

PRACTICE SESSION 6

DATA WRANGLING: JOIN, COMBINE, AND RESHAPE

Data Wrangling: Join, Combine, and Reshape

- > In many applications, data may be spread across a number of files or databases or be arranged in a form that is not easy to analyze.
- > This module focuses on tools to help combine, join, and rearrange data.

HIERARCHICAL INDEXING

Hierarchical Indexing

- > *Hierarchical indexing* is an important feature of pandas that enables you to have multiple (two or more) index *levels* on an axis.
- > It provides a way for you to work with higher dimensional data in a lower dimensional form.
- > Create a Series with a list of lists (or arrays) as the index:

Hierarchical Indexing

```
In [9]: 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]]))

In [10]: data
Out[10]:
a  1   -0.204708
     2    0.478943
     3   -0.519439
b  1   -0.555730
     3    1.965781
c  1    1.393406
     2    0.092908
d  2    0.281746
     3    0.769023
dtype: float64
```

Hierarchical Indexing

- > With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data

```
In [11]: data.index
Out[11]:
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]])
```

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
     3    1.965781
c  1    1.393406
     2    0.092908
dtype: float64
```

```
In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
     3    1.965781
d  2    0.281746
     3    0.769023
dtype: float64
```

Hierarchical Indexing

- > Selection is even possible from an “inner” level

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

Hierarchical Indexing

- > Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table.
- > For example, you could rearrange the data into a DataFrame using its **unstack** method:

```
In [16]: data.unstack()
Out[16]:
      1          2          3
a -0.204708  0.478943 -0.519439
b -0.555730      NaN  1.965781
c  1.393406  0.092908      NaN
d      NaN  0.281746  0.769023
```

Hierarchical Indexing

> The inverse operation of `unstack` is `stack`:

```
In [17]: data.unstack().stack()
Out[17]:
a 1 -0.204708
   2  0.478943
   3 -0.519439
b 1 -0.555730
   3  1.965781
c 1  1.393406
   2  0.092908
d 2  0.281746
   3  0.769023
dtype: float64
```

Hierarchical Indexing

> With a DataFrame, either axis can have a hierarchical index:

```
In [18]: 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']])
In [19]: frame
Out[19]:
      Ohio      Colorado
      Green Red      Green
a 1    0    1    2
   2    3    4    5
b 1    6    7    8
   2    9   10   11
```

Hierarchical Indexing

- > The hierarchical levels can have names

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
   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
```

Hierarchical Indexing

- > With partial column indexing you can similarly select groups of columns

```
In [23]: frame['Ohio']
Out[23]:
   color      Green    Red
key1 key2
   a    1        0        1
        2        3        4
   b    1        6        7
        2        9       10
```

Reordering and Sorting Levels

- > At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level.
- > The `swaplevel` takes two level numbers or names and returns a new object with the levels interchanged

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio      Colorado
color      Green   Red      Green
key2 key1
1    a        0     1      2
2    a        3     4      5
1    b        6     7      8
2    b        9    10     11
```

Reordering and Sorting Levels

- > `sort_index`, on the other hand, sorts the data using only the values in a single level.
- > When swapping levels, it's not uncommon to also use `sort_index` so that the result is lexicographically sorted by the indicated level:

```
In [25]: frame.sort_index(level=1)
Out[25]:
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 [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
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
```

Summary Statistics by Level

- > Many descriptive and summary statistics on DataFrame and Series have a level option in which you can specify the level you want to aggregate by on a particular axis.

```
In [27]: frame.sum(level='key2')
Out[27]:
state    Ohio      Colorado
color   Green   Red
key1 key2
1          6     8       10
2         12    14       16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green   Red
key1 key2
a        1       2     1
           2       8     4
b        1      14     7
           2      20    10
```

Indexing with a DataFrame's columns

- > You may want to use one or more columns from a DataFrame as the row index
- > You may also wish to move the row index into the DataFrame's columns.

