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

8.1 Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index levels on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example; create a Series with a list of lists (or arrays) as the index:

In [3]:
```
import pandas as pd
import numpy as np
data = pd.Series(np.random.randn(9),index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd',
data
```

Out[3]:
```
a  1   -1.511868
   2   -1.088178
   3    0.191063
b  1    2.568856
   3    0.508130
c  1   -1.248153
   2   -2.162173
d  2   -0.164398
   3   -0.054006
dtype: float64
```

What you're seeing is a prettified view of a Series with a MultiIndex as its index. The "gaps" in the index display mean "use the label directly above":

In [4]:
```
data.index
```

Out[4]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

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

In [5]:
```
data['b']
```

Out[5]:
```
1    2.568856
3    0.508130
dtype: float64
```

In [6]:
```
data['b':'c']
```

Out[6]:
```
b  1    2.568856
   3    0.508130
c  1   -1.248153
   2   -2.162173
dtype: float64
```

In [7]:
```
data.loc[['b', 'd']]
```

Out[7]:
```
b  1    2.568856
   3    0.508130
d  2   -0.164398
   3   -0.054006
dtype: float64
```

Selection is even possible from an “inner” level:

In [8]:
```
data.loc[:, 2]
```

Out[8]:
```
a   -1.088178
c   -2.162173
d   -0.164398
dtype: float64
```

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

Out[9]:

	1	2	3
a	-1.511868	-1.088178	0.191063
b	2.568856	NaN	0.508130
c	-1.248153	-2.162173	NaN
d	NaN	-0.164398	-0.054006

The inverse operation of unstack is stack:

In [12]:
```
data.unstack().stack()
```

Out[12]:
```
a  1   -1.511868
   2   -1.088178
   3    0.191063
b  1    2.568856
   3    0.508130
c  1   -1.248153
   2   -2.162173
d  2   -0.164398
   3   -0.054006
dtype: float64
```

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

In [13]:
```
frame = pd.DataFrame(np.arange(12).reshape((4, 3)),index=[['a', 'a', 'b', 'b'], [1,
```

In [14]:
```
frame
```

Out[14]:

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

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output:

In [15]:
```
frame.index.names = ['key1', 'key2']
```

In [16]:
```
frame.columns.names = ['state', 'color']
```

In [17]:
```
frame
```

Out[17]:

	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

Be careful to distinguish the index names 'state' and 'color' from the row labels.

With partial column indexing you can similarly select groups of columns:

In [18]:
```
frame['Ohio']
```

Out[18]:

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

A MultiIndex can be created by itself and then reused; the columns in the preceding DataFrame with level names could be created like this:

In [20]:
```
pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']]
```

Out[20]:
```
MultiIndex([(    'Ohio', 'Green'),
            (    'Ohio',   'Red'),
            ('Colorado', 'Green')],
           names=['state', 'color'])
```

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 (but the data is otherwise unaltered):

In [21]:
```
frame.swaplevel('key1', 'key2')
```

Out[21]:

	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

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 [22]:
```
frame.sort_index(level=1)
```

Out[22]:

	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 [23]:
```
frame.swaplevel(0, 1).sort_index(level=0)
```

Out[23]:

	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

Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level—that is, the result of calling sort_index(level=0) or sort_index().

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. Consider the above DataFrame; we can aggregate by level on either the rows or columns like so:

In [24]:
```
frame.sum(level='key2')
```

Out[24]:

state	Ohio		Colorado
color	Green	Red	Green
key2			
1	6	8	10
2	12	14	16

In [25]:
```
frame.sum(level='color', axis=1)
```

Out[25]:

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

Under the hood, this utilizes pandas's groupby machinery, which will be discussed in more detail later in the book.

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:

In [26]:
```
frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),'c': ['one', 'one', 'one'
```

In [27]:
```
frame
```

Out[27]:

	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

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

In [28]:
```
frame2 = frame.set_index(['c', 'd'])
```

In [29]:
```
frame2
```

Out[29]:

		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

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

In [30]:
```
frame.set_index(['c', 'd'], drop=False)
```

Out[30]:

		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

reset_index, on the other hand, does the opposite of set_index; the hierarchical index levels are moved into the columns:

In [31]:
```
frame2.reset_index()
```

Out[31]:

	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

8.2 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.

