Unit 3: Exploratory Analysis

Contents

Lab Questions

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Getting Started

With some data loaded and cleaned, we can begin to look at it more closely and see if we can identify any trends or relationships. To do this, we can rely on both quantitative methods such as the calculation and analysis of descriptive statistics as well as qualitative methods such as plotting. We'll rely on methods in pandas to calculate descriptive statistics. We'll rely on Matplotlib (https://matplotlib.org/) and [Seaborn] for plotting(https://seaborn.pydata.org/ (https://seaborn.pydata.org/)). Matplotlib is a popular plotting library capable of producing many different types of plots (https://matplotlib.org/gallery/index.html). Seaborn provides a simpler way of creating many of the plots commonly associated with data analysis and typically produces nicer looking plots (https://seaborn.pydata.org/examples/index.html).

To use Seaborn, we'll make sure it is installed with `pip`.

In [ ]:
```
!pip install seaborn
```

We'll be creating plots in this notebook. Before importing any modules, we should indicate to the notebook software how we would like to handle plots. We can use the `%matplotlib` (https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-matplotlib) magic command and `inline` to indicate that we would like plot to appear as static images in the notebook.

In [1]:
```
%matplotlib inline
```

With Seaborn installed, we can start importing modules for use in the notebook. Just as we followed convention and imported pandas as `pd`, we will import the Seaborn library as `sns`.

Following the import, we can set the appropriate pandas option to display 100 columns at a time. We can use the Searborn `set()` function to control figure size and the size of marker edges. Setting marker edges allows us to see outliers when working with box plots that would otherwise be invisible due to a bug in Matplotlib and Seaborn.

In [2]:
```
import pandas as pd
import seaborn as sns

sns.set(rc={'figure.figsize': (12, 10), "lines.markeredgewidth": 0.5 })
```

Selecting Data with Pandas

In this unit we'll continue to work with panda's Series and DataFrames to store and manipulate data. As part of that work, we'll be interested in examining parts of a larger DataFrame. A common way to select a subset of DataFrame is through the use of masks and filters - this is something we've already done. For example, consider the following DataFrame.

In [3]:
```
employees = pd.DataFrame(
    [
        ('John', 'Marketing', '123 Main St', 'Columbus', 'OH', 3),
        ('Jane', 'HR', '456 High Ave', 'Columbus', 'OH', 7),
        ('Bob',  'HR', '152 Market Rd', 'Cleveland', 'OH', 4),
        ('Sue', 'Marketing', '729 Green Blvd', 'Cleveland', 'OH', 8),
        ('Tom', 'IT', '314 Oak Dr', 'Cincinnati', 'OH', 11),
        ('Kate', 'IT', '841 Elm Ln', 'Cincinnati', 'OH', 2)
    ],
    columns = ("name", "department", "address",
               "city", "state", "years_with_company")
)

display(employees)
```

	name	department	address	city	state	years_with_company
0	John	Marketing	123 Main St	Columbus	OH	3
1	Jane	HR	456 High Ave	Columbus	OH	7
2	Bob	HR	152 Market Rd	Cleveland	OH	4
3	Sue	Marketing	729 Green Blvd	Cleveland	OH	8
4	Tom	IT	314 Oak Dr	Cincinnati	OH	11
5	Kate	IT	841 Elm Ln	Cincinnati	OH	2

If we want to work with a specific subset of the data, say only those employees that are in

Columbus, we could use the mask `employees.city == 'Columbus'` to filter the data.

In [4]:
```
employees[employees.city == 'Columbus']
```

Out[4]:

	name	department	address	city	state	years_with_company
0	John	Marketing	123 Main St	Columbus	OH	3
1	Jane	HR	456 High Ave	Columbus	OH	7

Pandas offers an alternative *query()* method (https://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-query) for selecting data that allows us to write statements that similar to standard Python syntax. For example, *query()* can be used to select only those employees in Columbus.

In [5]:
```
employees.query('city == "Columbus"')
```

Out[5]:

	name	department	address	city	state	years_with_company
0	John	Marketing	123 Main St	Columbus	OH	3
1	Jane	HR	456 High Ave	Columbus	OH	7

The argument we provide the *query()* method is a string that indicates the data we'd like extract from the original DataFame. When comparing string values, we have to to be sure to enclose the inner string in different quotes than the query itself.

Compare the following which both return data for employees that have been with the company for more than 4 years.

