Chapter 17

Joins

Contents

1 Helpful Table / Review

- When we started working in Pandas, we said one of the difficult parts was keeping track of what was being returned by an object. To help with this process, I've created the following, Table 17.1, which maps structure and operation to outcome.
- I personally don't have all these memorized, only a few which allow me to quickly deal with problems.

Data	Operator	Example	Result	Detail
Series	**Aggregation**	`.sum()`	Number	#
	With `.agg`	`.agg('sum')`	Number	#
	With `.agg` in list	`.agg(['sum'])`	Series	Row Index Agg
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	Series	Row Index Col Name
	With `.agg` in dict (lists)	`.agg(['sum', 'count'])`	N/A	Operation not allowed
df	Aggregation	`.sum()`	Series	Row Index Col Name
	With `.agg`	`.agg('sum')`	Series	Row Index Col Name
	With `.agg` in list	`.agg(['sum'])`	df	Row Index Agg
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	Series	Row Index Col Name
	With `.agg` in dict (list)	`.agg({'col1': ['sum']})`	df	Row Index Agg, possible Nulls
groupby	Aggregation	`.sum()`	df	Cols single-level idx
	With `.agg`	`.agg('sum')`	df	Cols single-level idx
	With `.agg` in list	`.agg(['sum'])`	df	Cols multiindex
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	df	Cols single-level idx
	With `.agg` in dict (list)	`.agg({'col1': ['sum']})`	df	Cols multiindex

Table 17.1: Pandas common operations and their results. Bolded are recommended forms.

2 Merging data in Pandas

- To merge DataFrames in Pandas we use the `pd.merge` command.
- The basic structure of merging is the same as in SQL. We need to identify (a) which column(s) we wish to merge on and (b) what type of merge we wish to do.
- There is one wrench that gets thrown into this, however, which is that Pandas requires you to identify if the column(s) you are merging on are part of an index or not.
- In terms of the *type* of merges, they are similar to SQL: left, inner, outer, right and cross are all done the same.
- Let's start with merging two datasets without an index, as demonstrated by the following example:

```
>>> class1 = pd.DataFrame({"sname": ['John', 'Jim', 'Kyle']
    , "grade": ['A', 'A', 'C']})

>>> class2 = pd.DataFrame({"sname": ['John', 'Jim', 'Ashley']
    , "grade": ['A', 'B', 'F']})

>>> pd.merge(class1, class2, on='sname', how='left')
  sname grade_x grade_y
0  John       A       A
1   Jim       A       B
2  Kyle       C     NaN
```

- We call the function using `pd.merge` and then provide it the DataFrames being merged. In this case we provided two DataFrames, class1 and class2. The first DataFrame is considered the "left" DataFrame and the second is considered the "right" DataFrame. We can specify "left" and "right" as parameters if we want to be pedantic `pd.merge(left=class1, right=class2, on='sname', how='l`
- The merge type, `how`, accepts any standard join: left, inner, outer, right and cross as a string.
- We use the `on` parameter to state which column(s) we are merging on. If the columns have the same name then we simply put them in the `on` parameter within the method. If we have more than one column to merge on, we can specify the columns in a list:

```
>>> pd.merge(class1, class2, on=['sname', 'grade'], how='inner')
  sname grade
0  John     A
```

- If the columns are named different things, then we use the "left_on" and "right_on" operators to do the merge:

```
>>> class1T = class1.rename(columns={ 'sname' : 's2'})

>>> pd.merge(class1T, class2, left_on='s2', right_on='sname', how = 'left')
     s2 grade_x sname grade_y
0  John       A  John       A
1   Jim       A   Jim       B
2  Kyle       C   NaN     NaN
```

- One interesting thing that pandas can do is create an indicator which tells you how the row came into the resulting dataset, called _merge, as in the example below:

```
>>> pd.merge(class1, class2, on='sname', how='outer', indicator=True)
    sname grade_x grade_y      _merge
0    John       A       A        both
1     Jim       A       B        both
2    Kyle       C     NaN   left_only
3  Ashley     NaN       F  right_only
```

- Instead of calling merge directly from the pandas module, you can also call it as a method from a DataFrame. When doing this, the calling DataFrame is considered the left DataFrame:

```
>>> class1.merge( class2, how='left', on='sname')
  sname grade_x grade_y
0  John       A       A
1   Jim       A       B
2  Kyle       C     NaN
```

  I prefer to call pd.merge rather than using the above notation. I find it a bit cleaner.

