Chapter 16

Aggregations

Contents

1 Introduction to the MTA dataset

- In this section we will using the NY MTA dataset, as we did in the section XX. In order to load this dataset, use the following command:

```
>>> dfMTA = pd.read_csv('<FILEPATH>/MTA_Hourly.tdf'
                     , sep='\t', engine='python', names=['plaza', 'mtadt'
         , 'hr', 'direction', 'vehiclesez', 'vehiclescash'])

>>> dfMTA.mtadt = pd.to_datetime(dfMTA.mtadt)
```

  where <FILEPATH> needs to be changed to the appropriate local location.

2 Simple Aggregations

- Pandas provides a number of ways to do simple aggregations – which we define as those over the entire dataset. The table below shows the available aggregation functions:

Name	Description
count	Number of non-NaN values
sum	Sum of non-NaN values
mean	Average of non-NaN values
size	Number of rows
median	Median of non-NaN values
quantitle(X)	X-th quantitle of non-NaN values
std	Standard Deviation
var	Variance
min	Min of non-NaN values
max	Max of non-NaN values
prod	Product of non-NaN values
first	First non-NaN value
last	Last non-NaN value
nunique	Number of unique values

- For the purposes of this course, we will only focus on those that match the SQL ones: `min`, `max`, `mean`, `sum`, `count` and `nunique` (essentially count distinct).[1]

- There are five aggregation methods, which we will term either "Simplifying" or "Equal" depending on what gets returned relative to the original object they are applied to:

  1. Using an aggregation function, such as `sum()` (Simplifying).
  2. Using the `agg` command, with a string (Simplifying).
  3. Using the `agg` command, with a list of operators (Equal).
  4. Using the `agg` command, with a dictionary of single strings
     - On a DataFrame (Simplifying).
     - On a Series (Equal).

[1]For a complete list, check out the list of "Computations / Descriptive Stats" in the pandas documentation, which can be found here: `http://pandas.pydata.org/pandas-docs/stable/reference/frame.html`

5. Using the agg command, with a dictionary of lists. **Cannot be done on Series.** (Equal).

All of the methods above, save the last one, can be applied to both a DataFrame or a Series. The most important thing to remember is that the object you get out from the operation is dependent on both the input type and the operation itself.

- The operations above which say *simplifying* return a "lower" complexity object then the original one while those that say *equal* return an object of "equal" complexity. In terms of complexity order DataFrames are more complex than Series which are more complex than atomic numbers. So, if you use the second method above on a Series we would expect it to return a number. While if we use the third method on a Series we would expect it to return a Series.

- We will first run through each method demonstrating the complexity change. After that we will talk about the internals of the resulting object.

  1. First method: directly applying an aggregation function:

     – If we start with a **Series** and use this *simplifying* method, we will get a **number**:

     ```
     >>> dfMTA.loc[:, 'vehiclescash'].sum()
     330901032
     ```

     – If we start with a **DataFrame** and use the same, *simplifying* method, we will get a **Series**:

     ```
     >>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].sum()
     vehiclescash     330901032
     vehiclesez      1484674162
     dtype: int64
     ```

  2. Second method: Applying an aggregate function using the agg function, but only a single item via a **string**. This is again a *simplifying* method.

     ```
     >>> type( dfMTA.loc[:, 'vehiclesez'].agg('sum') )
     <class 'numpy.int64'>

     >>> type( dfMTA.loc[:, ['vehiclesez']].agg('sum') )
     <class 'pandas.core.series.Series'>
     ```

  3. Third method: Applying aggregate function(s) using the agg function, but via a **list**. This is an *equal* method:

     ```
     >>> type( dfMTA.loc[:, 'vehiclesez'].agg(['sum']) )
     <class 'pandas.core.series.Series'>

     >>> type( dfMTA.loc[:, ['vehiclesez']].agg(['sum']) )
     <class 'pandas.core.frame.DataFrame'>
     ```

  4. Fourth method: Applying aggregate function(s) using the agg function, but via a **dictionary** where every value in the dictionary is a string.

