 (click to enlarge)

The columns used as the index will be dropped from the *frame2* by default. You can keep them by using the argument *drop=False* as shown in Figure 5.15:

```
In [26]: frame2.set_index(['c', 'd'], drop=False)
Out[26]:
```

	a	b	c	d
one 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

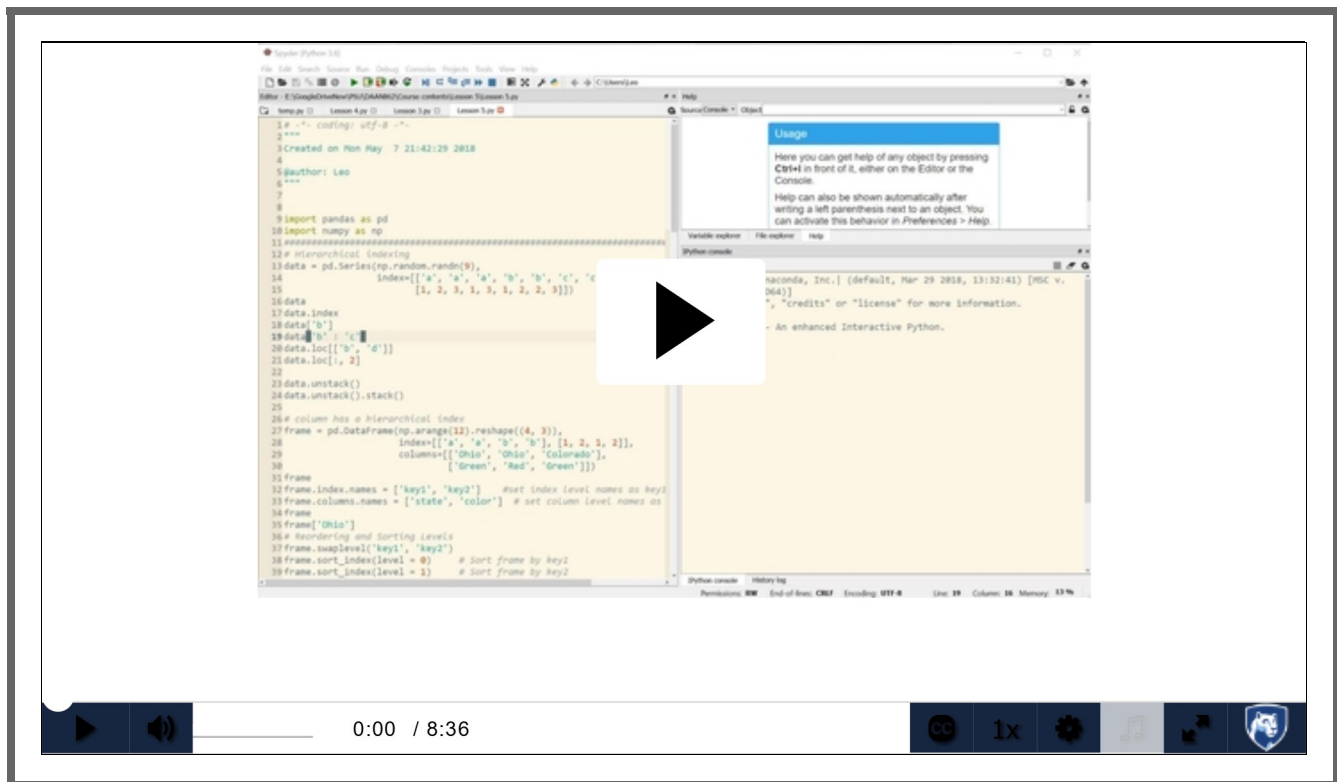
Fig 5.15 (click to enlarge)

The opposite operation of *set index* is the methods **reset index**, and all index levels will be added to the columns:

```
In [27]: frame2_indexed.reset_index()
Out[27]:
```

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

Fig 5.16 (click to enlarge)



Lesson 5.2: Merge DataSets (3 of 5)

Lesson 5.2: Merge DataSets

There are two main functions in pandas to combine different datasets: ***pandas.merge*** & ***pandas.concat***

(click the titles to learn more)

pandas.merge

See Table 5.2.1 for the arguments of the *merge* function:

Table 5.1.2: Arguments of the Merge Function

Argument	Description
left	DataFrame to be merged on the left side
right	DataFrame to be merged on the right side
how	One of 'inner', 'outer', 'left' or 'right'. 'inner' by default
on	Column names to join on. Must be found in both DataFrame objects
left on	Columns in left DataFrame to use as join keys
right on	Analogous to left_on for left DataFrame
left index	left_index Use row index in left as its join key (or keys, if a MultiIndex)
right index	Analogous to left_index
sort	Sort merged data lexicographically by join keys; True by default. Disable to get better performance in some cases on large datasets
suffixes	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y'). For example, if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result
copy	If False, avoid copying data into resulting data structure in some exceptional cases. By default always copies

(click the tabs to learn more about *pandas.merge*)

Merge on Key

Merge function can contribute datasets using one or more keys.

Figure 5.17 shows two DataFrames as examples:


```
In [28]: df1 = pd.DataFrame({'key': list('aabccde'),
...:                        'value1': range(7)})
...: df1
Out[28]:
```

	key	value1
0	a	0
1	a	1
2	b	2
3	c	3
4	c	4
5	d	5
6	e	6

```
In [29]: df2 = pd.DataFrame({'key': list('ace'),
...:                        'value2': range(4, 7)})
...: df2
Out[29]:
```

	key	value2
0	a	4
1	c	5
2	e	6

Fig 5.17 (click to enlarge)

Merge df1 and df2 is an example of a many-to-one situation. The 'key' column of df1 has multiple rows with the values of a and c, while the 'key' column of df2 has unique values. Figure 5.18 uses *merge* functions:

```
In [30]: pd.merge(df1, df2)
Out[30]:
```

	key	value1	value2
0	a	0	4
1	a	1	4
2	c	3	5
3	c	4	5
4	e	6	6

Fig 5.18 (click to enlarge)

Since we didn't set which column to join on, *merge* function will use the overlapping column as the keys, and by default it will use an inner join method. That's why the d and be values are missing in Figure 5.18

Let's try to specify the columns clearly which will return the same results as shown in Figure 5.19:

```
In [31]: pd.merge(df1, df2, on = 'key')
```

```
Out[31]:
```

	key	value1	value2
0	a	0	4
1	a	1	4
2	c	3	5
3	c	4	5
4	e	6	6

Fig 5.19 (click to enlarge)

Sometimes, the column names might be different in two datasets. In that case, you can specify them separately by using *left on* and *right on* arguments as shown in Figure 5.20:

```
In [33]: df1.columns = ['key1', 'value1'] # reset column names
...: df1
```

```
Out[33]:
```

	key1	value1
0	a	0
1	a	1
2	b	2
3	c	3
4	c	4
5	d	5
6	e	6

```
In [34]: df2.columns = ['key2', 'value2'] # reset column names
...: df2
```

```
Out[34]:
```

	key2	value2
0	a	4
1	c	5
2	e	6

```
In [35]: pd.merge(df1, df2, left_on = 'key1', right_on = 'key2')
```

```
Out[35]:
```

	key1	value1	key2	value2
0	a	0	a	4
1	a	1	a	4
2	c	3	c	5
3	c	4	c	5
4	e	6	e	6

Fig 5.20 (click to enlarge)

Instead of using default *how = 'inner'*, there are other possible options such as *'left'*, *'right'*, and *'outer'*.

Let's try to use *how = 'outer'*, which will give the union of the keys as shown in Figure 5.21:

```
In [36]: pd.merge(df1, df2, left_on = 'key1', right_on = 'key2', how = 'outer')
Out[36]:
```

	key1	value1	key2	value2
0	a	0	a	4.0
1	a	1	a	4.0
2	b	2	NaN	NaN
3	c	3	c	5.0
4	c	4	c	5.0
5	d	5	NaN	NaN
6	e	6	e	6.0

Fig 5.21 (click to enlarge)

Merging on Index

Sometimes, the keys might be the index of a DataFrame. In this case, you can use the argument `left_index=True` or `right_index=True` (or both) to specify that the index will be used as the key to merge.