- To do a cross-join in pandas (only available in versions greater than 1.5), you state the merge type as cross and do not put in any on condition.

```
>>> pd.merge(class1, class2, how='cross')
  sname_x grade_x sname_y grade_y
0    John       A    John       A
1    John       A     Jim       B
2    John       A  Ashley       F
3     Jim       A    John       A
4     Jim       A     Jim       B
5     Jim       A  Ashley       F
6    Kyle       C    John       A
7    Kyle       C     Jim       B
8    Kyle       C  Ashley       F
```

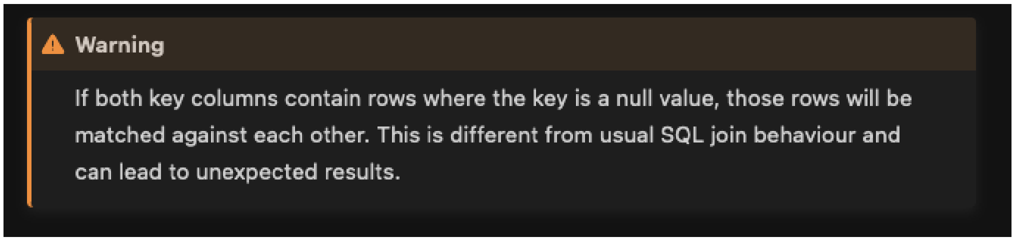
- **Index Merging:** In all of the examples above the data which was being merged on was stored as a value and was not a part of the index. If the column being merged on is in an index in one of the DataFrames then instead of using left_on and right_on, the parameters left_index and right_index need to be used, where a boolean True/False is given in the function. Consider the following example:

```
>>> class1idx = class1.set_index('sname')

>>> pd.merge(class1idx, class2, left_index=True, right_on = 'sname', how='left')
    grade_x sname grade_y
0.0       A  John       A
1.0       A   Jim       B
NaN       C  Kyle     NaN
```

  The first command in the above example changes the column "sname" in class1 to an index.

- **BIG THING:** When merging data with pandas *Nulls will match!* This is unlike SQL which has a consistent treatment of Null values. Per the pandas documentation:


Warning

If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

Figure 17.1: Null treatment in merges

3 Complex Join Conditions

- In all of the above examples we had simple equality joins where we wanted to match one column to its exact match within another column in a different DataFrame.
- However, there are many situations where a merge needs to be completed based on a more complex join condition, such as an inequality ($\geq$).
- Pandas, sadly, doesn't provide an easy method to implement non-equality join conditions. This means that when we join, we must either create a cross join style merge and then remove those rows that fail our actual join condition or use an equality join followed by the same filtering method.
- Let's return to our class tables and say that we want to join rows that have names that *don't* match. For example, I wish to create a dataset which allows me to compare each person against everyone else in the same class:

```
>>> d_1 = pd.merge( class1, class1, how='cross')

>>> d_1 = d_1.loc[(d_1.loc[:, 'sname_x'] != d_1.loc[:, 'sname_y']), :]

>>> d_1
  sname_x grade_x sname_y grade_y
1    John       A     Jim       A
2    John       A    Kyle       C
3     Jim       A    John       A
5     Jim       A    Kyle       C
6    Kyle       C    John       A
7    Kyle       C     Jim       A
```

In this case we implemented our more complex join condition *after* we did a cross-join style merge.

