CHAPTER 6

Data Wrangling

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.

6.1 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 [2]:
```
import pandas as pd
import numpy as np
```

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

Out[2]:
```
a  1    0.319425
   2    0.851751
   3    0.265340
b  1    0.727848
   3    0.551000
c  1   -1.521145
   2    0.525656
d  2    2.226440
   3    0.235736
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 [3]:
```
data.index
```

Out[3]:
```
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]])
```

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

In [4]:
```
data['b']
```

Out[4]:
```
1    0.727848
3    0.551000
dtype: float64
```

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

Out[5]:
```
b  1    0.727848
   3    0.551000
c  1   -1.521145
   2    0.525656
dtype: float64
```

In [6]:
```
 data.loc[['b', 'd']]
```

Out[6]:
```
b  1    0.727848
   3    0.551000
d  2    2.226440
   3    0.235736
dtype: float64
```

Selection is even possible from an "inner" level:

In [7]:
```
data.loc[:, 2]
```

Out[7]:
```
a    0.851751
c    0.525656
d    2.226440
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 [8]:
```
data.unstack()
```

Out[8]:

	1	2	3
a	0.319425	0.851751	0.265340
b	0.727848	NaN	0.551000
c	-1.521145	0.525656	NaN
d	NaN	2.226440	0.235736

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

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

Out[9]:
```
a  1    0.319425
   2    0.851751
   3    0.265340
b  1    0.727848
   3    0.551000
c  1   -1.521145
   2    0.525656
d  2    2.226440
   3    0.235736
dtype: float64
```

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

In [25]:
```
frame = pd.DataFrame(np.arange(12).reshape((4, 3)), index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                     columns=[['Ohio', 'Ohio', 'Colorado'],['Green', 'Red', 'Green']])
frame
```

Out[25]:

		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 [26]:
```
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']
frame
```

Out[26]:

	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

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

In [27]:
```
frame['Ohio']
```

Out[27]:

	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 [14]:
```
pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                          names=['state', 'color'])
```

Out[14]:
```
MultiIndex(levels=[['Colorado', 'Ohio'], ['Green', 'Red']],
           labels=[[1, 1, 0], [0, 1, 0]],
           names=['state', 'color'])
```

6.1.1 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 [28]:
```
frame
```

Out[28]:

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

In [29]:
```
frame.swaplevel('key1', 'key2')
```

Out[29]:

	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 [30]:
```
frame.sort_index(level=1)
```

Out[30]:

	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 [31]:
```
frame.swaplevel(0, 1).sort_index(level=0)
```

Out[31]:

	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

6.1.2 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 [32]:
```
frame
```

Out[32]:

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

In [33]:
```
frame.sum(level='key2')
```

Out[33]:

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

In [34]:
```
frame.sum(level='color', axis=1)
```

Out[34]:

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

6.1.3 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 [20]:
```
frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1), 'c': ['one', 'one', 'one', 'two', 'two',
'two', 'two'],
                      'd': [0, 1, 2, 0, 1, 2, 3]})
frame
```

Out[20]:

	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 [21]:
```
frame2 = frame.set_index(['c', 'd'])
frame2
```

Out[21]:

		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 [22]:
```
frame.set_index(['c', 'd'], drop=False)
```

Out[22]:

		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 [23]:
```
frame2.reset_index()
```

Out[23]:

	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

6.2 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.

6.2.1 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 data. Let's start with a simple example:

In [35]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],'data2': range(3)})
df1
```

Out[35]:

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

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 [36]:
```
pd.merge(df1, df2)
```

Out[36]:

	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

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, though:

In [37]:
```
pd.merge(df1, df2, on='key')
```

Out[37]:

	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

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

In [38]:
```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

Out[38]:

	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

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]:

	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

In [40]:
```
pd.merge(df1, df2, how='left')
```

Out[40]:

	data1	key	data2
0	0	b	1.0
1	1	b	1.0
2	2	a	0.0
3	3	c	NaN
4	4	a	0.0
5	5	a	0.0
6	6	b	1.0

In [41]:
```
pd.merge(df1, df2, how='right')
```

Out[41]:

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

See Table 6-1 for a summary of the options for *how*.

Table 6.1: 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 [3]:
```
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)})
df1
```

Out[3]:

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

In [4]:
```
df2
```

Out[4]:

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

In [5]:
```
pd.merge(df1, df2, on='key', how='left')
```

Out[5]:

	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

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 [6]:
```
pd.merge(df1, df2, how='inner')
```

Out[6]:

	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

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

In [7]:
```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'], 'key2': ['one', 'two', 'one'],
                      'lval': [1, 2, 3]})
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],'key2': ['one', 'one', 'one', 'two'],
                       'rval': [4, 5, 6, 7]})
```

In [8]:
```
left
```

Out[8]:

	key1	key2	lval
0	foo	one	1
1	foo	two	2
2	bar	one	3

In [9]:
```
right
```

Out[9]:

	key1	key2	rval
0	foo	one	4
1	foo	one	5
2	bar	one	6
3	bar	two	7

In [10]:
```
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

Out[10]:

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

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

In [11]:
```
pd.merge(left, right, on='key1')
```

Out[11]:

	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 [12]:
```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Out[12]:

	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 6-2 for an argument reference on **merge**. Joining using the DataFrame's row index is the subject of the next section.

Table 6.2: 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.

6.2.2 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 [13]:
```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'], 'value': range(6)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
left1
```

Out[13]:

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

In [14]:
```
right1
```

Out[14]:

	group_val
a	3.5
b	7.0

In [15]:
```
pd.merge(left1, right1, left_on='key', right_index=True)
```

Out[15]:

	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 [16]:
```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

Out[16]:

	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

6.2.3 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 [17]:
```
arr = np.arange(12).reshape((3, 4))
arr
```

Out[17]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [18]:
```
np.concatenate([arr, arr], axis=1)
```

Out[18]:
```
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. We'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

In [19]:
```
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 [20]:
```
pd.concat([s1, s2, s3])
```

Out[20]:
```
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 [21]:
```
pd.concat([s1, s2, s3], axis=1)
```

Out[21]:

	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 [22]:
```
s4 = pd.concat([s1, s3])
s4
```

Out[22]:
```
a    0
b    1
f    5
g    6
dtype: int64
```

In [23]:
```
pd.concat([s1, s4], axis=1)
```

Out[23]:

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

In [25]:
```
pd.concat([s1, s4], axis=1, join='inner')
```

Out[25]:

	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 [26]:
```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

Out[26]:

	0	1
a	0.0	0.0
c	NaN	NaN
b	1.0	1.0
e	NaN	NaN

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 [28]:
```
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
result
```

Out[28]:
```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

In [29]:
```
result.unstack()
```

Out[29]:

	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 [30]:
```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

Out[30]:

	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 [31]:
```
df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],columns=['one', 'two'])
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],columns=['three', 'four'])
df1
```

Out[31]:

	one	two
a	0	1
b	2	3
c	4	5

In [32]:
```
df2
```

Out[32]:

	three	four
a	5	6
c	7	8

In [33]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

Out[33]:

	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 [34]:
```
pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

Out[34]:

	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

See Table 6-3 for arguments reference on **concat**.

Table 6.3: 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 (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 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