3.3-Combining Datasets: Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

For convenience, we will start by redefining the `display()` functionality from the previous section:

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

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)
```

Table of Contents

1 Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of `Series` and `Dataframe` s. As we will see, these let you efficiently link data from different sources.

↥ back to top

2 Categories of Joins

The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like base::merge.data.frame in R). The reason for this is careful algorithmic design and internal layout of the data in DataFrame.

See the cookbook for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a comparison with SQL.

pandas provides a single function, merge, as the entry point for all standard database join operations between DataFrame objects:

```
merge(left, right, how='inner', on=None, left_on=None, right_on=None,
      left_index=False, right_index=False, sort=True,
      suffixes=('_x', '_y'), copy=True, indicator=False)
```

Here's a description of what each argument is for:

- left: A DataFrame object
- right: Another DataFrame object
- on: Columns (names) to join on. Must be found in both the left and right DataFrame objects. If not passed and left_index and right_index are False, the intersection of the columns in the DataFrames will be inferred to be the join keys
- left_on: Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame

- right_on: Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- left_index: If True, use the index (row labels) from the left DataFrame as its join key(s). In the case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame
- right_index: Same usage as left_index for the right DataFrame
- how: One of 'left', 'right', 'outer', 'inner'. Defaults to inner. See below for more detailed description of each method
- sort: Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve performance substantially in many cases
- suffixes: A tuple of string suffixes to apply to overlapping columns. Defaults to ('_x', '_y').
- copy: Always copy data (default True) from the passed DataFrame objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.
- indicator: Add a column to the output DataFrame called _merge with information on the source of each row. _merge is Categorical-type and takes on a value of left_only for observations whose merge key only appears in 'left' DataFrame, right_only for observations whose merge key only appears in 'right' DataFrame, and both if the observation's merge key is found in both.

↥ back to top

3 Specification of the Merge Key

We've already seen the default behavior of `pd.merge()` : it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

In [2]:

```
# Lets create dataset first
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
```

3.1 The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

In [3]:

```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

Out[3]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [ ]:

```
```

This option works only if both the left and right `DataFrame` s have the specified column name.

3.2 The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

In [4]:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
```

Out[4]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column that we can drop if desired–for example, by using the `drop()` method of `DataFrame` s:

In [5]:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

Out[5]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

3.3 The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

In [6]:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

Out[6]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()` :

In [7]:

```
display('df1a', 'df2a',
```

```
        "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

Out[7]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

For convenience, `DataFrame` s implement the `join()` method, which performs a merge that defaults to joining on indices:

In [8]:

```
display('df1a', 'df2a', 'df1a.join(df2a)')
```

Out[8]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

df1a.join(df2a)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

In [9]:

```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
```

Out[9]: df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1a, df3, left_index=True, right_on='name')

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the "Merge, Join, and Concatenate" section of the Pandas documentation.

↥ back to top

4 Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

In [10]:
```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

Out[10]: df6

	name	food
0	Peter	fish
1	Paul	beans

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7)

	name	food	drink
0	Mary	bread	wine

	name	food
2	Mary	bread

Here we have merged two datasets that have only a single "name" entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to `"inner"`:

In [11]:
```
pd.merge(df6, df7, how='inner')
```

Out[11]:

	name	food	drink
0	Mary	bread	wine

Other options for the `how` keyword are `'outer'`, `'left'`, and `'right'`. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

In [12]:
```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

Out[12]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

In [13]:
```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
```

Out[13]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='left')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding join types.

↥ back to top

5 Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input `DataFrame` s have conflicting column names. Consider this example:

In [14]:
```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

Out[14]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name")

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

In [15]:
```
display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
```

Out[15]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1

	name	rank_L	rank_R
2	Lisa	3	4
3	Sue	4	2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

↥ back to top

Great Job!