4 Stacking Data

- If we have a dataset and wish to stack or append it to another data set (similar to SQL's UNION or UNION ALL) we can use the "concat" operator. This operator takes a DataFrame and then puts multiple copies of the data back-to-back in a specified manner.
- Let's look at the following example:

```
>>> pd.concat([class1,class2])
    sname grade
0    John     A
1     Jim     A
2    Kyle     C
0    John     A
1     Jim     B
2  Ashley     F
```

The concat function, which is in the main pandas library, like merge, takes in data objects and then returns those data objects combined. There are two primary ways that concat is used, the first is above, in which case we wish to stack vertically.

- The concat method can also stack data frames horizontally. For example:

```
>>> pd.concat([class1, class2], axis=1)
  sname grade   sname grade
0  John     A    John     A
1   Jim     A     Jim     B
2  Kyle     C  Ashley     F
```

In the above example, the parameter "axis=1" was added. This parameter tells the concat method to stack the data along columns, rather than along rows. The default behavior is, unsurprisingly, "axis=0" which is the behavior in the previous example.

- A BIG difference between how pandas does concatenation and how relational databases do concatenation is that the columns in pandas are put in **name-alignment**. In other words, only columns which have the same name are matched together. Consider the following example:

```
>>> print("## Note that this is just class2 with a new column name and a new order")
## Note that this is just class2 with a new column name and a new order

>>> class3 = pd.DataFrame({"grade2": ['A', 'B', 'F']
    , "sname": ['John', 'Jim', 'Ashley'] })

>>> pd.concat([class1, class3])
    sname grade grade2
0    John     A    NaN
1     Jim     A    NaN
2    Kyle     C    NaN
0    John   NaN      A
1     Jim   NaN      B
2  Ashley   NaN      F
```

In the example above we see that grade is filled in with "NaN" values for data which was taken from the second DataFrame while grade2 contains "NaN" values for those observations taken from the first DataFrame.

Note also that the columns class1 and class3 were *not* in the same order and the function aligned those columns to those with similar names. In other words, this only appends columns which have the same name.

- One parameter of interest is the parameter "join" which defines which columns to return. If join is set to "inner" then only those columns in both DataFrames are included in the returned DataFrame while if "outer" is set, all columns are returned. Consider the following examples:

```
>>> class4 = class2.copy()

>>> class4.loc[:, 'test']  = 1

>>> pd.concat([class2, class4], join='inner')
    sname grade
0    John     A
1     Jim     B
2  Ashley     F
0    John     A
1     Jim     B
2  Ashley     F

>>> pd.concat([class2, class4], join='outer')
    sname grade  test
0    John     A   NaN
1     Jim     B   NaN
2  Ashley     F   NaN
0    John     A   1.0
1     Jim     B   1.0
2  Ashley     F   1.0
```

5 Lags and Leads

- A common operation with a DataFrame is to get the previous of next value within a Series. Generally called "lag" and "lead", these operations are done with the `shift` operator, which works on both Series and DataFrames.
- This operator takes in a number which represents how far back (or forward) in the DataFrame to step to get a value.
- Looking at the MTA data set we can use this information to get the previous hour's information:

```
>>> dfMTAC = dfMTA.loc[(dfMTA.loc[:, 'plaza'] == 1) & (dfMTA.loc[:, 'direction'] == 'I'), :]

>>> dfMTAC = dfMTAC.sort_values(['mtadt', 'hr'])

>>> dfMTAC.loc[:, 'pvsCash'] = dfMTAC.loc[:, 'vehiclescash'].shift(1)

>>> dfMTAC.loc[:, 'nxtCash'] = dfMTAC.loc[:, 'vehiclescash'].shift(-1)

>>> dfMTAC.head()
        plaza      mtadt  hr  ... vehiclescash  pvsCash  nxtCash
103440      1 2010-01-01   0  ...          474      NaN    717.0
103442      1 2010-01-01   1  ...          717    474.0    664.0
103444      1 2010-01-01   2  ...          664    717.0    595.0
103446      1 2010-01-01   3  ...          595    664.0    547.0
103448      1 2010-01-01   4  ...          547    595.0    450.0

[5 rows x 8 columns]
```

- In this case the "1" argument in the shift parameter tells Pandas to shift the dataset one row in the forward (or down) direction. In other words, positive values generate lags and negative values generate leads.