In Figure 5.22, we only set the key for df2 as the index, you can merge df1 and df2 by using `right_index=True`:

```
In [37]: df2 = df2.set_index('key2')          # use key2 column as index
...: df2
Out[37]:
```

	value2
key2	
a	4
c	5
e	6

```
In [38]: pd.merge(df1, df2, left_on = 'key1', right_index = True)
Out[38]:
```

	key1	value1	value2
0	a	0	4
1	a	1	4
3	c	3	5
4	c	4	5
6	e	6	6

Fig 5.22 (click to enlarge)

Figure 5.23 is a example of a case where the key in both DataFrame is index:

```

In [39]: df1 = df1.set_index('key1')      # use key1 column as index
...: df1
Out[39]:
      value1
key1
a          0
a          1
b          2
c          3
c          4
d          5
e          6

In [40]: pd.merge(df1, df2, left_index = True, right_index = True)
Out[40]:
      value1  value2
a          0         4
a          1         4
c          3         5
c          4         5
e          6         6

```

Fig 5.23 (click to enlarge)

pandas.concat

Another type of data combination operation is binding or stacking datasets. This operation can be used if you have two datasets which are indexed differently. A number of examples will be used to illustrate how it works.

Let's consider three Series without any index overlap as shown in Figure 5.24:

```

In [41]: s1 = pd.Series([4, 8], index = list('AB'))
...: s1
Out[41]:
A    4
B    8
dtype: int64

In [42]: s2 = pd.Series([2, 12, 4], index = list('DFG'))
...: s2
Out[42]:
D     2
F    12
G     4
dtype: int64

In [43]: s3 = pd.Series([1, 9], index = list('JK'))
...: s3
Out[43]:
J     1
K     9
dtype: int64

```

Figure 5.24 (click to enlarge)

You can *concat* s1, s2, and s3 along axis=0 (by default), producing another series as shown in Figure 5.25:

```
In [44]: pd.concat([s1, s2, s3])          # row binding
Out[44]:
A      4
B      8
D      2
F     12
G      4
J      1
K      9
dtype: int64
```

Fig 5.25 (click to enlarge)

As shown in Figure 5.26, if you pass `axis=1`, the output will be a DataFrame (`axis=1` is the columns) and missing values will be introduced if indexes cannot be found in original datasets:




Fig 5.26
(click to
enlarge)

`sort= True` is used to sort the columns as shown in Figure 5.26

The same *concat* can extend to DataFrame objects:

```
In [47]: df3 = pd.DataFrame(np.random.randn(4, 3), columns = list('ABC'))
...: df3
Out[47]:
```

	A	B	C
0	-1.077206	0.180466	-1.361480
1	-0.173679	-0.805681	0.320618
2	-0.069039	0.611569	0.811432
3	-0.103109	0.287313	0.153060

```
In [48]: df4 = pd.DataFrame(np.random.rand(3, 4),
...:                        columns = list('BCDE'),
...:                        index = range(5, 8))
...: df4
Out[48]:
```

	B	C	D	E
5	0.278789	0.026195	0.610256	0.339056
6	0.450828	0.198214	0.662145	0.645518
7	0.561556	0.439431	0.949794	0.178216

