Chapter 8. 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 convenient to analyze. This chapter focuses on tools to help combine, join, and rearrange data.

First, I introduce the concept of *hierarchical indexing* in pandas, which is used extensively in some of these operations. I then dig into the particular data manipulations. You can see various applied usages of these tools in Chapter 13.

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. Another way of thinking about it is that 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 [11]: data = pd.Series(np.random.uniform(size=9),
   ....:                  index=[["a", "a", "a", "b", "b", "c", "c", "d", "d
   ....:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [12]: data
Out[12]:
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
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 [13]: data.index
Out[13]:
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 [14]: data["b"]
Out[14]:
1    0.204560
3    0.567725
dtype: float64

In [15]: data["b":"c"]
Out[15]:
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
dtype: float64

In [16]: data.loc[["b", "d"]]
Out[16]:
b  1    0.204560
   3    0.567725
d  2    0.653177
   3    0.748907
dtype: float64
```

Selection is even possible from an “inner” level. Here I select all of the values having the value `2` from the second index level:

```
In [17]: data.loc[:, 2]
Out[17]:
a    0.316376
c    0.964515
```

```
d    0.653177
dtype: float64
```

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

```
In [18]: data.unstack()
Out[18]:
          1         2         3
a  0.929616  0.316376  0.183919
b  0.204560       NaN  0.567725
c  0.595545  0.964515       NaN
d       NaN  0.653177  0.748907
```

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

```
In [19]: data.unstack().stack()
Out[19]:
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64
```

`stack` and `unstack` will be explored in more detail later in Section 8.3, "Reshaping and Pivoting,".

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

```
In [20]: 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 [21]: frame
Out[21]:
     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 [22]: frame.index.names = ["key1", "key2"]

In [23]: frame.columns.names = ["state", "color"]

In [24]: frame
Out[24]:
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
```

These names supersede the `name` attribute, which is used only with single-level indexes.

CAUTION

Be careful to note that the index names `"state"` and `"color"` are not part of the row labels (the `frame.index` values).

You can see how many levels an index has by accessing its `nlevels` attribute:

```
In [25]: frame.index.nlevels
Out[25]: 2
```

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

```
In [26]: frame["Ohio"]
Out[26]:
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 also be created like this:

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

Reordering and Sorting Levels

At times you may need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The `swaplevel` method takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [27]: frame.swaplevel("key1", "key2")
Out[27]:
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` by default sorts the data lexicographically using all the index levels, but you can choose to use only a single level or a subset of levels to sort by passing the `level` argument. For example:

```
In [28]: frame.sort_index(level=1)
Out[28]:
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 [29]: frame.swaplevel(0, 1).sort_index(level=0)
Out[29]:
state       Ohio     Colorado
color      Green Red    Green
key2 key1
1    a         0   1        2
     b         6   7        8
2    a         3   4        5
     b         9  10       11
```

NOTE

Data 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 [30]: frame.groupby(level="key2").sum()
Out[30]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16

In [31]: frame.groupby(level="color", axis="columns").sum()
Out[31]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

We will discuss `groupby` in much more detail later in Chapter 10.

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 [32]: 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 [33]: frame
Out[33]:
   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 [34]: frame2 = frame.set_index(["c", "d"])

In [35]: frame2
Out[35]:
       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 by passing `drop=False` to `set_index`:

```
In [36]: frame.set_index(["c", "d"], drop=False)
Out[36]:
       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 [37]: frame2.reset_index()
Out[37]:
     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 in a number of ways:

`pandas.merge`

Connect 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`

Concatenate or "stack" objects together along an axis.

`combine_first`

Splice 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 particularly important in relational databases (e.g., SQL-based). The `pandas.merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [38]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"],
   ....:                     "data1": pd.Series(range(7), dtype="Int64")})

In [39]: df2 = pd.DataFrame({"key": ["a", "b", "d"],
   ....:                     "data2": pd.Series(range(3), dtype="Int64")})

In [40]: df1
Out[40]:
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6

In [41]: df2
Out[41]:
  key  data2
0   a      0
1   b      1
2   d      2
```

Here I am using pandas's `Int64` extension type for nullable integers, discussed in Section 7.3, "Extension Data Types,".