- The order of the rows is set by the `sort_values` command previous in the script. Once the order is set, the shift command steps back a row and the method with the `loc` then sets the values.

- The `shift` operator can also be used in conjunctions with a `groupby` in order to do lags and leads within a particular group. For example:

```
>>> dfMTAC = dfMTA.loc[(dfMTA.loc[:, 'direction'] == 'I'), :]

>>> dfMTAC = dfMTAC.sort_values(['plaza', 'mtadt', 'hr'])

>>> dfMTAgb = dfMTAC.groupby('plaza')

>>> dfMTAC.loc[:, 'pvsCash'] = dfMTAgb.shift(1).loc[:, 'vehiclescash']

>>> dfMTAC.loc[:, 'nxtCash'] = dfMTAgb.shift(-1).loc[:, 'vehiclescash']

>>> dfMTAC.iloc[61487:61490, :]
         plaza      mtadt  hr  ... vehiclescash  pvsCash  nxtCash
1163398      1 2017-01-07  23  ...          191    194.0      NaN
206928       2 2010-01-01   0  ...          290      NaN    363.0
206930       2 2010-01-01   1  ...          363    290.0    346.0

[3 rows x 8 columns]
```

- Note that we created two objects – the copy and one using a `groupby` in order to do this operation. The `groupby` facilitates the segmentation, but to do the assignment we then rely on returning the Series to the copied DataFrame. Since we haven't sorted the data between these operations we can be assured that the rows are still aligned.

- Shift can also work on an entire DataFrame:

```
>>> dfMTAC = (dfMTA
    .loc[ dfMTA.loc[:, 'direction']=='I', ['plaza', 'mtadt', 'hr', 'vehiclesez', 'vehiclescash']]
    .sort_values(['plaza', 'mtadt', 'hr'])
    )

>>> dfMTAC.shift(1).head()
        plaza      mtadt   hr  vehiclesez  vehiclescash
103440    NaN        NaT  NaN         NaN           NaN
103442    1.0 2010-01-01  0.0       415.0         474.0
103444    1.0 2010-01-01  1.0       702.0         717.0
103446    1.0 2010-01-01  2.0       559.0         664.0
103448    1.0 2010-01-01  3.0       480.0         595.0
```

6 Apply, map and applymap: Advanced Transformations

- In this section we consider three advanced methods for transforming columns: `map`, `apply` and `applymap`. These functions allow you to take a DataFrame or Series and apply an arbitrary function to it.

- The first of these we will consider is `applymap` which applies a function to a DataFrame element by element. Note that this function only works on entire DataFrames and not on series:

```
>>> from math import log10

>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head().applymap(log10)
   vehiclescash  vehiclesez
0      2.311754    2.678518
1      2.401401    2.686636
2      2.232996    2.544068
3      2.260071    2.487138
4      2.123852    2.447158
```

- I rarely use the function `applymap` since it applies a function to *every* value in a DataFrame, which isn't that helpful when you have mixed types within a DataFrame *and* that function does not already exist.

- When an alternative exists, `applymap` is generally slower since it applies operations element-by-element rather than vectorizing them in a multi-threaded manner. The two code snippets below do the same operation, but the second is faster (and easier to read).

```
>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head().applymap(lambda x: x**2)
   vehiclescash  vehiclesez
0         42025      227529
1         63504      236196
2         29241      122500
3         33124       94249
4         17689       78400

>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head() ** 2
   vehiclescash  vehiclesez
0         42025      227529
1         63504      236196
2         29241      122500
3         33124       94249
4         17689       78400
```

- Note also that you pass it a function which takes in a single argument (in the case above this was taking log base 10). If you need to pass in a function which takes in multiple arguments then you will need to use a lambda function.