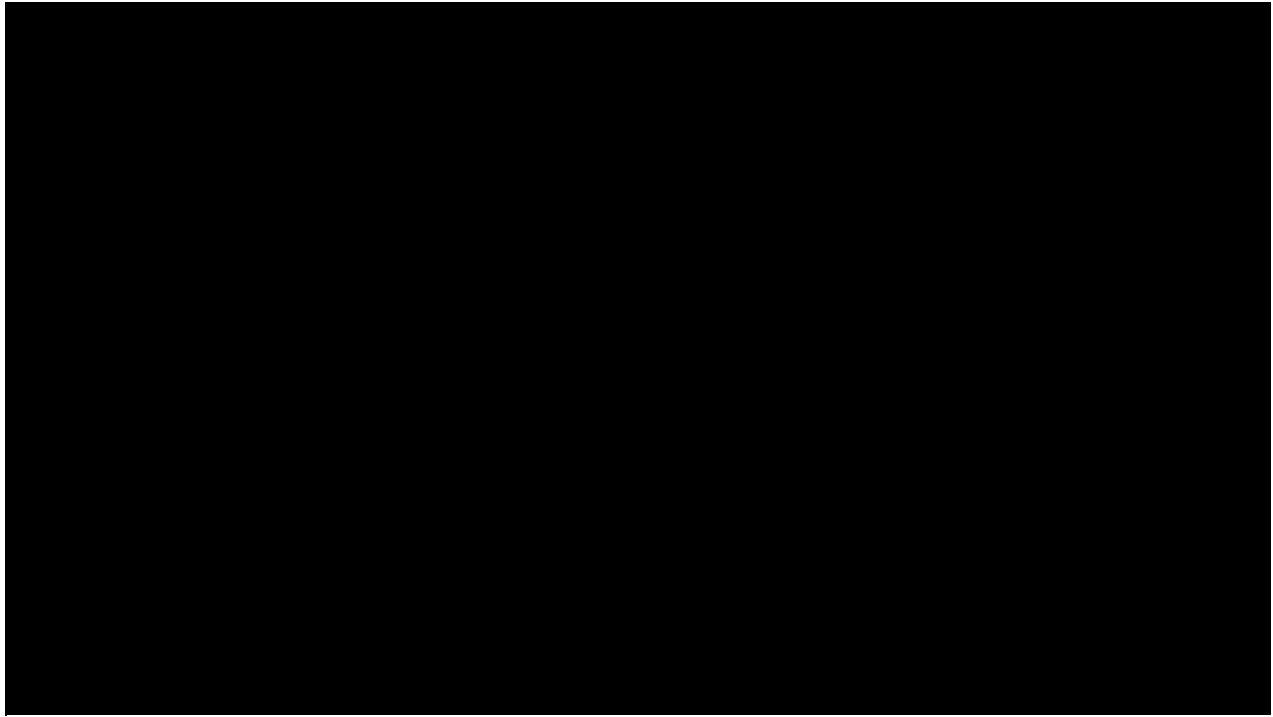
```
In [49]: pd.concat([df3, df4], sort=True)
Out[49]:
```

	A	B	C	D	E
0	-1.077206	0.180466	-1.361480	NaN	NaN
1	-0.173679	-0.805681	0.320618	NaN	NaN
2	-0.069039	0.611569	0.811432	NaN	NaN
3	-0.103109	0.287313	0.153060	NaN	NaN
5	NaN	0.278789	0.026195	0.610256	0.339056
6	NaN	0.450828	0.198214	0.662145	0.645518
7	NaN	0.561556	0.439431	0.949794	0.178216

Fig 5.27 (click to enlarge)

pandas.merge combines rows in DataFrame by one or more keys

pandas.concat stacks datasets along an axis



Lesson 5.3: Reshaping and Pivoting (4 of 5)

Lesson 5.3: Reshaping and Pivoting

There are a number of operations to rearrange tabular data. These are referred to as reshaping or pivoting.

(click titles to learn more)

Reshaping with Hierarchical Indexing

Hierarchical Indexing provides a convenient way to rearrange data.

There are two primary actions:

stack: This "rotates" or pivots from the columns in the data to the rows

unstack: This pivots from the rows into the columns



Examples

Let's use a pivot-table like DataFrame with row and column indexes as shown in Figure 5.28:

Click the dropdown to view Figure 5.28:

```
In [50]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
...:                          index=pd.Index(['Ohio', 'Colorado'], name='state'),
...:                          columns=pd.Index(['one', 'two', 'three'],
...:                                           name='number'))
...: data
Out[50]:
```

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

Fig 5.28 (click to enlarge)

Applying the stack method to this data converts the tree columns into an inner-level index and returns a Series as shown in Figure 5.29:

Click the dropdown to view Figure 5.29

```
In [51]: result = data.stack()

In [52]: result
Out[52]:
```

state	number	
Ohio	one	0
	two	1
	three	2
Colorado	one	3
	two	4
	three	5

```
dtype: int32

In [53]: type(result)
Out[53]: pandas.core.series.Series

In [54]: result.index
Out[54]:
```

```
MultiIndex(levels=[['Ohio', 'Colorado'], ['one', 'two', 'three']],
            labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]],
            names=['state', 'number'])
```

Fig 5.29 (click to enlarge)

You can use `unstack` method as shown in Figure 5.30 to convert the data back from a hierarchically indexed Series and you can indicate which index level should be unstacked:

Click the dropdown to view Figure 5.30

```
In [55]: result.unstack()                # Unstack the inner level
Out[55]:
number    one  two  three
state
Ohio      0   1   2
Colorado  3   4   5

In [56]: result.unstack('state')        # Unstack the outer level
Out[56]:
state  Ohio  Colorado
number
one     0     3
two     1     4
three   2     5
```