```
In [29]: 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]})

In [30]: frame
Out[30]:
   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
```

Indexing with a DataFrame's columns

- > DataFrame's `set_index` function will create a new DataFrame using one or more of its columns as the index:

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
   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
```

Indexing with a DataFrame's columns

- > By default the columns are removed from the DataFrame, though you can leave them in:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
   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
```

Indexing with a DataFrame's columns

> `reset_index`, on the other hand, does the opposite of `set_index`

> The hierarchical index levels are moved into the columns:

```
In [34]: frame2.reset_index()
Out[34]:
   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
```

COMBINING AND MERGING DATASETS

Combining and Merging Datasets

- > Data contained in pandas objects can be combined together in a number of ways:
 - `pandas.merge` connects rows in DataFrames based on one or more keys.
 - This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
 - `pandas.concat` concatenates or “stacks” together objects along an axis.
 - The `combine_first` instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

Database-Style DataFrame Joins

- > *Merge* or *join* operations combine datasets by linking rows using one or more keys.
- > These operations are central to relational databases (e.g., SQL-based).
- > The `merge` function in pandas is the main entry point for using these algorithms on your data.

Database-Style DataFrame Joins

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
....:                         'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
....:                         'data2': range(3)})

In [37]: df1
Out[37]:
   data1  key
0      0    b
1      1    b
2      2    a
3      3    c
4      4    a
5      5    a
6      6    b

In [38]: df2
Out[38]:
   data2  key
0      0    a
1      1    b
2      2    d
```

Database-Style DataFrame Joins

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1  key  data2
0      0    b      1
1      1    b      1
2      6    b      1
3      2    a      0
4      4    a      0
5      5    a      0
```

- > This is an example of a **many-to-one** join; the data in **df1** has multiple rows labeled **a** and **b**, whereas **df2** has only one row for each value in the key column.

Database-Style DataFrame Joins

- > Note that we didn't specify which column to join on.
- > If that information is not specified, merge uses the overlapping column names as the keys.
- > It's a good practice to specify explicitly:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1  key  data2
0      0    b      1
1      1    b      1
2      6    b      1
3      2    a      0
4      4    a      0
5      5    a      0
```

Database-Style DataFrame Joins

- > If the column names are different in each object, you can specify them separately:

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                           ....: 'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
                           ....: 'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1  lkey  data2  rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

Database-Style DataFrame Joins

- > You may notice that the '**c**' and '**d**' values and associated data are missing from the result.
- > By default **merge** does an '**inner**' join
 - the keys in the result are the intersection, or the common set found in both tables.
- > Other possible options are '**left**', '**right**', and '**outer**'.

Database-Style DataFrame Joins

- > The outer join takes the union of the keys, combining the effect of applying both left and right joins

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1  key  data2
0    0.0    b    1.0
1    1.0    b    1.0
2    6.0    b    1.0
3    2.0    a    0.0
4    4.0    a    0.0
5    5.0    a    0.0
6    3.0    c    NaN
7    NaN    d    2.0
```

Database-Style DataFrame Joins

- > *Many-to-many* merges have well-defined, though not necessarily intuitive, behavior.

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
....:                      'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
....:                      'data2': range(5)})

In [47]: df1
Out[47]:
   data1  key
0      0    b
1      1    b
2      2    a
3      3    c
4      4    a
5      5    b

In [48]: df2
Out[48]:
   data2  key
0      0    a
1      1    b
2      2    a
3      3    b
4      4    d
```

Database-Style DataFrame Joins

- > 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.

```
In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
   data1  key  data2
0      0    b    1.0
1      0    b    3.0
2      1    b    1.0
3      1    b    3.0
4      2    a    0.0
5      2    a    2.0
6      3    c    NaN
7      4    a    0.0
8      4    a    2.0
9      5    b    1.0
10     5    b    3.0
```

Database-Style DataFrame Joins

- > The join method only affects the distinct key values appearing in the result

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1  key  data2
0      0    b     1
1      0    b     3
2      1    b     1
3      1    b     3
4      5    b     1
5      5    b     3
6      2    a     0
7      2    a     2
8      4    a     0
9      4    a     2
```

Database-Style DataFrame Joins

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

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
....:                         'key2': ['one', 'two', 'one'],
....:                         'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
....:                           'key2': ['one', 'one', 'one', 'two'],
....:                           'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[53]:
   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
```

Database-Style DataFrame Joins

- > A last issue to consider in merge operations is the treatment of overlapping column names.
- > merge has a **suffixes** option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
   key1  key2_x  lval  key2_y  rval
0  foo     one     1    one     4
1  foo     one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar     one     3    one     6
5  bar     one     3   two     7
```

Database-Style DataFrame Joins

- > A last issue to consider in merge operations is the treatment of overlapping column names.
- > merge has a **suffixes** option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[55]:
   key1  key2_left  lval  key2_right  rval
0  foo     one     1        one     4
1  foo     one     1        one     5
2  foo    two     2        one     4
3  foo    two     2        one     5
4  bar     one     3        one     6
5  bar     one     3       two     7
```

Different join types with how argument

Option	Behavior
'inner'	Use only the key combinations observed in both tables
'left'	Use all key combinations found in the left table
'right'	Use all key combinations found in the right table
'outer'	Use all key combinations observed in both tables together

merge function arguments

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 '; defaults to ' inner '
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in left and right as the join keys.
left_on	Columns in left DataFrame to use as join keys.
right_on	Analogous to left_on for left DataFrame.
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).

merge function arguments

Argument	Description
suffixes	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., 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.
indicator	Adds a special column <code>_merge</code> that indicates the source of each row; values will be ' <code>left_only</code> ', ' <code>right_only</code> ', or ' <code>both</code> ' based on the origin of the joined data in each row.

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

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [58]: left1
Out[58]:
   key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0
```

Merging on Index

```
In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
   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
```

Merging on Index

- > Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
   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
5   c      5       NaN
```

Merging on Index

- > With hierarchically indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge:

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
.....:                               'Nevada', 'Nevada'],
.....:                               'key2': [2000, 2001, 2002, 2001, 2002],
.....:                               'data': np.arange(5.)})

In [63]: 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'])
```

Merging on Index

	In [64]: lefth	In [65]: righth
Out[64]:		Out[65]:
	data key1 key2	event1 event2
0	0.0 Ohio 2000	Nevada 2001 0 1
1	1.0 Ohio 2001	2000 2 3
2	2.0 Ohio 2002	Ohio 2000 4 5
3	3.0 Nevada 2001	2000 6 7
4	4.0 Nevada 2002	2001 8 9
		2002 10 11

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data  key1  key2  event1  event2
0  0.0  Ohio  2000      4      5
0  0.0  Ohio  2000      6      7
1  1.0  Ohio  2001      8      9
2  2.0  Ohio  2002     10     11
3  3.0 Nevada 2001      0      1
```

```
In [64]: lefth
Out[64]:
   data  key1  key2
0  0.0  Ohio  2000
1  1.0  Ohio  2001
2  2.0  Ohio  2002
3  3.0 Nevada 2001
4  4.0 Nevada 2002

In [65]: righth
Out[65]:
           event1  event2
Nevada  2001      0      1
          2000      2      3
Ohio    2000      4      5
          2001      6      7
          2002      8      9
                     10     11
```

```
In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
.....:                 right_index=True, how='outer')
Out[67]:
   data  key1  key2  event1  event2
0  0.0  Ohio  2000    4.0    5.0
0  0.0  Ohio  2000    6.0    7.0
1  1.0  Ohio  2001    8.0    9.0
2  2.0  Ohio  2002   10.0   11.0
3  3.0 Nevada 2001    0.0    1.0
4  4.0 Nevada 2002    NaN    NaN
4  NaN  Nevada 2000    2.0    3.0
```