I will address each of these and give a number of examples. They’ll be utilized in examples throughout the rest of the book

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.

Let’s start with a simple example:

In [32]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],'data2': range(3)})
```

In [33]:
```
df1
```

Out[33]:

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

In [34]:
```
df2
```

Out[34]:

	key	data2
0	a	0
1	b	1
2	d	2

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. Calling merge with these objects we obtain:

In [35]:
```
pd.merge(df1, df2)
```

Out[35]:

	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

Note that I 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, though:

In [36]:
```
pd.merge(df1, df2, on='key')
```

Out[36]:

	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

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

In [37]:
```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],'data2': range(3)})
```

In [38]:
```
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

Out[38]:

	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

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'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

In [39]:
```
pd.merge(df1, df2, how='outer')
```

Out[39]:

	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

Table 8-1. 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

'output' --> Use all key combinations observed in both tables together

Many-to-many merges have well-defined, though not necessarily intuitive, behavior. Here's an example:

In [40]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],'data1': range(6)})
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],'data2': range(5)})
```

In [41]:
```
df1
```

Out[41]:

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

In [42]:
```
df2
```

Out[42]:

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

In [43]:
```
pd.merge(df1, df2, on='key', how='left')
```

Out[43]:

	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:

In [44]:
```
pd.merge(df1, df2, how='inner')
```

Out[44]:

	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 [45]:
```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],'key2': ['one', 'two', 'one'],'l
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],'key2': ['one', 'one', '
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

Out[45]:

	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

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it's not actually implemented that way).

When you're joining columns-on-columns, the indexes on the passed DataFrame objects are discarded

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:

In [46]:
```
pd.merge(left, right, on='key1')
```

Out[46]:

	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

In [47]:
```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Out[47]:

	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

Table 8-2. 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).

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 _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' 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 [48]:
```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],'value': range(6)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

In [49]:
```
left1
```

Out[49]:

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

In [50]:
```
right1
```

Out[50]:

	group_val
a	3.5
b	7.0

In [51]:
```
pd.merge(left1, right1, left_on='key', right_index=True)
```

Out[51]:

	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

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

In [52]:
```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

Out[52]:

	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

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

In [53]:
```
lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio','Nevada', 'Nevada'],'key2': [
righth = pd.DataFrame(np.arange(12).reshape((6, 2)),index=[['Nevada', 'Nevada', 'Oh
lefth
```

Out[53]:

	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 [54]:
```
righth
```

Out[54]:

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

In this case, you have to indicate multiple columns to merge on as a list (note the handling of duplicate index values with how='outer'):

In [55]:
```
pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
```

Out[55]:

	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 [56]:
```
pd.merge(lefth, righth, left_on=['key1', 'key2'],right_index=True, how='outer')
```

Out[56]:

	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 [57]:
```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],index=['a', 'c', 'e'],columns=[
```

In [58]:
```
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],index=['b', 'c',
```

In [59]:
```python
left2
```

Out[59]:

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

In [60]:
```python
right2
```

Out[60]:

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

In [61]:
```python
pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

Out[61]:

	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

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 the prior example, we could have written:

In [62]:
```python
left2.join(right2, how='outer')
```

Out[62]:

	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 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 col- umns of the calling DataFrame:

In [63]:
```
left1.join(right1, on='key')
```

Out[63]:

	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

Lastly, 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 described in the next section:

In [64]:
```
another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],index=['a', 'c
```

In [65]:
```
another
```

Out[65]:

	New York	Oregon
a	7.0	8.0
c	9.0	10.0
e	11.0	12.0
f	16.0	17.0

In [66]:
```
left2.join([right2, another])
```

Out[66]:

	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 [67]:
```
left2.join([right2, another], how='outer')
```

Out[67]:

	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
b	NaN	NaN	7.0	8.0	NaN	NaN
d	NaN	NaN	11.0	12.0	NaN	NaN
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 [68]:
```
arr = np.arange(12).reshape((3, 4))
```

In [69]:
```
arr
```

Out[69]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [70]:
```
np.concatenate([arr, arr], axis=1)
```

Out[70]:
```
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]])
```

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.