Fig 5.30 (click to enlarge)

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups as shown in Figure 5.31:

Click the dropdown to view Figure 5.31



```
In [57]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
In [58]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
In [59]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [60]: data2
Out[60]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64

In [61]: data2.unstack()
Out[61]:
      a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

Fig 5.31 (click to enlarge)

Figure 5.32 shows stacking filters out missing data by default, so the operation is more easily invertible:

Click the dropdown to view Figure 5.32

```
In [62]: data2.unstack().stack()
Out[62]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64

In [63]: data2.unstack().stack(dropna=False)
Out[63]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

Fig 5.32 (click to enlarge)

When you unstack in a DataFrame, the index level unstacked becomes the lowest level in the result as shown in Figure 5.33:

Click the dropdown to view Figure 5.33

```
In [64]: df = pd.DataFrame({'left': result, 'right': result + 5},
...:                        columns=pd.Index(['left', 'right'], name='side'))

In [65]: df
Out[65]:
```

side		left	right
Ohio	one	0	5
	two	1	6
	three	2	7
Colorado	one	3	8
	two	4	9
	three	5	10

```
In [66]: df.unstack('state')
Out[66]:
```

side	left		right	
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

Fig 5.33 (click to enlarge)

When calling stack, we can indicate the name of the axis to stack as shown in Figure 5.34:

Click the dropdown to view Figure 5.34

```
In [67]: df.unstack('state').stack('side')
Out[67]:
```

state		Colorado	Ohio
number	side		
one	left	3	0
	right	8	5
two	left	4	1
	right	9	6
three	left	5	2
	right	10	7

Fig 5.34 (click to enlarge)

Pivoting "Wide" to "Long" Format

An inverse operation to pivot for DataFrames is `pandas.melt`. Instead of splitting a column into many columns, it merges several columns into one producing a DataFrame that is longer than the input.

Lets look at Figure 5.35's example:

```
In [68]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
...:                       'A': [1, 2, 3],
...:                       'B': [4, 5, 6],
...:                       'C': [7, 8, 9]})

In [69]: df
Out[69]:
  key  A  B  C
0  foo  1  4  7
1  bar  2  5  8
2  baz  3  6  9
```

Fig 5.35 (click to enlarge)

The 'key' column can be used as a group indicator(*id vars*), while the rest of the columns are values(*value vars*) as shown in Figure 5.36:

```
In [70]: melted = pd.melt(df, ['key'])

In [71]: melted
Out[71]:
  key variable  value
0  foo         A      1
1  bar         A      2
2  baz         A      3
3  foo         B      4
4  bar         B      5
5  baz         B      6
6  foo         C      7
7  bar         C      8
8  baz         C      9
```

Fig 5.36 (click to enlarge)

You can convert melted data back to the original format by using *pd.pivot* as shown in Figure 5.37:

```
In [72]: reshaped = melted.pivot('key', 'variable', 'value')

In [73]: reshaped
Out[73]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

Fig 5.37 (click to enlarge)

Since the result of pivot creates an index from the column used as the row tables, we may want to use *reset index* as shown in Figure 5.38 to move the data back into a column:

```
In [74]: reshaped.reset_index()
Out[74]:
```

	variable	key	A	B	C
0		bar	2	5	8
1		baz	3	6	9
2		foo	1	4	7

Fig 5.38 (click to enlarge)

You are also able to specify columns to be value_vars. In Figure 5.39 we only choose A and B columns to melt:

```
In [75]: pd.melt(df, id_vars = ['key'], value_vars = ['A', 'B'])
Out[75]:
```

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6

Fig 5.39 (click to enlarge)

You can also use melt function without specifying *id vars*:

```
In [76]: pd.melt(df, value_vars = ['A', 'B', 'C'])
Out[76]:
```

	variable	value
0	A	1
1	A	2
2	A	3
3	B	4
4	B	5
5	B	6
6	C	7
7	C	8
8	C	9

Fig 5.40 (click to enlarge)



Lesson 5 References (5 of 5)

Lesson 5 References

- <https://pandas.pydata.org/pandas-docs/stable/tutorials.html> (<https://pandas.pydata.org/pandas-docs/stable/tutorials.html>)

Please direct questions to the [IT Service Desk](https://www.it.psu.edu/support/) (<https://www.it.psu.edu/support/>) |

The Pennsylvania State University © 2022