In [6]:
```
employees[employees.years_with_company > 4]
```

Out[6]:

	name	department	address	city	state	years_with_company
1	Jane	HR	456 High Ave	Columbus	OH	7
3	Sue	Marketing	729 Green Blvd	Cleveland	OH	8
4	Tom	IT	314 Oak Dr	Cincinnati	OH	11

In [7]:
```
employees.query('years_with_company > 4')
```

Out[7]:

	name	department	address	city	state	years_with_company
1	Jane	HR	456 High Ave	Columbus	OH	7
3	Sue	Marketing	729 Green Blvd	Cleveland	OH	8
4	Tom	IT	314 Oak Dr	Cincinnati	OH	11

So far there doesn't seem to be much of an advantage to one method over the other. A benefit to using *query()* becomes apparent when our conditions become more complex. Compare the two

methods when we want to find staff that have been with the company for more than 4 years and live in either Cleveland or Columbus. White space such as line returns and extra indentation has been added for clarity and is not required.

In [8]:
```
employees[(employees.years_with_company > 4) &
          ((employees.city == "Cleveland") |
           (employees.city == "Columbus"))]
```

Out[8]:

	name	department	address	city	state	years_with_company
1	Jane	HR	456 High Ave	Columbus	OH	7
3	Sue	Marketing	729 Green Blvd	Cleveland	OH	8

In [9]:
```
employees.query(
    "years_with_company > 4 and"
    "(city == 'Cleveland' or "
    " city == 'Columbus')"
)
```

Out[9]:

	name	department	address	city	state	years_with_company
1	Jane	HR	456 High Ave	Columbus	OH	7
3	Sue	Marketing	729 Green Blvd	Cleveland	OH	8

The argument supplied to the *query()* method is more concise and probably easier to read; generally, we write code with readability in mind as it is easier to share or understand later.

Lab 1 In the cell below, use the *query()* method to find all employees that live in Columbus and work for HR.

In [10]:
```
employees.query(
    "city == 'Columbus' and "
    "department == 'HR'"
)
```

Out[10]:

	name	department	address	city	state	years_with_company
1	Jane	HR	456 High Ave	Columbus	OH	7

One potential advantage to the non-*query()* method is programmability. As we write scripts, we often use variables to store values that will change and use the variables in our selection criteria. For example, suppose we have a function that routinely allows different departments to find their staff with more than 4 years with the company.

In [11]:
```
def senior_staff(dept):
    # return only those employees in the
    # specified department that have been
    # with the company for more than 4 years
    pass
```

Working with the notation we have been using, writing the function body the returns the correct result for a specified department is straightforward.

In [12]:
```
def senior_staff(dept):
    return employees[employees.department == dept]

senior_staff("IT")
```

Out[12]:

	name	department	address	city	state	years_with_company
4	Tom	IT	314 Oak Dr	Cincinnati	OH	11
5	Kate	IT	841 Elm Ln	Cincinnati	OH	2

Working with the *query()* method, there are multiple ways we can achieve the same result. The method suggested by the pandas' documentation makes use of `@`. We can reference existing variables in the query string in-line by prefixing their names with `@`.

In [13]:
```
def senior_staff(dept):
    return employees.query("department == @dept")

senior_staff("IT")
```

Out[13]:

	name	department	address	city	state	years_with_company
4	Tom	IT	314 Oak Dr	Cincinnati	OH	11
5	Kate	IT	841 Elm Ln	Cincinnati	OH	2

Both methods of selecting data have benefits and disadvantages. We'll primarily use the method we've been working with but occasionally use *query()*.

Exploring a Dataset

Let's start with the EPA/Department of Energy fuel economy dataset set we looked at last time. Often data cleaning, merging, and exploration are done together - data is cleaned as we examine it for relationships and trends and then the pertinent/interesting data is merged and stored for further analysis. Though we cleaned and merged the fuel economy and vehicle sales data previously, lets start with the original datasets for this initial exploration.

We can load the data from the `./data/02-vehicles.csv` file using pandas' `read_csv()` function.