     – In the case of a Series this is an *equal* method:

```
>>> type(dfMTA.loc[:, 'vehiclesez'].agg({'vehiclesez' : 'sum'}))
<class 'pandas.core.series.Series'>
```

– In the case of DataFrame it is a *simplifying* method:

```
>>> type(dfMTA.agg({'vehiclesez' : 'sum'}))
<class 'pandas.core.series.Series'>
```

5. Fifth method: Applying aggregate function(s) using the `agg` function, but via a **dictionary** where every value in the dictionary is a list. This method only exists on DataFrames, it will **not** work on a Series.

```
>>> type(dfMTA.agg({'vehiclesez' : ['sum']}))
<class 'pandas.core.frame.DataFrame'>
```

- Similar to applying the aggregation functions directly there are different objects that can be returned depending on the form of the input. The final form (columns and indexes) of the returned value is also dependent on the data type and the operation that is completed. When looking at our examples, there are a finite number of possibilities:

1. **Returns a number:** This will return an atomic number.
2. **Return a Series:** Returns a Series, two possible forms:

(a) Index based on aggregate function name

```
>>> dfMTA.loc[:, 'vehiclesez'].agg(['count','sum'])
count       1165728
sum      1484674162
Name: vehiclesez, dtype: int64
```

(b) Index based on column name

```
>>> dfMTA.loc[:, ['vehiclesez', 'vehiclescash']].agg('sum')
vehiclesez      1484674162
vehiclescash     330901032
dtype: int64
```

3. **Return a DataFrame:** Returns a DataFrame:

(a) Index based on aggregate function names (therefore columns are column names)

```
>>> dfMTA.agg({'vehiclesez' : ['count', 'sum'], 'vehiclescash' : 'sum'})
       vehiclesez  vehiclescash
count     1165728           NaN
sum    1484674162   330901032.0
```

- Importantly the table in 17.1 contains a list of how our five operators produce output. In this table the **bolded options** present what is recommended for both coverage and ease of remembering.
- A very useful aggregation is `nunique` which counts the number of unique values in a list:

```
>>> dfMTA.loc[:, ['plaza','hr']].agg(['nunique'])
         plaza  hr
nunique     10  24
```

- Lets answer a quick question about the MTA dataset: What percentage of cars which pass through a toll place use an EZ pass?

```
>>> dfMTA.loc[:, 'vehiclesez'].sum()
    / (dfMTA.loc[:, 'vehiclesez'].sum() + dfMTA.loc[:, 'vehiclescash'].sum())
0.8177431410753236
```

- The command above is relatively straightforward. Each of the three aggregation operations returns an `numpy.int64` object. Traditional addition and division are then applied to get the final answer.

3 GroupBy Objects

- More complex aggregation operations require using the `groupby` method in pandas.
- The `groupby` method is a piece of the "split-apply-combine" pattern for handling subsetted data aggregation. This pattern involves taking a data set and *spliting* it along a dimension (usually values within a column or set of columns) *applying* an operation (such as sum) to similar values within those groups and then *combining* the results. Figure 16.1 presents this visually.