Merging on Index

- > Using the indexes of both sides of the merge is also possible

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
.....:                         index=['a', 'c', 'e'],
.....:                         columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13., 14.]],
.....:                         index=['b', 'c', 'd', 'e'],
.....:                         columns=['Missouri', 'Alabama'])
```

	In [70]: left2		In [71]: right2		
	Out[70]:	Out[71]:	Out[71]:	Out[71]:	
	Ohio	Nevada	Missouri	Alabama	
a	1.0	2.0	b	7.0	8.0
c	3.0	4.0	c	9.0	10.0
e	5.0	6.0	d	11.0	12.0
			e	13.0	14.0

Merging on Index

- > Using the indexes of both sides of the merge is also possible

```
In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[72]:
   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
```

Merging on Index

- > DataFrame has a convenient `join` instance for merging by index.
- > It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   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
```

Merging on Index

- > In part for legacy reasons (i.e., much earlier versions of pandas), DataFrame's **join** method performs a left join on the join keys, exactly preserving the left frame's row index.
- > It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [74]: left1.join(right1, on='key')
Out[74]:
   key  value  group_val
0    a      0       3.5
1    b      1       7.0
2    a      2       3.5
3    a      3       3.5
4    b      4       7.0
5    c      5       NaN
```

Merging on Index

- > For simple index-on-index merges, you can pass a list of DataFrames to **join** as an alternative to using the more general **concat** function

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
....:                               index=['a', 'c', 'e', 'f'],
....:                               columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
   New York  Oregon
a        7.0     8.0
c        9.0    10.0
e       11.0    12.0
f       16.0    17.0
```

Merging on Index

```
In [77]: left2.join([right2, another])
Out[77]:
      Ohio  Nevada  Missouri  Alabama  New York  Oregon
a    1.0      2.0       NaN       NaN      7.0      8.0
c    3.0      4.0       9.0      10.0      9.0     10.0
e    5.0      6.0      13.0      14.0     11.0     12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
      Ohio  Nevada  Missouri  Alabama  New York  Oregon
a    1.0      2.0       NaN       NaN      7.0      8.0
b    NaN      NaN       7.0      8.0      NaN      NaN
c    3.0      4.0       9.0      10.0      9.0     10.0
d    NaN      NaN      11.0      12.0      NaN      NaN
e    5.0      6.0      13.0      14.0     11.0     12.0
f    NaN      NaN       NaN       NaN     16.0     17.0
```

Concatenating Along an Axis

- > Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking.
- > NumPy's **concatenate** function can do this with NumPy arrays:

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

Concatenating Along an Axis

- > In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation.
- > In particular, you have a number of additional things to think about:
 - If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?
 - Do the concatenated chunks of data need to be identifiable in the resulting object?
 - Does the “concatenation axis” contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation.

Concatenating Along an Axis

- > The **concat** function in pandas provides a consistent way to address these concerns.

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])  
In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])  
In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
In [85]: pd.concat([s1, s2, s3])  
Out[85]:  
a    0  
b    1  
c    2  
d    3  
e    4  
f    5  
g    6  
dtype: int64
```

Concatenating Along an Axis

- > By default `concat` works along `axis=0`, producing another Series. If you pass `axis=1`, the result will instead be a DataFrame (`axis=1` is the columns):

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
   0   1   2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

Concatenating Along an Axis

- > In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes.
- > You can instead intersect them by passing `join='inner'`

```
In [87]: s4 = pd.concat([s1, s3])
In [88]: s4
Out[88]:
a    0
b    1
f    5
g    6
dtype: int64
```

```
In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
   0   1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6
```

```
In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
   0   1
a  0   0
b  1   1
```

Concatenating Along an Axis

> You can even specify the axes to be used on the other axes with `join_axes`:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[91]:
      0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

Concatenating Along an Axis

> The same logic extends to DataFrame objects:

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
....:                  columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
....:                  columns=['three', 'four'])

In [98]: df1
Out[98]:
   one  two
a    0   1
b    2   3
c    4   5

In [99]: df2
Out[99]:
   three  four
a      5    6
c      7    8
```

```
In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
      level1    level2
      one  two  three  four
a      0   1    5.0   6.0
b      2   3    NaN   NaN
c      4   5    7.0   8.0
```