In [14]:
```
epa_data = pd.read_csv("./data/02-vehicles.csv", engine="python")
epa_data.head()
```

Out[14]:

	barrels08	barrelsA08	charge120	charge240	city08	city08U	cityA08	cityA08U	cityCD	cityE
0	15.695714	0.0	0.0	0.0	19	0.0	0	0.0	0.0	0.0
1	29.964545	0.0	0.0	0.0	9	0.0	0	0.0	0.0	0.0
2	12.207778	0.0	0.0	0.0	23	0.0	0	0.0	0.0	0.0
3	29.964545	0.0	0.0	0.0	10	0.0	0	0.0	0.0	0.0
4	17.347895	0.0	0.0	0.0	17	0.0	0	0.0	0.0	0.0

5 rows × 83 columns

We also have a summarized data description document for this data, we can display it with the `HTML` function in the `IPython.display` module.

In [15]:
```
from IPython.display import HTML
HTML(filename="./data/02-vehicles-description.html")
```

Out[15]:

Data Description

From [fueleconomy.gov (https://www.fueleconomy.gov/feg/ws/index.shtml)](https://www.fueleconomy.gov (https://www.fueleconomy.gov/feg/ws/index.shtml)).

- atvtype - type of alternative fuel or advanced technology vehicle
  - Bifuel (CNG) - Bi-fuel gasoline and compressed natural gas vehicle
  - Bifuel (LPG) - Bi-fuel gasoline and propane vehicle
  - CNG - Compressed natural gas vehicle
  - Diesel - Diesel vehicle
  - EV - Electric vehicle
  - FFV - Flexible fueled vehicle (gasoline or E85)
  - Hybrid - Hybrid vehicle
  - Plug-in Hybrid - Plug-in hybrid vehicle
- barrels08 - annual petroleum consumption in barrels for fuelType1
- barrelsA08 - annual petroleum consumption in barrels for fuelType2
- charge120 - time to charge an electric vehicle in hours at 120 V

As part of our initial exploration, we'll attempt to catalog/categorize the values in the following columns and see if there are any relationships between pairs of them.

- `city08`

- city08U
- co2
- c02TailpipeGpm
- comb08
- comb08U
- cylinders
- displ
- fuelType1
- highway08
- highway08U
- year
- VClass

We'll also keep the following fields for each row.

- make
- model

Lab 2 In the cell below, create a copy of the `epa_data` DataFrame containing only the columns listed above. Store the new DataFrame in a variable named `epa_subset` . Use the *head()* method to confirm that the new DataFrame contains the correct column data.

In [16]:
```
columns = ["city08", "city08U", "co2", "co2TailpipeGpm", "comb08", "comb08U"
           "cylinders", "displ", "fuelType1", "highway08", "highway08U",
           "make", "model", "VClass", "year"]
epa_subset = epa_data[columns].copy()
epa_subset.head()
```

Out[16]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highway08
0	19	0.0	-1	423.190476	21	0.0	4.0	2.0	Regular Gasoline	25
1	9	0.0	-1	807.909091	11	0.0	12.0	4.9	Regular Gasoline	14
2	23	0.0	-1	329.148148	27	0.0	4.0	2.2	Regular Gasoline	33
3	10	0.0	-1	807.909091	11	0.0	8.0	5.2	Regular Gasoline	12
4	17	0.0	-1	467.736842	19	0.0	4.0	2.2	Premium Gasoline	23

As we've seen before, we can use the DataFrame *describe()* method to quickly calculate some discriptive statistics for each of the numeric columns in the dataframe.

Lab 3 In the cell below, use the *describe()* method to calculate descriptive statistics for the numeric columns of `epa_subset`.

In [17]:
```
epa_subset.describe()
```

Out[17]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	
count	39518.000000	39518.000000	39518.000000	39518.000000	39518.000000	39518.000000	3935
mean	18.160787	5.261330	75.711549	469.675894	20.405790	5.889232	
std	7.307656	10.842512	165.824216	122.861548	7.167849	11.573437	
min	6.000000	0.000000	-1.000000	0.000000	7.000000	0.000000	
25%	15.000000	0.000000	-1.000000	386.391304	17.000000	0.000000	
50%	17.000000	0.000000	-1.000000	454.000000	20.000000	0.000000	
75%	20.000000	0.000000	-1.000000	535.000000	23.000000	0.000000	
max	150.000000	150.000000	847.000000	1269.571429	136.000000	136.000000	1

Let's look at one column to get an idea of what these values represent. According to the documentation, the `city08` column represnts the fuel economy for city driving with the primary fuel type. The rows have the following meaning.

- `count` : the number of non-null elements in the column; here there are 39,518 non-null values in the `city08` column
- `mean` : the sum of all values divided by the number of values; the mean value for `city08` is 18.2 mpg.
- `std` : the standard deviation - a measure of the variation of values within a collection, can be thought of as an "average" distance to the mean among all the values; the standard deviation of values in the `city08` column is 7.3 mpg.
- `min` and `max` : the smallest and largest values, respectively; here we have 6.0 mpg and 150.0 mpg.
- `25%`, `50%`, and `75%` : the quartile values that allow us to divide the data into four parts. The first quartile, 25%, corresponds to the value between the minimum and the median; 25% of values are less than this value. The second quartile is the median, the middle most number among the values; 50% of values are less than the median and 50% of values are greater than the median. The third quartile represents the middle value between the median and the maximum; 25% of values are greater than this value.

As we noted previously, *describe()* only display results for numeric columns. For non-numeric columns, we might be interested in knowing about the data values and how often those values appear. Below, we iterate through the columns of the DataFrame and if the the column is a string data we display the column name with the output from the *value_counts()* method.

In [18]:
```
for column in epa_subset.columns:
    if pd.api.types.is_string_dtype(epa_subset[column]):
        display(column, epa_subset[column].value_counts())
```

```
'fuelType1'
```

```
Regular Gasoline     27134
Premium Gasoline     10950
Diesel                1118
Electricity            162
Midgrade Gasoline       94
Natural Gas             60
Name: fuelType1, dtype: int64
```

```
'make'
```

While having access to these results can be useful, we often rely on visualizations to help characterize data or provide insights into potential relationships. Before generating visualizations, let's address some data quality issues. First, we can remove duplicates.

Lab 4 In the cell below, remove duplicate rows from the `epa_subset` DataFrame.

In [19]:
```
epa_subset.drop_duplicates(inplace=True)
```

We'll also need to account for missing data - while some methods we'll use to explore the data are able to ignore missing values, other will fail and throw exceptions.

From the code below we can see that the `cylinders` and `displ` columns are missing data.

In [20]:
```
epa_subset.isna().sum()
```

Out[20]:
```
city08              0
city08U             0
co2                 0
co2TailpipeGpm      0
comb08              0
comb08U             0
cylinders         165
displ             163
fuelType1           0
highway08           0
highway08U          0
make                0
model               0
VClass              0
year                0
dtype: int64
```

Let's see if we can identify any common properties for rows missing `cylinders` or `displ` data. We can filter the DataFrame using a mask that corresponds to a row in which any column value is missing.

In [21]:
```
epa_subset[epa_subset.isna().any(axis=1)].head()
```

Out[21]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highway
7138	81	0.0	0	0.0	85	0.0	NaN	NaN	Electricity	
7139	81	0.0	0	0.0	72	0.0	NaN	NaN	Electricity	
8143	81	0.0	0	0.0	72	0.0	NaN	NaN	Electricity	
8144	74	0.0	0	0.0	65	0.0	NaN	NaN	Electricity	
8146	45	0.0	0	0.0	39	0.0	NaN	NaN	Electricity	

It looks like the rows with missing cylinder and displacement data correspond to electric vehicles. This makes sense given the fact that electric vehicles do no have an internal combustion engine. Let's refine the mask to exclude rows where `fuelType` is `Electricity`.

In [22]:
```
epa_subset[(epa_subset.isna().any(axis=1)) &
           (epa_subset.fuelType1 != 'Electricity')].head()
```

Out[22]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highwa
21409	22	0.0	-1	370.291667	24	0.0	NaN	NaN	Regular Gasoline	
21410	21	0.0	-1	386.391304	23	0.0	NaN	NaN	Regular Gasoline	
21502	15	0.0	-1	493.722222	18	0.0	NaN	1.3	Regular Gasoline	

It appears that these rows are anomalies and are simply missing data.

Before continuing on, we'll remove rows with missing data.

In [23]:
```
epa_subset.dropna(inplace=True)
```

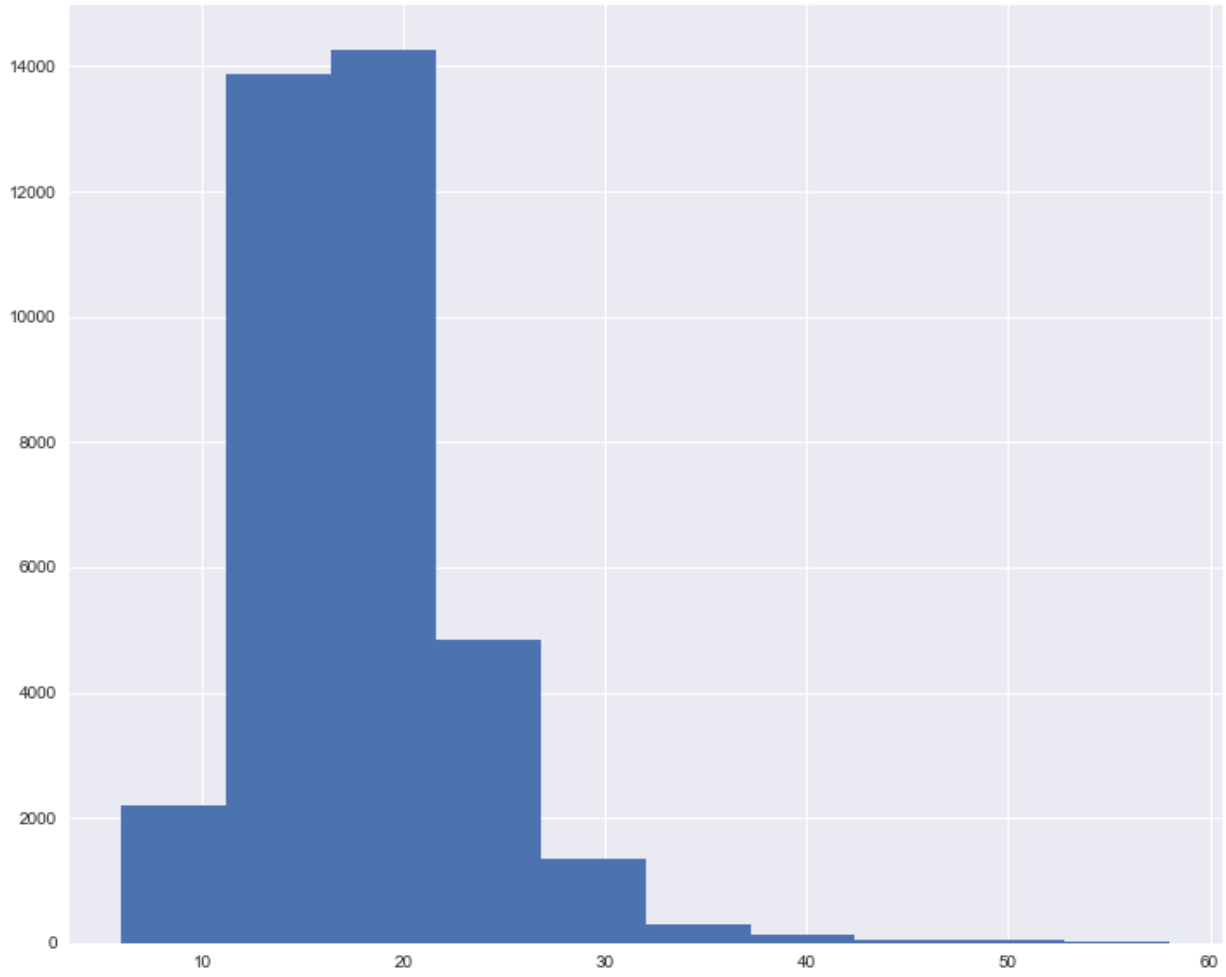
Histograms

To begin getting a higher-level picture of our data, we can use visualizations. While we have some sense of the distribution of data values from the quartile information calculated by the *describe()* method, a histograms (https://en.wikipedia.org/wiki/Histogram) can be used to visualize the date distribution.

Both pandas DataFrames and Series have *hist()* methods that can be used to plot histograms. This allows us to create a histogram for a specific column or for each column in a DataFrame with numeric values.