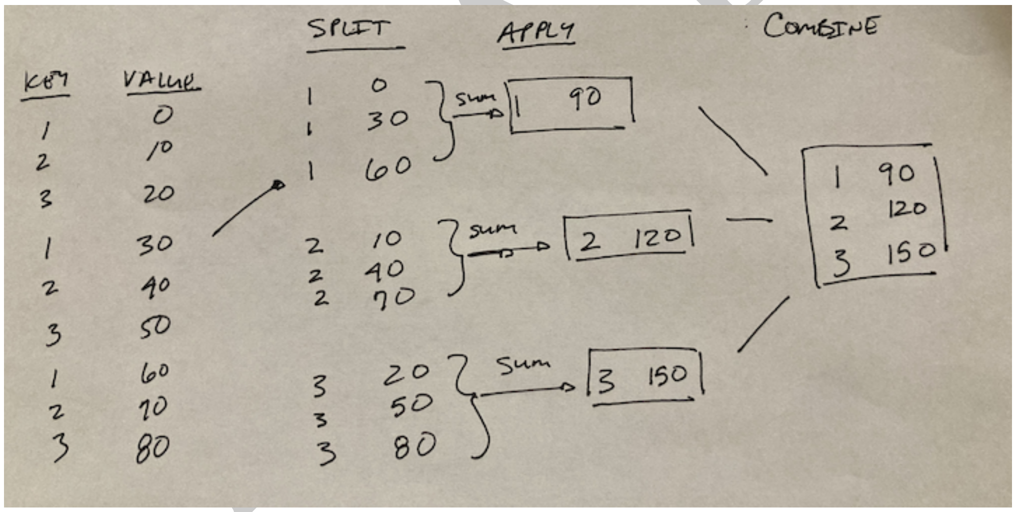

Figure 16.1: Split-Apply-Combine pattern

- For new users of pandas, the `groupby` object can be difficult to understand because it is not a static data result. Instead, the `groupby` object only contains information about the split definition – not the data itself.
- As a first example lets calculate the max `vehiclescash` by `plaza` in the dataset:

```
>>> dfMTAg = dfMTA.loc[:, ['plaza', 'vehiclescash']].groupby('plaza')

>>> type(dfMTAg)
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>

>>> x_1 = dfMTAg.max()

>>> x_1
       vehiclescash
plaza
1              1352
2              1040
3              1594
4              1368
5               674
6               844
7               727
8               599
9              1320
11             2116

>>> type(x_1)
<class 'pandas.core.frame.DataFrame'>
```

The first thing that we do is define a new object `dfMTAg` which is a `groupby` object. Similar to `loc` we provide it with a similar sized object in order to define the grouping. In this case we have told `groupby` to group by `plaza`.

Secondly, we limit our `groupby` object to only the `vehiclescash` column and then use the aggregation function `max` to calculate the maximum value. Instead of the above, we also could have done it this way which doesn't limit the columns calculated:

```
>>> dfMTAg = dfMTA.groupby('plaza')

>>> dfMTAg.max().loc[:, 'vehiclescash']
plaza
1     1352
2     1040
3     1594
4     1368
5      674
6      844
7      727
8      599
9     1320
11    2116
Name: vehiclescash, dtype: int64
```

However, this way would calculate the max across *all* columns before returning the `vehiclescash` column, which is much less efficient.

Note that this has a row index! The result of this calculation is a DataFrame which has an index equal to the columns chosen via the `groupby` method. This would be an index composed of integers.

```
>>> dfMTA.loc[:, ['plaza', 'vehiclescash']].groupby('plaza').max().index
Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 11], dtype='int64', name='plaza')
```

- We can combine multiple operations, such as restricting ourselves to only the second plaza. We either have to do the filtering *before* we create the GroupBy object or do it *after* we have calculated our aggregate functions. Both of the commands below return the same thing, but demonstrate this difference:

```
>>> (dfMTA
     .loc[(dfMTA.loc[:, 'plaza'] == 2)]
     .groupby('plaza')
     .max()
     .loc[:, 'vehiclescash'])
plaza
2    1040
Name: vehiclescash, dtype: int64
```

Once again – *what is being returned in each?* In the first situation we are getting returned a Series with an index (plaza), but only a single row while the second is returning only a single number, in this case a numpy integer.

- How about calculating the percentage of cars using EZ pass over each year?

```
>>> grp = (dfMTA
    .loc[:, ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
    .assign(yr=dfMTA.loc[:, 'mtadt'].dt.year)
    .groupby('yr')
    .sum())

>>> grp.loc[:, 'vehiclesez'] / (grp.loc[:, 'vehiclesez'] + grp.loc[:, 'vehiclescash'])
yr
2010    0.757150
2011    0.792285
2012    0.808791
2013    0.828819
2014    0.836587
2015    0.845386
2016    0.851932
2017    0.851355
dtype: float64
```

This works because the `groupby` object, after the sum has been applied, is just a *standard DataFrame* which we can do whatever expected math we'd like.

- If we want to group by multiple columns at the same time, we put the columns, as a list, into the `groupby` object. For example if we want to know the number of cars using the EZ pass which go through the toll plaza in each direction, we can do the following:

```
>>> d_1 = dfMTA.groupby(['plaza', 'direction']).sum(numeric_only=True)

>>> d_1.head()
                     hr  vehiclesez  vehiclescash
plaza direction
1     I          707112    71598396      26925764
      O          707112    75466906      27433718
2     I          707112   104041531      20567017
      O          707112    81520997      17442388
3     I          707112   104486985      32400024
```

Note the use of the `numeric_only` argument which is set to True. In versions of pandas before 2.0 this was not required, but now it is and if you do not use it a warning will appear. In general pandas will attempt the operation on all allowable columns.

Look carefully at the row index and we can see that we now have a multiindex / hierarchal index on the rows! We will be deep diving into this shortly.

- If we do not want the rows returned as an index we can add the argument `as_index` and set it to False in the `groupby`. Doing this returns the grouping variables as a column, rather than index. It is roughly equivalent to adding a `reset_index` to the result of the aggregation:

```
>>> d_1 = dfMTA.groupby(['plaza', 'direction'], as_index=False).sum(numeric_only=True)

>>> d_1.head()
   plaza direction      hr  vehiclesez  vehiclescash
0      1         I  707112    71598396      26925764
1      1         O  707112    75466906      27433718
2      2         I  707112   104041531      20567017
3      2         O  707112    81520997      17442388
4      3         I  707112   104486985      32400024
```

- Just as with DataFrames and Series there are five methods for doing aggregation and they can either be "simplifying" or "equal" in terms of complexity. In all cases, however, a DataFrame is returned and the simple versus equal refers to the shape of the output data and specifically what is returned in the columns.

  1. Using an aggregation command directly (simplifying).
  2. Using an aggregation command with a string (simplifying)
  3. Using an aggregation command with a list of aggregation functions (equal)
  4. Using an aggregation command with a dictionary of single strings (simplifying)
  5. Using an aggregation command with a dictionary of lists (equal).

- In all cases the row will be an index based on the contents of the `groupby`, so if there is a single item then there will be a single index, if there are multiple items it is a hierarchal/multiindex.

- With `groupby` the distinction between the simplifying and equal operations is how the columns are named / handled and there are two possibilities:

  1. (Similar to *simplifying* methods) We get a DataFrame where the index is the variables in the `groupby` and the columns have the name of the original columns.

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg('sum')

>>> d_1
           hr  vehiclesez
plaza
1      1414224   147065302
2      1414224   185562528
3      1414224   217926485
4      1387176   135142255
5      1414224    46534289
6      1414224    44735980
7      1414224   172800186
8      1412016   106104332
9      1414224   232681943
11      707112   196120862

>>> d_1.columns
Index(['hr', 'vehiclesez'], dtype='object')
```

This method does *not* state what aggregation function was used to get the result.

2. (Similar to *equal* methods) In the more complex case, then the result has a multiindex on the column:

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1
           hr     vehiclesez
          sum max        sum   max
plaza
1      1414224  23  147065302  3066
2      1414224  23  185562528  4307
3      1414224  23  217926485  4572
4      1387176  23  135142255  3640
5      1414224  23   46534289  1747
6      1414224  23   44735980  1604
7      1414224  23  172800186  4042
8      1412016  23  106104332  3402
9      1414224  23  232681943  4926
11      707112  23  196120862  8345

>>> d_1.columns
MultiIndex([(        'hr', 'sum'),
            (        'hr', 'max'),
            ('vehiclesez', 'sum'),
            ('vehiclesez', 'max')],
           )
```

In this case, the object being returned is another DataFrame. A few important notes:

  - First, as we saw before the plaza variable has been turned into an index on the rows, as this is the grouping column.
  - The columns though are a *total* mess. In this case we have what is called a hierarchal or multi-index on the columns.
  - The outer level has the name of the column begin aggregated while the inner level has the aggregation function.
  - We can see this index more clearly by looking at the `columns` attribute of the DataFrame which puts a list of tuples as the column information.