Concatenating Along an Axis

- > A last consideration concerns DataFrames in which the row index does not contain any relevant data:

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

```
In [105]: df1
```

```
Out[105]:
```

```
      a        b        c        d  
0  1.246435  1.007189 -1.296221  0.274992  
1  0.228913  1.352917  0.886429 -2.001637  
2 -0.371843  1.669025 -0.438570 -0.539741
```

```
In [106]: df2
```

```
Out[106]:
```

```
      b        d        a  
0  0.476985  3.248944 -1.021228  
1 -0.577087  0.124121  0.302614
```

```
In [107]: pd.concat([df1, df2], ignore_index=True)
```

```
Out[107]:
```

```
      a        b        c        d  
0  1.246435  1.007189 -1.296221  0.274992  
1  0.228913  1.352917  0.886429 -2.001637  
2 -0.371843  1.669025 -0.438570 -0.539741  
3 -1.021228  0.476985      NaN  3.248944  
4  0.302614 -0.577087      NaN  0.124121
```

Concatenating Along an Axis

Argument	Description
levels	Specific indexes to use as hierarchical index level or levels if keys passed
names	Names for created hierarchical levels if keys and/or levels passed
verify_integrity	Check new axis in concatenated object for duplicates and raise exception if so; by default (False) allows duplicates
ignore_index	Do not preserve indexes along concatenation axis , instead producing a new range(total_length) index

RESHAPING WITH HIERARCHICAL INDEXING

Reshaping with Hierarchical Indexing

- > There are a number of basic operations for rearranging tabular data.
- > These are alternately referred to as *reshape* or *pivot* operations.

Reshaping with Hierarchical Indexing

- > Hierarchical indexing provides a consistent way to rearrange data in a DataFrame.
- > 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

Reshaping with Hierarchical Indexing

- > Example:

```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),  
.....: index=pd.Index(['Ohio', 'Colorado'], name='state'),  
.....: columns=pd.Index(['one', 'two', 'three'],  
.....: name='number'))  
  
In [121]: data  
Out[121]:  
number    one   two   three  
state  
Ohio      0     1     2  
Colorado  3     4     5
```

Reshaping with Hierarchical Indexing

- > Using the **stack** method on this data pivots the columns into the rows, producing a Series:

```
In [122]: result = data.stack()

In [123]: result
Out[123]:
state    number
Ohio      one    0
          two    1
          three   2
Colorado  one    3
          two    4
          three   5
dtype: int64
```

Reshaping with Hierarchical Indexing

- > From a hierarchically indexed Series, you can rearrange the data back into a Data-Frame with **unstack**:

```
In [124]: result.unstack()
Out[124]:
number   one  two  three
state
Ohio      0    1    2
Colorado  3    4    5
```

Reshaping with Hierarchical Indexing

- > By default the innermost level is unstacked (same with stack).
- > You can unstack a different level by passing a level number or name:

```
In [125]: result.unstack(0)
Out[125]:
state    Ohio    Colorado
number
one        0        3
two        1        4
three      2        5

In [126]: result.unstack('state')
Out[126]:
state    Ohio    Colorado
number
one        0        3
two        1        4
three      2        5
```

Reshaping with Hierarchical Indexing

- > Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [130]: data2
Out[130]:
one   a    0
      b    1
      c    2
      d    3
two   c    4
      d    5
      e    6
dtype: int64
```

```
In [131]: data2.unstack()
Out[131]:
           a    b    c    d    e
one   0.0  1.0  2.0  3.0  NaN
two   NaN  NaN  4.0  5.0  6.0
```

Reshaping with Hierarchical Indexing

- > Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [132]: data2.unstack()  
Out[132]:  
      a    b    c    d    e  
one  0.0  1.0  2.0  3.0  NaN  
two  NaN  NaN  4.0  5.0  6.0
```

```
In [133]: data2.unstack().stack()  
Out[133]:  
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
```