The concat function in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

In [71]:
```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

Calling concat with these objects in a list glues together the values and indexes:

In [72]:
```
pd.concat([s1, s2, s3])
```

Out[72]:
```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

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 [73]:
```
pd.concat([s1, s2, s3], axis=1)
```

Out[73]:

	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

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 [74]:
```
s4 = pd.concat([s1, s3])
s4
```

Out[74]:
```
a    0
b    1
f    5
g    6
dtype: int64
```

In [75]:
```
pd.concat([s1, s4], axis=1)
```

Out[75]:

	0	1
a	0.0	0
b	1.0	1
f	NaN	5
g	NaN	6

In [76]:
```
pd.concat([s1, s4], axis=1, join='inner')
```

Out[76]:

	0	1
a	0	0
b	1	1

In this last example, the 'f' and 'g' labels disappeared because of the join='inner' option.

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

In [77]:
```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-77-28e446bc353b> in <module>
----> 1 pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])

TypeError: concat() got an unexpected keyword argument 'join_axes'
```

A potential issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

In [78]:
```
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
result
```

Out[78]:
```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

In [79]:
```
result.unstack()
```

Out[79]:

	a	b	f	g
one	0.0	1.0	NaN	NaN
two	0.0	1.0	NaN	NaN
three	NaN	NaN	5.0	6.0

In the case of combining Series along axis=1, the keys become the DataFrame column headers:

In [80]:
```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

Out[80]:

	one	two	three
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

The same logic extends to DataFrame objects:

In [81]:
```
df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],columns=['one'
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],columns=['three
```

In [82]:
```
df1
```

Out[82]:

	one	two
a	0	1
b	2	3
c	4	5

In [83]:
```
df2
```

Out[83]:

	three	four
a	5	6
c	7	8

In [84]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

Out[84]:

	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

If you pass a dict of objects instead of a list, the dict's keys will be used for the keys option:

In [85]:
```
pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

Out[85]:

	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

There are additional arguments governing how the hierarchical index is created (see Table 8-3). For example, we can name the created axis levels with the names argument:

In [86]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],names=['upper', 'lower'])
```

Out[86]:

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

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

In [87]:
```
df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
df1
```

Out[87]:

	a	b	c	d
0	-0.804745	-1.753527	1.518087	0.369079
1	0.864903	-1.528546	1.801207	1.262016
2	-0.062502	0.139772	0.729604	0.221411

In [88]:
```
df2
```

Out[88]:

	b	d	a
0	0.424978	-0.770740	0.515311
1	-1.126576	-0.590416	1.684357

In this case, you can pass ignore_index=True:

In [89]:
```
pd.concat([df1, df2], ignore_index=True)
```

Out[89]:

	a	b	c	d
0	-0.804745	-1.753527	1.518087	0.369079
1	0.864903	-1.528546	1.801207	1.262016
2	-0.062502	0.139772	0.729604	0.221411
3	0.515311	0.424978	NaN	-0.770740
4	1.684357	-1.126576	NaN	-0.590416

Table 8-3. concat function arguments

Argument --> Description

objs --> List or dict of pandas objects to be concatenated; this is the only required argument

axis --> Axis to concatenate along; defaults to 0 (along rows)

join --> Either 'inner' or 'outer' ('outer' by default); whether to intersection (inner) or union (outer) together indexes along the other axes

join_axes --> Specific indexes to use for the other n–1 axes instead of performing union/intersection logic

keys --> Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (gif multiple-level arrays passed in levels)

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

Combining Data with Overlap

There is another data combination situation that can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's where function, which performs the array-oriented equivalent of an if-else expression:

In [91]:
```
a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],index=['f', 'e', 'd', 'c', 'b
b = pd.Series(np.arange(len(a), dtype=np.float64),index=['f', 'e', 'd', 'c', 'b', '
b[-1] = np.nan
```

In [92]:
```
a
```

Out[92]:
```
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64
```

In [93]:
```
b
```

Out[93]:
```
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64
```

In [94]:
```
np.where(pd.isnull(a), b, a)
```

Out[94]:
```
array([0. , 2.5, 2. , 3.5, 4.5, nan])
```

Series has a combine_first method, which performs the equivalent of this operation along with pandas's usual data alignment logic:

In [95]:
```
b[:-2].combine_first(a[2:])
```

Out[95]:
```
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

With DataFrames, combine_first does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass:

In [96]:
```
df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],'b': [np.nan, 2., np.nan, 6.],'c'
df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],'b': [np.nan, 3., 4., 6., 8.]})
df1
```

Out[96]:

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

In [97]:
```
df2
```

Out[97]:

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

In [98]:
```
df1.combine_first(df2)
```

Out[98]:

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

8.3 Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are alternatingly 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

I’ll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

In [99]:
```
data = pd.DataFrame(np.arange(6).reshape((2, 3)),index=pd.Index(['Ohio', 'Colorado'
data
```

Out[99]:

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

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

In [100…
```
result = data.stack()
result
```

Out[100…
```
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int32
```

From a hierarchically indexed Series, you can rearrange the data back into a DataFrame with unstack:

In [101…
```
result.unstack()
```

Out[101…

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

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

In [102…
```
result.unstack(0)
```

Out[102…

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

In [103…

```
result.unstack('state')
```

Out[103…

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

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

In [104…

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

Out[104…

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

In [105…

```
data2.unstack()
```

Out[105…

	a	b	c	d	e
one	0.0	1.0	2.0	3.0	NaN
two	NaN	NaN	4.0	5.0	6.0

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

In [106…

```
data2.unstack()
```

Out[106…

	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 [107…

```
data2.unstack().stack()
```

Out[107…

```
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 [108…

```
data2.unstack().stack(dropna=False)
```

Out[108…

```
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
```

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

In [109…

```
df = pd.DataFrame({'left': result, 'right': result + 5},columns=pd.Index(['left', '
df
```

Out[109…

	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 [110…

```
df.unstack('state')
```

Out[110…

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

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

In [111…

```
df.unstack('state').stack('side')
```

Out[111…

	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. Let’s load some example data and do a small amount of time series wrangling and other data cleaning:

In [112…

```
data = pd.read_csv('macrodata.csv')
data.head()
```

Out[112…

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	tbilrate	ur
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.98	139.7	2.82	
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.15	141.7	3.08	
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.35	140.5	3.82	
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.37	140.0	4.33	
4	1960.0	1.0	2847.699	1770.5	331.722	462.199	1955.5	29.54	139.6	3.50	

In [113…
```
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,name='date')
columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')
data = data.reindex(columns=columns)
data.index = periods.to_timestamp('D', 'end')
ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

We will look at PeriodIndex a bit more closely in Chapter 11. In short, it combines the year and quarter columns to create a kind of time interval type.

Now, ldata looks like:

In [114…
```
ldata[:10]
```

Out[114…

	date	item	value
0	1959-03-31 23:59:59.999999999	realgdp	2710.349
1	1959-03-31 23:59:59.999999999	infl	0.000
2	1959-03-31 23:59:59.999999999	unemp	5.800
3	1959-06-30 23:59:59.999999999	realgdp	2778.801
4	1959-06-30 23:59:59.999999999	infl	2.340
5	1959-06-30 23:59:59.999999999	unemp	5.100
6	1959-09-30 23:59:59.999999999	realgdp	2775.488
7	1959-09-30 23:59:59.999999999	infl	2.740
8	1959-09-30 23:59:59.999999999	unemp	5.300
9	1959-12-31 23:59:59.999999999	realgdp	2785.204

This is the so-called long format for multiple time series, or other observational data with two or more keys (here, our keys are date and item). Each row in the table represents a single observation.

Data is frequently stored this way in relational databases like MySQL, as a fixed schema (column names and data types) allows the number of distinct values in the item column to change as data is added to the table. In the previous example, date and item would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins. In some cases, the data may be more difficult to work with in this format; you might prefer to have a DataFrame containing one column per distinct item value indexed by timestamps in the date column. DataFrame's pivot method performs exactly this transformation:

In [115…
```
pivoted = ldata.pivot('date', 'item', 'value')
pivoted
```

Out[115…

item	infl	realgdp	unemp
date			
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2
...	...	...	...
2008-09-30 23:59:59.999999999	-3.16	13324.600	6.0
2008-12-31 23:59:59.999999999	-8.79	13141.920	6.9
2009-03-31 23:59:59.999999999	0.94	12925.410	8.1
2009-06-30 23:59:59.999999999	3.37	12901.504	9.2
2009-09-30 23:59:59.999999999	3.56	12990.341	9.6

203 rows × 3 columns

The first two values passed are the columns to be used respectively as the row and column index, then finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

In [116…