In [24]:
```
epa_subset.city08.hist()
```

Out[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10b7ce470>
```



The *hist()* method returns an `AxesSubplot` object that can be used to manipulate the plot - this is what the text above the plot refers to - we can ignore this now.

From the plot we can see that most of the values are concentrated between 10 and 30 mpg. We can also see that the distribution has a positive skew (https://en.wikipedia.org/wiki/Skewness). We can confirm this using the column's `skew()` method. Similarly, we can calculate the kurtosis (https://en.wikipedia.org/wiki/Kurtosis) using the *kurtosis()* method.

Lab 5 Calculate and display the skew and kurtosis of the data in the `city08` column.

In [25]:
```
display(epa_subset.city08.skew())
display(epa_subset.city08.kurtosis())
```

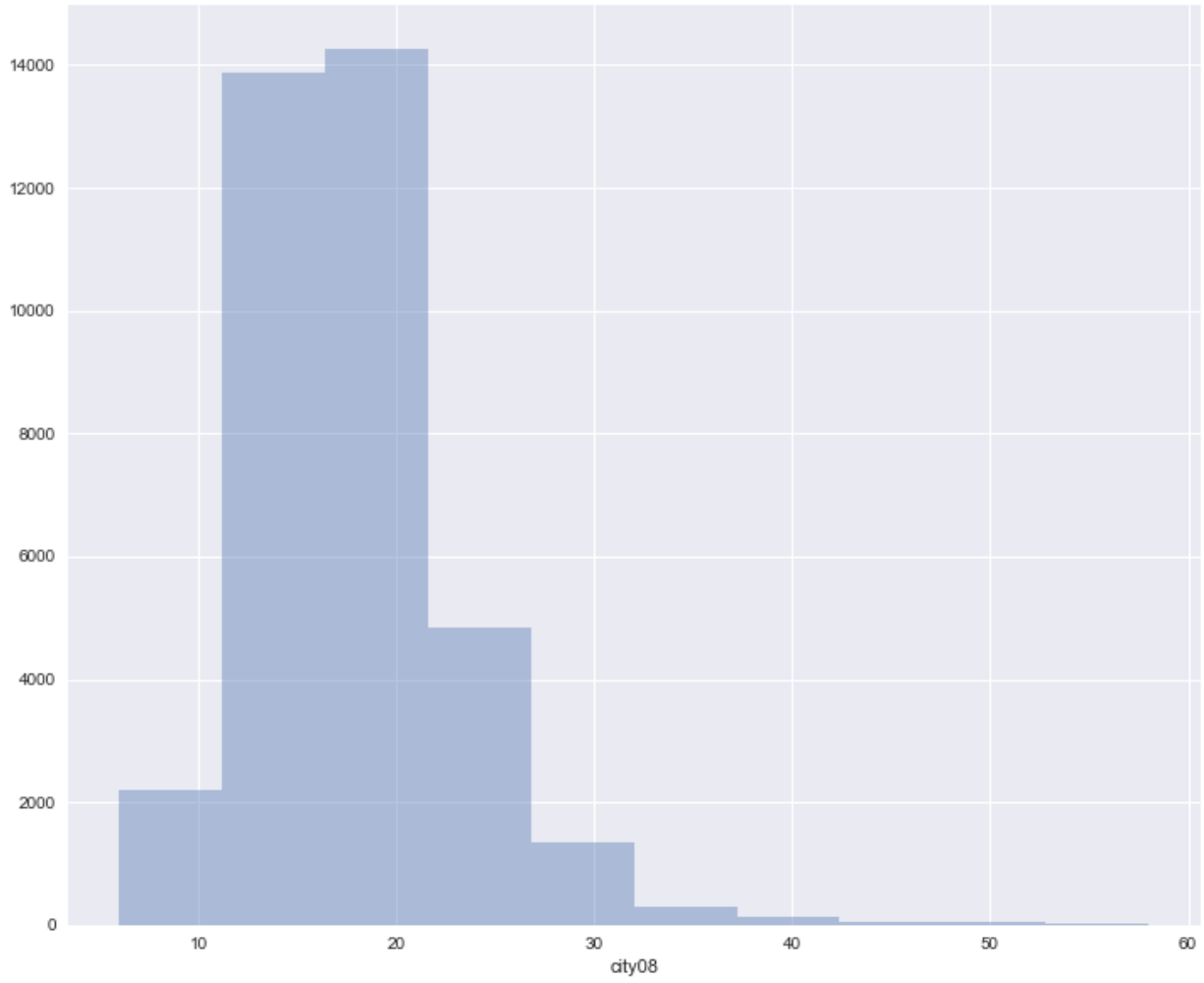
```
1.488601172240319
```