Reshaping with Hierarchical Indexing

- > Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [134]: data2.unstack().stack(dropna=False)  
Out[134]:  
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
```

Reshaping with Hierarchical Indexing

- > When you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},  
.....:           columns=pd.Index(['left', 'right'], name='side'))
```

```
In [136]: df  
Out[136]:  
      side      left  right  
state  number  
Ohio    one       0     5  
       two       1     6  
       three      2     7  
Colorado one      3     8  
       two      4     9  
       three     5    10
```

```
In [137]: df.unstack('state')  
Out[137]:  
      side      left      right  
state Ohio Colorado Ohio Colorado  
number  
one       0       3       5       8  
two       1       4       6       9  
three      2       5       7      10
```

Reshaping with Hierarchical Indexing

- > When calling stack, we can indicate the name of the axis to stack:

```
In [138]: df.unstack('state').stack('side')  
Out[138]:  
      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
```

Pivoting “Long” to “Wide” Format

- > A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format:

```
In [139]: data = pd.read_csv('examples/macrodata.csv')

In [140]: data.head()
Out[140]:
   year  quarter  realgdp  realcons  realinv  realgovt  realdpi  cpi \
0  1959.0      1.0  2710.349  1707.4  286.898  470.045  1886.9  28.98
1  1959.0      2.0  2778.801  1733.7  310.859  481.301  1919.7  29.15
2  1959.0      3.0  2775.488  1751.8  289.226  491.260  1916.4  29.35
3  1959.0      4.0  2785.204  1753.7  299.356  484.052  1931.3  29.37
4  1960.0      1.0  2847.699  1770.5  331.722  462.199  1955.5  29.54

   m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00      0.00
1  141.7      3.08    5.1  177.830  2.34      0.74
2  140.5      3.82    5.3  178.657  2.74      1.09
3  140.0      4.33    5.6  179.386  0.27      4.06
4  139.6      3.50    5.2  180.007  2.31      1.19
```

Pivoting “Long” to “Wide” Format

```
In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
.....:                               name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end')

In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

Pivoting “Long” to “Wide” Format

```
In [146]: ldata[:10]
Out[146]:
    date      item    value
0 1959-03-31  realgdp  2710.349
1 1959-03-31      infl   0.000
2 1959-03-31     unemp   5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30      infl   2.340
5 1959-06-30     unemp   5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30      infl   2.740
8 1959-09-30     unemp   5.300
9 1959-12-31  realgdp  2785.204
```

Pivoting “Wide” to “Long” Format

- > An inverse **operation** to pivot for DataFrames is **pandas.melt**.
- > Rather than transforming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input.

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
.....:                   'A': [1, 2, 3],
.....:                   'B': [4, 5, 6],
.....:                   'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

Pivoting “Wide” to “Long” Format

- > 'key' column may be a group indicator, and other columns are data values
- > When using `pandas.melt`, we must indicate which columns (if any) are group indicators.
- > Let's use 'key' as the only group indicator here:

```
In [159]: melted = pd.melt(df, ['key'])
```

```
In [160]: melted
Out[160]:
   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
```

Pivoting “Wide” to “Long” Format

- > Using pivot, we can reshape back to the original layout:

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')
```

```
In [162]: reshaped
Out[162]:
variable  A  B  C
key
bar      2  5  8
baz      3  6  9
foo      1  4  7
```

Pivoting “Wide” to “Long” Format

- > Since the result of `pivot` creates an index from the column used as the row labels, we may want to use `reset_index` to move the data back into a column:

```
In [163]: reshaped.reset_index()
Out[163]:
variable  key   A   B   C
0          bar   2   5   8
1          baz   3   6   9
2          foo   1   4   7
```

Pivoting “Wide” to “Long” Format

- > You can also specify a subset of columns to use as value columns:

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
    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
```

Pivoting “Wide” to “Long” Format

> `pandas.melt` can be used without any group identifiers, too:

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
   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
```

Pivoting “Wide” to “Long” Format

```
In [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[166]:
   variable  value
0        key    foo
1        key    bar
2        key    baz
3          A     1
4          A     2
5          A     3
6          B     4
7          B     5
8          B     6
```