```
ldata['value2'] = np.random.randn(len(ldata))
ldata[:10]
```

Out[116…

	date	item	value	value2
0	1959-03-31 23:59:59.999999999	realgdp	2710.349	0.611860
1	1959-03-31 23:59:59.999999999	infl	0.000	-0.995615
2	1959-03-31 23:59:59.999999999	unemp	5.800	0.253419
3	1959-06-30 23:59:59.999999999	realgdp	2778.801	0.966742
4	1959-06-30 23:59:59.999999999	infl	2.340	-0.562305
5	1959-06-30 23:59:59.999999999	unemp	5.100	-0.523427
6	1959-09-30 23:59:59.999999999	realgdp	2775.488	0.522959
7	1959-09-30 23:59:59.999999999	infl	2.740	-1.296349
8	1959-09-30 23:59:59.999999999	unemp	5.300	-1.078606
9	1959-12-31 23:59:59.999999999	realgdp	2785.204	-0.360990

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

In [117…
```
pivoted = ldata.pivot('date', 'item')
pivoted[:5]
```

Out[117…

	value			value2		
item	**infl**	**realgdp**	**unemp**	**infl**	**realgdp**	**unemp**
date						
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8	-0.995615	0.611860	0.253419
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1	-0.562305	0.966742	-0.523427
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3	-1.296349	0.522959	-1.078606
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6	1.982862	-0.360990	-0.403380
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2	-0.434300	1.086619	-0.778612

In [118…
```
pivoted['value'][:5]
```

Out[118…

item	infl	realgdp	unemp
date			
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2

Note that pivot is equivalent to creating a hierarchical index using set_index followed by a call to unstack:

In [119…
```
unstacked = ldata.set_index(['date', 'item']).unstack('item')
```

In [120…
```
unstacked[:7]
```

Out[120…

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8	-0.995615	0.611860	0.253419
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1	-0.562305	0.966742	-0.523427
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3	-1.296349	0.522959	-1.078606
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6	1.982862	-0.360990	-0.403380
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2	-0.434300	1.086619	-0.778612
1960-06-30 23:59:59.999999999	0.14	2834.390	5.2	0.401136	1.343009	0.515892
1960-09-30 23:59:59.999999999	2.70	2839.022	5.6	0.533218	-0.654924	1.597632

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. Let’s look at an example:

In [121…

```
df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],'A': [1, 2, 3],'B': [4, 5, 6],'C':
df
```

Out[121…

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

The 'key' column may be a group indicator, and the 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 [122…

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

Out[122...

	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

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

In [123...
```
reshaped = melted.pivot('key', 'variable', 'value')
```

In [124...
```
reshaped
```

Out[124...

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

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 [125...
```
reshaped.reset_index()
```

Out[125...

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

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

In [126...
```
pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
```

Out[126…

	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

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

In [127…

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

Out[127…

	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

In [128…

```
pd.melt(df, value_vars=['key', 'A', 'B'])
```

Out[128…

	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

In [ ]: