

03.6 Concat And Append

May 8, 2018

This notebook contains an excerpt from the [Python Data Science Handbook](#) by Jake VanderPlas; the content is available [on GitHub](#).

1 Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. Series and DataFrames are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Here we'll take a look at simple concatenation of Series and DataFrames with the `pd.concat` function; later we'll dive into more sophisticated in-memory merges and joins implemented in Pandas.

We begin with the standard imports:

```
In [1]: import pandas as pd
import numpy as np
```

For convenience, we'll define this function which creates a DataFrame of a particular form that will be useful below:

```
In [2]: def make_df(cols, ind):
        """Quickly make a DataFrame"""
        data = {c: [str(c) + str(i) for i in ind]
                 for c in cols}
        return pd.DataFrame(data, ind)

        # example DataFrame
        make_df('ABC', range(3))
```

```
Out[2]:
```

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class that allows us to display multiple DataFrames side by side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

```
In [3]: class display(object):
        """Display HTML representation of multiple objects"""
        template = """<div style="float: left; padding: 10px;">
        <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
        </div>"""
        def __init__(self, *args):
            self.args = args

        def _repr_html_(self):
            return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                              for a in self.args)

        def __repr__(self):
            return '\n\n'.join(a + '\n' + repr(eval(a))
                                for a in self.args)
```

The use of this will become clearer as we continue our discussion in the following section.

1.1 Recall: Concatenation of NumPy Arrays

Concatenation of Series and DataFrame objects is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function as discussed in [The Basics of NumPy Arrays]. Recall that with it, you can combine the contents of two or more arrays into a single array:

```
In [4]: x = [1, 2, 3]
        y = [4, 5, 6]
        z = [7, 8, 9]
        np.concatenate([x, y, z])

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

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
In [5]: x = [[1, 2],
            [3, 4]]
        np.concatenate([x, x], axis=1)

Out[5]: array([[1, 2, 1, 2],
              [3, 4, 3, 4]])
```

1.2 Simple Concatenation with `pd.concat`

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of Series or DataFrame objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
In [6]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
        ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
        pd.concat([ser1, ser2])
```

```
Out[6]: 1    A
        2    B
        3    C
        4    D
        5    E
        6    F
        dtype: object
```

It also works to concatenate higher-dimensional objects, such as DataFrames:

```
In [7]: df1 = make_df('AB', [1, 2])
        df2 = make_df('AB', [3, 4])
        display('df1', 'df2', 'pd.concat([df1, df2])')
```

```
Out[7]: df1
         A  B
1  A1  B1
2  A2  B2

df2
         A  B
3  A3  B3
4  A4  B4

pd.concat([df1, df2])
         A  B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
```

By default, the concatenation takes place row-wise within the DataFrame (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
In [8]: df3 = make_df('AB', [0, 1])
        df4 = make_df('CD', [0, 1])
        display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
```

```
Out[8]: df3
         A  B
0  A0  B0
```

```

1  A1  B1

df4
   C  D
0  C0 D0
1  C1 D1

pd.concat([df3, df4], axis='col')
   A  B  C  D
0  A0 B0 C0 D0
1  A1 B1 C1 D1

```

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

1.2.1 Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

```

In [9]: x = make_df('AB', [0, 1])
        y = make_df('AB', [2, 3])
        y.index = x.index # make duplicate indices!
        display('x', 'y', 'pd.concat([x, y])')

```

```

Out [9]: x
   A  B
0  A0 B0
1  A1 B1

y
   A  B
0  A2 B2
1  A3 B3

pd.concat([x, y])
   A  B
0  A0 B0
1  A1 B1
0  A2 B2
1  A3 B3

```

Notice the repeated indices in the result. While this is valid within DataFrames, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the repeats as an error If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to `True`, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```
In [10]: try:
          pd.concat([x, y], verify_integrity=True)
        except ValueError as e:
          print("ValueError:", e)
```

ValueError: Indexes have overlapping values: [0, 1]

Ignoring the index Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting Series:

```
In [11]: display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

```
Out[11]: x
      A  B
0  A0  B0
1  A1  B1

y
      A  B
0  A2  B2
1  A3  B3

pd.concat([x, y], ignore_index=True)
      A  B
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3
```

Adding MultiIndex keys Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```
In [12]: display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

```
Out[12]: x
      A  B
0  A0  B0
1  A1  B1

y
      A  B
0  A2  B2
1  A3  B3

pd.concat([x, y], keys=['x', 'y'])
      A  B
```

```

x 0  A0  B0
   1  A1  B1
y 0  A2  B2
   1  A3  B3

```

The result is a multiply indexed DataFrame, and we can use the tools discussed in [Hierarchical Indexing] -Hierarchical-Indexing.ipynb) to transform this data into the representation we're interested in.

1.2.2 Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating DataFrames with shared column names. In practice, data from different sources might have different sets of column names, and `pd.concat` offers several options in this case. Consider the concatenation of the following two DataFrames, which have some (but not all!) columns in common:

```

In [13]: df5 = make_df('ABC', [1, 2])
         df6 = make_df('BCD', [3, 4])
         display('df5', 'df6', 'pd.concat([df5, df6])')

```

```

Out[13]: df5
         A  B  C
1  A1  B1  C1
2  A2  B2  C2

```

```

df6
         B  C  D
3  B3  C3  D3
4  B4  C4  D4

```

```

pd.concat([df5, df6])
         A  B  C  D
1  A1  B1  C1  NaN
2  A2  B2  C2  NaN
3  NaN  B3  C3  D3
4  NaN  B4  C4  D4

```

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the `join` and `join_axes` parameters of the `concatenate` function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```

In [14]: display('df5', 'df6',
                 "pd.concat([df5, df6], join='inner')")

```

```

Out[14]: df5
         A  B  C
1  A1  B1  C1
2  A2  B2  C2

```

```
df6
   B  C  D
3 B3 C3 D3
4 B4 C4 D4

pd.concat([df5, df6], join='inner')
   B  C
1 B1 C1
2 B2 C2
3 B3 C3
4 B4 C4
```

Another option is to directly specify the index of the remaining columns using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

```
In [15]: display('df5', 'df6',
                "pd.concat([df5, df6], join_axes=[df5.columns])")
```

```
Out[15]: df5
   A  B  C
1 A1 B1 C1
2 A2 B2 C2

df6
   B  C  D
3 B3 C3 D3
4 B4 C4 D4

pd.concat([df5, df6], join_axes=[df5.columns])
   A  B  C
1  A1 B1 C1
2  A2 B2 C2
3  NaN B3 C3
4  NaN B4 C4
```

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data.

1.2.3 The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call `df1.append(df2)`:

```
In [16]: display('df1', 'df2', 'df1.append(df2)')
```

```
Out[16]: df1
   A  B
1  A1 B1
2  A2 B2

df2
   A  B
3  A3 B3
4  A4 B4

df1.append(df2)
   A  B
1  A1 B1
2  A2 B2
3  A3 B3
4  A4 B4
```

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the `append()` method in Pandas does not modify the original object—instead it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index *and* data buffer. Thus, if you plan to do multiple append operations, it is generally better to build a list of DataFrames and pass them all at once to the `concat()` function.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the ["Merge, Join, and Concatenate"](#) section of the Pandas documentation.