- The function `map` is the same thing as `applymap` but now on a Series, not on a DataFrame. Once again, it applies an element-by-element operation on a series.

```
>>> dfMTA.loc[:, 'vehiclescash'].head().map(lambda x: x ** 2)
0    42025
1    63504
2    29241
3    33124
4    17689
Name: vehiclescash, dtype: int64
```

- One useful application of `map` is that you can pass it a dictionary and it will apply it as a map to that Series:

```
>>> TransDict =  {1 : 'Robert F. Kennedy Bridge Bronx Plaza (TBX)'
    , 2 : 'Robert F. Kennedy Bridge Manhattan Plaza (TBM)'
    , 3 : 'Bronx-Whitestone Bridge (BWB)'
    , 4 : 'Henry Hudson Bridge (HHB)'
    , 5 : 'Marine Parkway-Gil Hodges Memorial Bridge (MPB)'
    , 6 : 'Cross Bay Veterans Memorial Bridge (CBB)'
    , 7 : 'Queens Midtown Tunnel (QMT)'
    , 8 : 'Brooklyn-Battery Tunnel (BBT)'
    , 9 : 'Throgs Neck Bridge (TNB)'
    , 11 : 'Verrazano-Narrows Bridge (VNB)'}

>>> dfMTA.loc[:, 'plaza'].drop_duplicates().map(TransDict).reset_index(drop=True)
0         Robert F. Kennedy Bridge Bronx Plaza (TBX)
1     Robert F. Kennedy Bridge Manhattan Plaza (TBM)
2                      Bronx-Whitestone Bridge (BWB)
3                          Henry Hudson Bridge (HHB)
4    Marine Parkway-Gil Hodges Memorial Bridge (MPB)
5           Cross Bay Veterans Memorial Bridge (CBB)
6                        Queens Midtown Tunnel (QMT)
7                      Brooklyn-Battery Tunnel (BBT)
8                           Throgs Neck Bridge (TNB)
9                     Verrazano-Narrows Bridge (VNB)
Name: plaza, dtype: object
```

- The last of the complex transforms is `apply` which has both DataFrame and Series methods.
- The reason that `apply` is the most complex is that it is the most general on how it takes in a data as well as what it returns. Consider the following simple examples:

```
>>> d_1 = pd.DataFrame({'A' : [1,2,3], 'B': [4,5,6]})

>>> d_1.apply(np.sum, axis=1)
0    5
1    7
2    9
dtype: int64

>>> d_1.apply(np.sum, axis=0)
A     6
B    15
dtype: int64
```

  In these examples a function is passed to apply which takes in a list and returns a scalar, which is then returned. The axis argument tells tells `apply` in which direction the data is to be passed. When `axis` is equal to 1 then rows are passed to the function while if `axis` is equal to zero, then columns are passed.

- Importantly, `apply` can return complex objects:

```
>>> l_1 = lambda x: pd.Series([sum(x), len(x)])

>>> d_1.apply(l_1, axis=1)
   0  1
0  5  2
1  7  2
2  9  2

>>> d_1.apply(l_1, axis=0)
   A   B
0  6  15
1  3   3
```

The lambda function `t` returns a Series which is then stacked into a DataFrame by the `apply` method.

- We can also define more complex functions which are specific to the DataFrame in question:

```
>>> def f_1(x): return abs( x.loc['vehiclesez'] - x.loc['vehiclescash'] ) ** 2

>>> dfMTA.head().apply(f_1, axis=1)
0    73984
1    54756
2    32041
3    15625
4    21609
dtype: int64
```

This function does something specific to this DataFrame on a row-by-row basis.

- Note that all three of these methods create a *new* object and that it must be assigned back to the DataFrame if you want to access it later:

```
>>> dfMTAC = dfMTA.copy()

>>> def f_2(x): return abs( x.loc['vehiclesez'] - x.loc['vehiclescash'] ) ** 2

>>> dfMTAC.loc[:, 'newcol'] = dfMTAC.apply(f_2, axis=1)

>>> dfMTAC.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2015-11-28   0         I         477           205   73984
1      1 2015-11-28   0         O         486           252   54756
2      1 2015-11-28   1         I         350           171   32041
3      1 2015-11-28   1         O         307           182   15625
4      1 2015-11-28   2         I         280           133   21609
```