- I recommend, when using `gropuby` to lean into the the list/dictionary method as, while the multiindex columns aren't straightforward, it returns the name of the aggregation function that was used:

```
>>> d_1 = (dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
             .groupby( ['hr', 'plaza'] )
             .agg({'vehiclesez' : ['sum', 'max'], 'vehiclescash' : 'max'})
           )

>>> d_1.head()
         vehiclesez       vehiclescash
                sum   max          max
hr plaza
0  1        2535610  1860          900
   2        2626113  1948          556
   3        3521056  2753         1227
   4        1493721  1415          349
   5         480565   430          132
```

4 Advanced Index / Multiindex

- As before we have a DataFrame which has a multi-index on the rows. If we wish to remove this multi-index and return it to a column we can use `reset_index` or use the `as_index` argument in the `groupby` as described previously.

```
>>> d_1 = (dfMTA
    .groupby(['plaza', 'direction'])
    .agg({'mtadt' : ['first'], 'vehiclescash' : ['sum']})
    )

>>> d_1.reset_index(inplace=True)

>>> d_1.head()
  plaza direction      mtadt vehiclescash
                       first          sum
0     1         I 2015-11-28     26925764
1     1         O 2015-11-28     27433718
2     2         I 2015-11-28     20567017
3     2         O 2015-11-28     17442388
4     3         I 2015-11-28     32400024
```

- Before beginning this discussion, I want to preface this by stating that I'm not a big fan of index based methods in pandas and I think that they have some serious limitations.
- In this section we are going to cover how to reference values (both columns and rows) which have mulitindexes.
- The `loc` command can be used to access any index-based method on a **row**. For example:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[2, :]
    )
vehiclesez    sum    185562528
vehiclescash  sum     38009405
Name: 2, dtype: int64
```

In this example the "2" refers to plaza values which are equal to "2" and it gets returned as a Series.

- If we want to return it as a row we can put the selector inside a list:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[[2], :]
    )
      vehiclesez vehiclescash
             sum          sum
plaza
2      185562528     38009405
```

- In some ways this mirrors how we reference columns within `loc` commands. We can use any slice based reference, such as the following two examples demonstrate:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[2:4, :]
    )
      vehiclesez vehiclescash
             sum          sum
plaza
2      185562528     38009405
3      217926485     67000523
4      135142255     21397862
```

```
>>> (dfCars
    .groupby(['countyname'])
    .agg({'annualfee' : ['sum'], 'registrations' : ['count']})
    .loc[ "A":"B", :])
             annualfee registrations
                   sum         count
countyname
Adair       25300774.0           410
Adams       12645641.0           382
Allamakee   37068964.0           403
Appanoose   29947777.0           407
Audubon     20501452.0           393
```

- Note that in the above we *cannot* put the slice inside a list – this will raise an error:

```
>>> (dfCars
    .groupby(['countyname'])
    .agg({'annualfee' : ['sum'], 'registrations' : ['count']})
    .loc[ ["A":"B"], :])

  File "<stdin>", line 4
    .loc[ ["A":"B"], :])
             ^
SyntaxError: invalid syntax

>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[[2:4], :]
    )


  File "<stdin>", line 4
    .loc[ [2:4], :])
             ^
SyntaxError: invalid syntax
```

- As a reminder Python is case sensitive in its sorting:

```
>>> "C" < "D" < "a"
True
```

- In the case of a multi-index we need to use tuples to access rows. In this first example a Series is returned.

```
>>> (dfMTA
    .groupby(['plaza', 'hr'])
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[ (2,4), :])
vehiclesez    sum    1527122
vehiclescash  sum     533945
Name: (2, 4), dtype: int64
```

However, in this case, a DataFrame is returned:

```
>>> (dfMTA
    .groupby(['plaza', 'hr'])
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[ [(2,4)], :])
          vehiclesez vehiclescash
                 sum          sum
plaza hr
2     4     1527122       533945
```

- Personally, I tend to avoid using these systems as I find them to be complex and filled with a ton of "gotchas", but we will cover it for completeness so that you have seen it.
- Let's consider a specific example using the MTA data as we discuss accessing columns and specifying multiple levels of information.

```
>>> d_1 = (dfMTA
      .groupby(['plaza', 'direction'])
      .agg({'vehiclesez' : ['max', 'min'], 'vehiclescash' : ['max', 'min', 'sum']})
     )

>>> d_1.head()
                vehiclesez     vehiclescash
                       max min          max min       sum
plaza direction
1     I               2962   0         1232   0  26925764
      O               3066   0         1352   0  27433718
2     I               4307   0         1040   0  20567017
      O               3255   0          927   0  17442388
3     I               4572   0         1575   0  32400024
```

- The DataFrame above has an index (plaza and direction) and five columns associated with the aggregations. There are 19 total rows in the DataFrame (the 11th plaza only has Inbound activity). Both the rows and columns have multiindexes.
- To reference objects we can use tuples with our `loc`:

```
>>> d_1.loc[ :, ('vehiclescash', 'max')].head()
plaza  direction
1      I            1232
       O            1352
2      I            1040
       O             927
3      I            1575
Name: (vehiclescash, max), dtype: int64

>>> d_1.loc[ :, [('vehiclescash', 'max')]].head()
                vehiclescash
                         max
plaza direction
1     I                 1232
      O                 1352
2     I                 1040
      O                  927
3     I                 1575

>>> d_1.loc[ (1,'I'), :]
vehiclesez    max        2962
              min           0
vehiclescash  max        1232
              min           0
              sum    26925764
Name: (1, I), dtype: int64

>>> type( d_1.loc[ [(1,'I')], :] )
<class 'pandas.core.frame.DataFrame'>

>>> d_1.loc[ (1,'I'), ('vehiclescash', 'max')]
1232
```

Taking a look at the three examples above, we see that by replacing our traditional column name with a tuple we can reference single objects within our DataFrame.

- The first example above is straightforward and behaves as expected. We pass in a colon to return all rows and then pass in a tuple to identify the columns of interest. This returns a SERIES.
- The second example passes the tuple in as a list and (surprise, surprise), this returns the same data in the previous example, but this time as a DataFrame.
- The third example is similar, expect we use this select rows, rather than columns. HOWEVER, in this case, we find that the object returned is *not* a DataFrame, but instead a Series! This is just because we are selecting a *single* row, which we have completely specified from the original DataFrame. Note that this can happen when selecting rows based off of an index even without using tuples, as seen below.[2]

```
>>> type( dfMTA.iloc[0] )
<class 'pandas.core.series.Series'>
```

[2] Let's call this gotcha #1

- The fourth example applies the same list logic as with columns. In this case, instead of returning a Series, it returns a DataFrame containing the same data as the previous series.
- The fifth example has us selecting both a row and column based on tuples and it returns single value.
- To reference multiple values inside the tuple, we use the command `slice(None)` to create a slice which contains nothing. For example:

```
>>> d_1.loc[ : , ('vehiclesez', slice(None))].head()
                vehiclesez
                       max min
plaza direction
1     I               2962   0
      O               3066   0
2     I               4307   0
      O               3255   0
3     I               4572   0
```

  Or:

```
>>> d_1.loc[ (slice(None), 'I'), (slice(None), 'max')].head()
                vehiclesez vehiclescash
                       max          max
plaza direction
1     I               2962         1232
2     I               4307         1040
3     I               4572         1575
4     I               3640         1368
5     I               1675          674
```

  In this second example we use the tuples to return all the max values in the inbound direction.

- What if we want to select a few different values in a tuple, rather than a single one? We can use a list, which get interpreted as a filter.

```
>>> d_1.loc[ ([1,2], 'I'), (slice(None), 'max')]
                vehiclesez vehiclescash
                       max          max
plaza direction
1     I               2962         1232
2     I               4307         1040
```

  The command above uses a list to select either plaza #1 or plaza #2.

- Note that this tuple logic is *only* when using multiindexes. If you are accessing data based on the contents of the data (and not the index), then the traditional logic we used with `loc` is what you want to use.
- I find the tuple / `slice(None)` logic to be pretty discordant with how I use pandas. Because of this my normal pattern is to avoid using indexes unless there is an operation that requires them. In that case I then `set_index` do the operation and then `reset_index` in order to remove the index. It's too confusing for my small mind.

5 If not indexes...

- I commonly choose to remove the indexes on columns and usually do it via one of the methods below:
  1. **Drop a "level":** This method removes one of the levels (usually the outer one). It's easy to do, but the downside is that the resulting columns may have repeating names. The command to do this is the `droplevel` method of the multi-index and then reassign those values back to the columns:

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1.columns = d_1.columns.droplevel()

>>> d_1.head()
           sum  max        sum   max
plaza
1      1414224   23  147065302  3066
2      1414224   23  185562528  4307
3      1414224   23  217926485  4572
4      1387176   23  135142255  3640
5      1414224   23   46534289  1747
```