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 `pandas.merge` with these objects, we obtain:

```
In [42]: pd.merge(df1, df2)
Out[42]:
  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, `pandas.merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [43]: pd.merge(df1, df2, on="key")
Out[43]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

In general, the order of column output in `pandas.merge` operations is unspecified.

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

```
In [44]: df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c", "a", "a", "b"],
   ....:                     "data1": pd.Series(range(7), dtype="Int64")})

In [45]: df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
   ....:                     "data2": pd.Series(range(3), dtype="Int64")})

In [46]: pd.merge(df3, df4, left_on="lkey", right_on="rkey")
Out[46]:
  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, `pandas.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 [47]: pd.merge(df1, df2, how="outer")
Out[47]:
  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
6   c      3   <NA>
7   d   <NA>      2

In [48]: pd.merge(df3, df4, left_on="lkey", right_on="rkey", how="outer")
Out[48]:
  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
6    c      3  NaN   <NA>
7  NaN   <NA>    d      2
```

In an outer join, rows from the left or right DataFrame objects that do not match on keys in the other DataFrame will appear with NA values in the other DataFrame's columns for the nonmatching rows.

See Table 8-1 for a summary of the options for how.

Table 8-1. Different join types with the `how` argument

Option	Behavior
`how="inner"`	Use only the key combinations observed in both tables
`how="left"`	Use all key combinations found in the left table
`how="right"`	Use all key combinations found in the right table
`how="outer"`	Use all key combinations observed in both tables together

Many-to-many merges form the Cartesian product of the matching keys. Here's an example:

```
In [49]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
   ....:                     "data1": pd.Series(range(6), dtype="Int64")})

In [50]: df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
   ....:                     "data2": pd.Series(range(5), dtype="Int64")})

In [51]: df1
Out[51]:
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5

In [52]: df2
Out[52]:
  key  data2
0   a      0
1   b      1
2   a      2
3   b      3
4   d      4

In [53]: pd.merge(df1, df2, on="key", how="left")
```

```
Out[53]:
   key  data1  data2
0    b      0      1
1    b      0      3
2    b      1      1
3    b      1      3
4    a      2      0
5    a      2      2
6    c      3   <NA>
7    a      4      0
8    a      4      2
9    b      5      1
10   b      5      3
```

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 passed to the `how` keyword argument affects only the distinct key values appearing in the result:

```
In [54]: pd.merge(df1, df2, how="inner")
Out[54]:
  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 [55]: left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
   ....:                      "key2": ["one", "two", "one"],
   ....:                      "lval": pd.Series([1, 2, 3], dtype='Int64')})

In [56]: right = pd.DataFrame({"key1": ["foo", "foo", "bar", "bar"],
   ....:                       "key2": ["one", "one", "one", "two"],
   ....:                       "rval": pd.Series([4, 5, 6, 7], dtype='Int64'

In [57]: pd.merge(left, right, on=["key1", "key2"], how="outer")
Out[57]:
```

```
  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  foo  two     2  <NA>
3  bar  one     3     6
4  bar  two  <NA>     7
```

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.

CAUTION

When you’re joining columns on columns, the indexes on the passed DataFrame objects are discarded. If you need to preserve the index values, you can use `reset_index` to append the index to the columns.

A last issue to consider in merge operations is the treatment of overlapping column names. For example:

```
In [58]: pd.merge(left, right, on="key1")
Out[58]:
  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
```

While you can address the overlap manually (see the section [“Renaming Axis Indexes”]() for renaming axis labels), `pandas.merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [59]: pd.merge(left, right, on="key1", suffixes=("_left", "_right"))
Out[59]:
  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
```

See Table 8-2 for an argument reference on `pandas.merge`. The next section covers joining using the DataFrame’s row index.

Table 8-2. `pandas.merge` function arguments

Argument	Description
`left`	DataFrame to be merged on the left side.
`right`	DataFrame to be merged on the right side.
`how`	Type of join to apply: 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. Can be a single column name or a list of column names.
`right_on`	Analogous to `left_on` for `right` 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; `False` by default.
`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.

Argument	Description
`validate`	Verifies if the merge is of the specified type, whether one-to-one, one-to-many, or many-to-many. See the docstring for full details on the options.
`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 (row labels). 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 [60]: left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
   ....:                       "value": pd.Series(range(6), dtype="Int64")})

In [61]: right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])

In [62]: left1
Out[62]: 
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [63]: right1
Out[63]: 
   group_val
a        3.5
b        7.0

In [64]: pd.merge(left1, right1, left_on="key", right_index=True)
Out[64]: 
  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
```

NOTE

If you look carefully here, you will see that the index values for `left1` have been preserved, whereas in other examples above, the indexes of the input DataFrame objects are dropped. Because the index of `right1` is unique, this "many-to-one" merge (with the default `how="inner"` method) can preserve the index values from `left1` that correspond to rows in the output.

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

```
In [65]: pd.merge(left1, right1, left_on="key", right_index=True, how="outer
Out[65]:
  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 equivalent to a multiple-key merge:

```
In [66]: lefth = pd.DataFrame({"key1": ["Ohio", "Ohio", "Ohio",
   ....:                                "Nevada", "Nevada"],
   ....:                       "key2": [2000, 2001, 2002, 2001, 2002],
   ....:                       "data": pd.Series(range(5), dtype="Int64")})

In [67]: righth_index = pd.MultiIndex.from_arrays(
   ....:     [
   ....:         ["Nevada", "Nevada", "Ohio", "Ohio", "Ohio", "Ohio"],
   ....:         [2001, 2000, 2000, 2000, 2001, 2002]
   ....:     ]
   ....: )

In [68]: righth = pd.DataFrame({"event1": pd.Series([0, 2, 4, 6, 8, 10], dty
nt64",
   ....:                                            index=righth_index),
   ....:                        "event2": pd.Series([1, 3, 5, 7, 9, 11], dty
```

```
nt64",
   ....:                                                             index=righth_index)})

In [69]: lefth
Out[69]:
     key1  key2  data
0    Ohio  2000     0
1    Ohio  2001     1
2    Ohio  2002     2
3  Nevada  2001     3
4  Nevada  2002     4

In [70]: righth
Out[70]:
             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 [71]: pd.merge(lefth, righth, left_on=["key1", "key2"], right_index=True)
Out[71]:
     key1  key2  data  event1  event2
0    Ohio  2000     0       4       5
0    Ohio  2000     0       6       7
1    Ohio  2001     1       8       9
2    Ohio  2002     2      10      11
3  Nevada  2001     3       0       1

In [72]: pd.merge(lefth, righth, left_on=["key1", "key2"],
   ....:          right_index=True, how="outer")
Out[72]:
     key1  key2  data  event1  event2
0    Ohio  2000     0       4       5
0    Ohio  2000     0       6       7
1    Ohio  2001     1       8       9
2    Ohio  2002     2      10      11
3  Nevada  2001     3       0       1
4  Nevada  2002     4    <NA>    <NA>
4  Nevada  2000  <NA>       2       3
```

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

```
In [73]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=["a", "c", "e"],
   ....:                      columns=["Ohio", "Nevada"]).astype("Int64")

In [74]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                       index=["b", "c", "d", "e"],
   ....:                       columns=["Missouri", "Alabama"]).astype("Int6

In [75]: left2
Out[75]:
   Ohio  Nevada
a     1       2
c     3       4
e     5       6

In [76]: right2
Out[76]:
   Missouri  Alabama
b         7        8
c         9       10
d        11       12
e        13       14

In [77]: pd.merge(left2, right2, how="outer", left_index=True, right_index=T
Out[77]:
   Ohio  Nevada  Missouri  Alabama
a     1       2      <NA>     <NA>
b  <NA>    <NA>         7        8
c     3       4         9       10
d  <NA>    <NA>        11       12
e     5       6        13       14
```

DataFrame has a `join` instance method to simplify merging by index. It can also be used to combine many DataFrame objects having the same or similar indexes but nonoverlapping columns. In the prior example, we could have written:

```
In [78]: left2.join(right2, how="outer")
Out[78]:
   Ohio  Nevada  Missouri  Alabama
a     1       2      <NA>     <NA>
b  <NA>    <NA>         7        8
c     3       4         9       10
```

```
d  <NA>    <NA>        11        12
e     5       6        13        14
```

Compared with `pandas.merge`, DataFrame's `join` method performs a left join on the join keys by default. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [79]: left1.join(right1, on="key")
Out[79]:
  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
```

You can think of this method as joining data "into" the object whose `join` method was called.

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `pandas.concat` function described in the next section:

```
In [80]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:                        index=["a", "c", "e", "f"],
   ....:                        columns=["New York", "Oregon"])

In [81]: another
Out[81]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0

In [82]: left2.join([right2, another])
Out[82]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2      <NA>     <NA>       7.0     8.0
c     3       4         9       10       9.0    10.0
e     5       6        13       14      11.0    12.0

In [83]: left2.join([right2, another], how="outer")
Out[83]:
```

```
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2      <NA>     <NA>       7.0     8.0
c     3       4         9       10       9.0    10.0
e     5       6        13       14      11.0    12.0
b  <NA>    <NA>         7        8       NaN     NaN
d  <NA>    <NA>        11       12       NaN     NaN
f  <NA>    <NA>      <NA>     <NA>      16.0    17.0
```

Concatenating Along an Axis

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

```
In [84]: arr = np.arange(12).reshape((3, 4))

In [85]: arr
Out[85]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [86]: np.concatenate([arr, arr], axis=1)
Out[86]:
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 concerns:

- If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the values in common?
- Do the concatenated chunks of data need to be identifiable as such 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 questions. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [87]: s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")

In [88]: s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64")

In [89]: s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
```

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

```
In [90]: s1
Out[90]:
a    0
b    1
dtype: Int64

In [91]: s2
Out[91]:
c    2
d    3
e    4
dtype: Int64

In [92]: s3
Out[92]:
f    5
g    6
dtype: Int64

In [93]: pd.concat([s1, s2, s3])
Out[93]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: Int64
```

By default, `pandas.concat` works along `axis="index"`, producing another Series. If you pass `axis="columns"`, the result will instead be a DataFrame:

```
In [94]: pd.concat([s1, s2, s3], axis="columns")
Out[94]:
      0     1     2
a     0  <NA>  <NA>
b     1  <NA>  <NA>
c  <NA>     2  <NA>
d  <NA>     3  <NA>
e  <NA>     4  <NA>
f  <NA>  <NA>     5
g  <NA>  <NA>     6
```

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

```
In [95]: s4 = pd.concat([s1, s3])

In [96]: s4
Out[96]:
a    0
b    1
f    5
g    6
dtype: Int64

In [97]: pd.concat([s1, s4], axis="columns")
Out[97]:
      0  1
a     0  0
b     1  1
f  <NA>  5
g  <NA>  6

In [98]: pd.concat([s1, s4], axis="columns", join="inner")
Out[98]:
   0  1
a  0  0
b  1  1
```

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

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 [99]: result = pd.concat([s1, s1, s3], keys=["one", "two", "three"])

In [100]: result
Out[100]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: Int64

In [101]: result.unstack()
Out[101]:
          a     b     f     g
one       0     1  <NA>  <NA>
two       0     1  <NA>  <NA>
three  <NA>  <NA>     5     6
```

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

```
In [102]: pd.concat([s1, s2, s3], axis="columns", keys=["one", "two", "three
Out[102]:
    one   two  three
a     0  <NA>   <NA>
b     1  <NA>   <NA>
c  <NA>     2   <NA>
d  <NA>     3   <NA>
e  <NA>     4   <NA>
f  <NA>  <NA>      5
g  <NA>  <NA>      6
```

The same logic extends to DataFrame objects:

```
In [103]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=["a", "b", "c
   .....:                    columns=["one", "two"])

In [104]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=["a", "c"
   .....:                    columns=["three", "four"])

In [105]: df1
Out[105]:
   one  two
a    0    1
```

```
b    2    3
c    4    5

In [106]: df2
Out[106]:
   three  four
a      5     6
c      7     8

In [107]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
Out[107]:
  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
```

Here the `keys` argument is used to create a hierarchical index where the first level can be used to identify each of the concatenated DataFrame objects.

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

```
In [108]: pd.concat({"level1": df1, "level2": df2}, axis="columns")
Out[108]:
  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 [109]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"],
   .....:           names=["upper", "lower"])
Out[109]:
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 [110]: df1 = pd.DataFrame(np.random.standard_normal((3, 4)),
   .....:                    columns=["a", "b", "c", "d"])

In [111]: df2 = pd.DataFrame(np.random.standard_normal((2, 3)),
   .....:                    columns=["b", "d", "a"])

In [112]: df1
Out[112]:
          a         b         c         d
0  1.248804  0.774191 -0.319657 -0.624964
1  1.078814  0.544647  0.855588  1.343268
2 -0.267175  1.793095 -0.652929 -1.886837

In [113]: df2
Out[113]:
          b         d         a
0  1.059626  0.644448 -0.007799
1 -0.449204  2.448963  0.667226
```

In this case, you can pass `ignore_index=True`, which discards the indexes from each DataFrame and concatenates the data in the columns only, assigning a new default index:

```
In [114]: pd.concat([df1, df2], ignore_index=True)
Out[114]:
          a         b         c         d
0  1.248804  0.774191 -0.319657 -0.624964
1  1.078814  0.544647  0.855588  1.343268
2 -0.267175  1.793095 -0.652929 -1.886837
3 -0.007799  1.059626       NaN  0.644448
4  0.667226 -0.449204       NaN  2.448963
```

Table 8-3 describes the `pandas.concat` function arguments.

Table 8-3. `pandas.concat` function arguments

Argument	Description
`objs`	List or dictionary of pandas objects to be concatenated; this is the only required argument
`axis`	Axis to concatenate along; defaults to concatenating along rows (`axis="index"`)
`join`	Either `"inner"` or `"outer"` (`"outer"` by default); whether to intersect (inner) or union (outer) indexes along the other axes
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can be a list or array of arbitrary values, an array of tuples, or a list of arrays (if 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 an exception if so; by default (`False`) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation `axis`, instead produce 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 with indexes that overlap in full or in part. As a motivating example, consider NumPy's `where` function, which performs the array-oriented equivalent of an if-else expression:

```
In [115]: a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
   .....:               index=["f", "e", "d", "c", "b", "a"])

In [116]: b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
   .....:               index=["a", "b", "c", "d", "e", "f"])

In [117]: a
Out[117]:
f    NaN
e    2.5
d    0.0
c    3.5
b    4.5
a    NaN
dtype: float64

In [118]: b
Out[118]:
a    0.0
b    NaN
c    2.0
d    NaN
e    NaN
f    5.0
dtype: float64

In [119]: np.where(pd.isna(a), b, a)
Out[119]: array([0. , 2.5, 0. , 3.5, 4.5, 5. ])
```

Here, whenever values in `a` are null, values from `b` are selected, otherwise the non-null values from `a` are selected. Using `numpy.where` does not check whether the index labels are aligned or not (and does not even require the objects to be the same length), so if you want to line up values by index, use the Series `combine_first` method:

```
In [120]: a.combine_first(b)
Out[120]:
a    0.0
b    4.5
c    3.5
d    0.0
e    2.5
f    5.0
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 [121]: df1 = pd.DataFrame({"a": [1., np.nan, 5., np.nan],
   .....:                     "b": [np.nan, 2., np.nan, 6.],
   .....:                     "c": range(2, 18, 4)})

In [122]: df2 = pd.DataFrame({"a": [5., 4., np.nan, 3., 7.],
   .....:                     "b": [np.nan, 3., 4., 6., 8.]})

In [123]: df1
Out[123]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [124]: df2
Out[124]:
     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 [125]: df1.combine_first(df2)
Out[125]:
     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
```

The output of `combine_first` with DataFrame objects will have the union of all the column names.

8.3 Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are 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 [126]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                     index=pd.Index(["Ohio", "Colorado"], name="sta
   .....:                     columns=pd.Index(["one", "two", "three"],
   .....:                     name="number"))

In [127]: data
Out[127]:
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 [128]: result = data.stack()

In [129]: result
Out[129]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

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

```
In [130]: result.unstack()
Out[130]:
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 [131]: result.unstack(level=0)
Out[131]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [132]: result.unstack(level="state")
Out[132]:
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 subgroup:

```
In [133]: s1 = pd.Series([0, 1, 2, 3], index=["a", "b", "c", "d"], dtype="In

In [134]: s2 = pd.Series([4, 5, 6], index=["c", "d", "e"], dtype="Int64")

In [135]: data2 = pd.concat([s1, s2], keys=["one", "two"])

In [136]: data2
Out[136]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
```

```
     d    5
     e    6
dtype: Int64
```

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

```
In [137]: data2.unstack()
Out[137]:
        a     b  c  d     e
one     0     1  2  3  <NA>
two  <NA>  <NA>  4  5     6

In [138]: data2.unstack().stack()
Out[138]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64

In [139]: data2.unstack().stack(dropna=False)
Out[139]:
one  a       0
     b       1
     c       2
     d       3
     e    <NA>
two  a    <NA>
     b    <NA>
     c       4
     d       5
     e       6
dtype: Int64
```

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

```
In [140]: df = pd.DataFrame({"left": result, "right": result + 5},
   .....:                   columns=pd.Index(["left", "right"], name="side"))

In [141]: df
Out[141]:
```

```
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 [142]: df.unstack(level="state")
Out[142]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

As with `unstack`, when calling `stack` we can indicate the name of the axis to stack:

```
In [143]: df.unstack(level="state").stack(level="side")
Out[143]:
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 files is what is sometimes called *long* or *stacked* format. In this format, individual values are represented by a single row in a table rather than multiple values per row.

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

```
In [144]: data = pd.read_csv("examples/macrodata.csv")
```

```
In [145]: data = data.loc[:, ["year", "quarter", "realgdp", "infl", "unemp"]]

In [146]: data.head()
Out[146]:
   year  quarter   realgdp  infl  unemp
0  1959        1  2710.349  0.00    5.8
1  1959        2  2778.801  2.34    5.1
2  1959        3  2775.488  2.74    5.3
3  1959        4  2785.204  0.27    5.6
4  1960        1  2847.699  2.31    5.2
```

First, I use `pandas.PeriodIndex` (which represents time intervals rather than points in time), discussed in more detail in Chapter 11, to combine the `year` and `quarter` columns to set the index to consist of `datetime` values at the end of each quarter:

```
In [147]: periods = pd.PeriodIndex(year=data.pop("year"),
   .....:                          quarter=data.pop("quarter"),
   .....:                          name="date")

In [148]: periods
Out[148]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203)

In [149]: data.index = periods.to_timestamp("D")

In [150]: data.head()
Out[150]:
             realgdp  infl  unemp
date
1959-01-01  2710.349  0.00    5.8
1959-04-01  2778.801  2.34    5.1
1959-07-01  2775.488  2.74    5.3
1959-10-01  2785.204  0.27    5.6
1960-01-01  2847.699  2.31    5.2
```

Here I used the `pop` method on the DataFrame, which returns a column while deleting it from the DataFrame at the same time.

Then, I select a subset of columns and give the `columns` index the name `"item"`:

```
In [151]: data = data.reindex(columns=["realgdp", "infl", "unemp"])

In [152]: data.columns.name = "item"

In [153]: data.head()
Out[153]:
item          realgdp  infl  unemp
date
1959-01-01  2710.349  0.00    5.8
1959-04-01  2778.801  2.34    5.1
1959-07-01  2775.488  2.74    5.3
1959-10-01  2785.204  0.27    5.6
1960-01-01  2847.699  2.31    5.2
```

Lastly, I reshape with `stack`, turn the new index levels into columns with `reset_index`, and finally give the column containing the data values the name `"value"`:

```
In [154]: long_data = (data.stack()
   .....:              .reset_index()
   .....:              .rename(columns={0: "value"}))
```

Now, `ldata` looks like:

```
In [155]: long_data[:10]
Out[155]:
        date     item     value
0 1959-01-01  realgdp  2710.349
1 1959-01-01     infl     0.000
2 1959-01-01    unemp     5.800
3 1959-04-01  realgdp  2778.801
4 1959-04-01     infl     2.340
5 1959-04-01    unemp     5.100
6 1959-07-01  realgdp  2775.488
7 1959-07-01     infl     2.740
8 1959-07-01    unemp     5.300
9 1959-10-01  realgdp  2785.204
```

In this so-called *long* format for multiple time series, each row in the table represents a single observation.

Data is frequently stored this way in relational SQL databases, 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 [156]: pivoted = long_data.pivot(index="date", columns="item",
   .....:                           values="value")

In [157]: pivoted.head()
Out[157]:
item        infl   realgdp  unemp
date
1959-01-01  0.00  2710.349    5.8
1959-04-01  2.34  2778.801    5.1
1959-07-01  2.74  2775.488    5.3
1959-10-01  0.27  2785.204    5.6
1960-01-01  2.31  2847.699    5.2
```

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 [158]: long_data["value2"] = np.random.standard_normal(len(long_data))

In [159]: long_data[:10]
Out[159]:
        date     item     value    value2
0 1959-01-01  realgdp  2710.349  0.802926
1 1959-01-01     infl     0.000  0.575721
2 1959-01-01    unemp     5.800  1.381918
3 1959-04-01  realgdp  2778.801  0.000992
4 1959-04-01     infl     2.340 -0.143492
5 1959-04-01    unemp     5.100 -0.206282
6 1959-07-01  realgdp  2775.488 -0.222392
7 1959-07-01     infl     2.740 -1.682403
8 1959-07-01    unemp     5.300  1.811659
9 1959-10-01  realgdp  2785.204 -0.351305
```

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

```
In [160]: pivoted = long_data.pivot(index="date", columns="item")

In [161]: pivoted.head()
Out[161]:
            value                    value2
item         infl   realgdp unemp      infl   realgdp     unemp
date
1959-01-01  0.00  2710.349   5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801   5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488   5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204   5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699   5.2 -0.615939  0.498327  0.174072

In [162]: pivoted["value"].head()
Out[162]:
item         infl   realgdp  unemp
date
1959-01-01  0.00  2710.349    5.8
1959-04-01  2.34  2778.801    5.1
1959-07-01  2.74  2775.488    5.3
1959-10-01  0.27  2785.204    5.6
1960-01-01  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 [163]: unstacked = long_data.set_index(["date", "item"]).unstack(level="i

In [164]: unstacked.head()
Out[164]:
            value                    value2
item         infl   realgdp unemp      infl   realgdp     unemp
date
1959-01-01  0.00  2710.349   5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801   5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488   5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204   5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699   5.2 -0.615939  0.498327  0.174072
```

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

In [167]: df
Out[167]:
   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 [168]: melted = pd.melt(df, id_vars="key")

In [169]: melted
Out[169]:
   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 [170]: reshaped = melted.pivot(index="key", columns="variable",
   .....:                         values="value")

In [171]: reshaped
Out[171]:
```

```
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 [172]: reshaped.reset_index()
Out[172]:
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 [173]: pd.melt(df, id_vars="key", value_vars=["A", "B"])
Out[173]:
   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 [174]: pd.melt(df, value_vars=["A", "B", "C"])
Out[174]:
  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 [175]: pd.melt(df, value_vars=["key", "A", "B"])
Out[175]:
  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
```

8.4 Conclusion

Now that you have some pandas basics for data import, cleaning, and reorganization under your belt, we are ready to move on to data visualization with matplotlib. We will return to explore other areas of pandas later in the book when we discuss more advanced analytics.