     The `droplevel` method, without any parameters, drops the outermost level and, in this case, the resulting set of columns have repeating names. If we wish to change those repeated names the best way to do this is by reassigning them via the `columns` parameter.[3]

```
>>> d_1.columns = ['sum_hr', 'max_hr', 'sum_vec', 'max_vec']
```

  2. **Concat 'em:** The following piece of code is something that I use frequently in order to remove the multi columns and replace them with a concatenated string.

[3]Since the columns have repeating names, the rename method does not work.

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1.columns = ['_'.join(col).strip() for col
    in d_1.columns.values]

>>> d_1
        hr_sum  hr_max  vehiclesez_sum  vehiclesez_max
plaza
1      1414224      23       147065302            3066
2      1414224      23       185562528            4307
3      1414224      23       217926485            4572
4      1387176      23       135142255            3640
5      1414224      23        46534289            1747
6      1414224      23        44735980            1604
7      1414224      23       172800186            4042
8      1412016      23       106104332            3402
9      1414224      23       232681943            4926
11      707112      23       196120862            8345
```

3. **Blow it all up:** This is the most common solution I use when doing EDA. In this solution I simply keep the columns I'm interested in and rename the rest by setting the column attribute, similar to what we did after the droplevel command earlier:

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1.columns = ['sum_hr', 'max_hr', 'sum_vec', 'max_vec']
```

6 Indexing with aggregations, a big Gotcha

- When using `groupby` objects it's important to pay attention to what you are grouping on and, specifically, it is an index or not.
- Frequently we wish the output of a `groupby` to not create an index, but instead just return a standard column. To do this, we use the `as_index=False` option.
- This function is pretty handy because it "seems" to allows us to move data back and forth between index to column when we do a `groupby`.
- HOWEVER, there is a big, big, gotcha with this! That gotcha is that even if you set `as_index=False` it will *not* pull data from an index to a column.
- Consider the following example:

```
>>> d_1 = (dfMTA
     .loc[:, ['mtadt', 'vehiclesez']]
     .groupby('mtadt', as_index=False)
     .sum())

>>> d_1.head()
       mtadt  vehiclesez
0 2010-01-01      316187
1 2010-01-02      380746
2 2010-01-03      359420
3 2010-01-04      494168
4 2010-01-05      518537

>>> d_2 = (dfMTA
     .loc[:, ['mtadt', 'vehiclesez']]
     .set_index(['mtadt'])
     .groupby('mtadt', as_index=False)
     .sum())

>>> d_2.head()
   vehiclesez
0      316187
1      380746
2      359420
3      494168
4      518537
```

In *both* cases we have set as_index=False, but in the second case, when the column being grouped is a part of the index *we lose the mtadt*! As stated, this is because when the column is originally an index, the as_index method will not pull the column out of the index, instead it will ignore it.

- This can also happen when using as_index=False and then using the column in the aggregation. In the example below we have added as_index=False and then aggregated by group by column. By doing this, plaza only appears once which, in this case is for the aggregation function and not from a column generated from an index.

```
>>> (dfMTA
    .groupby('plaza', as_index=False)
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum'], 'plaza' : 'count'})
    )
  vehiclesez vehiclescash   plaza
         sum          sum   count
0  147065302     54359482  122976
1  185562528     38009405  122976
2  217926485     67000523  122976
3  135142255     21397862  120624
4   46534289      7798630  122976
5   44735980      9676265  122976
6  172800186     26578805  122976
7  106104332     14368223  122784
8  232681943     53530379  122976
9  196120862     38181458   61488
```