```
5.280570889879309
```

The pandas *hist()* method relies on Matplotlib and Seaborn to generate the plot. We can generate a histogram directly from Seaborn if we'd like. To do this, we can use the *distplot()* (https://seaborn.pydata.org/generated/seaborn.distplot.html) function. By default, the function generates a plot representing the probability distribution of observations rather than the count of values. To generate a plot based on the count, we have to provide the `kde=False` argument. We can also specify `bins=10` for consistency with the previous plot.

In [26]:
```
sns.distplot(epa_subset.city08, kde=False, bins=10)
```

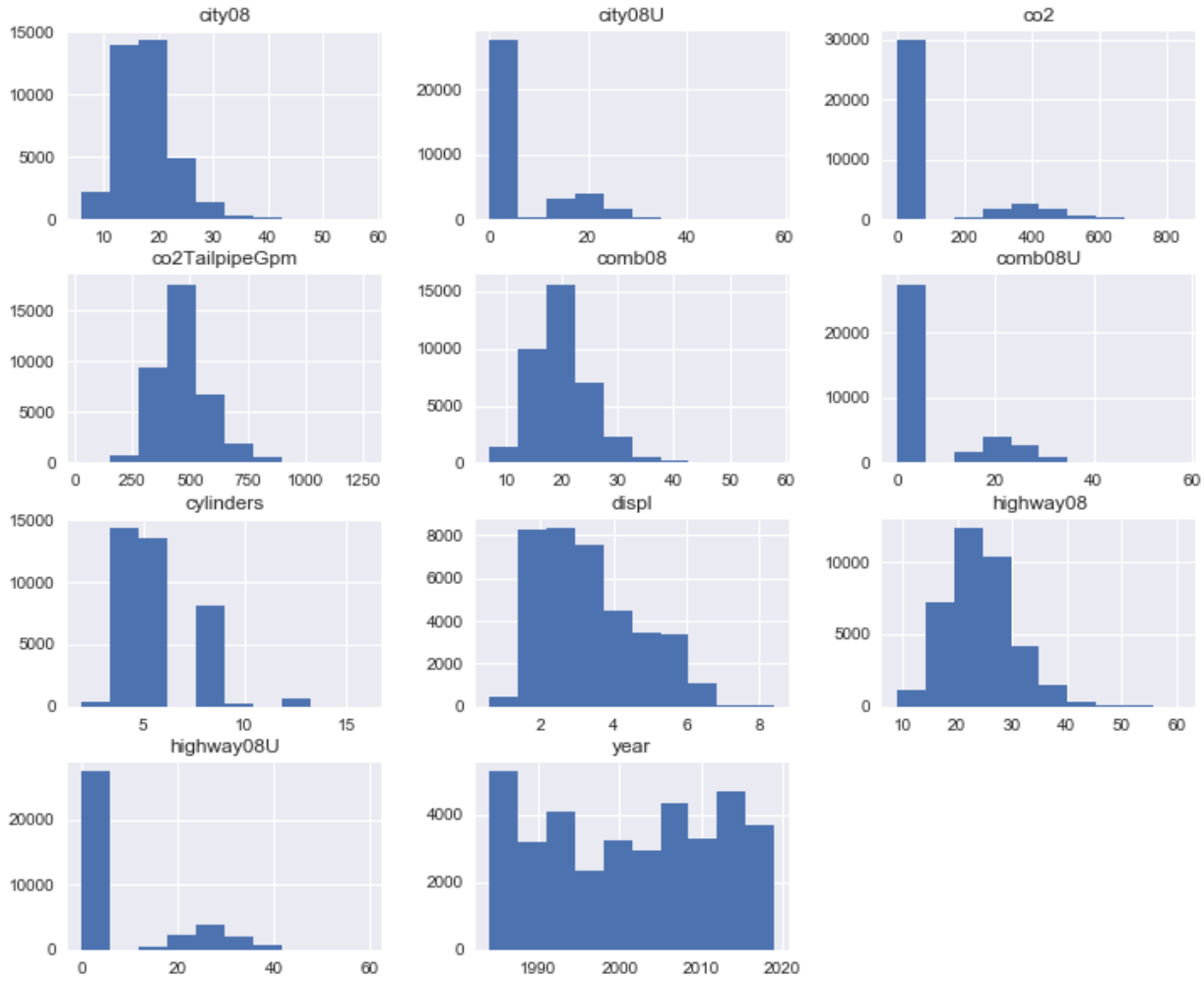
Out[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10b7db6a0>
```



To generate the histograms for each of the numeric columns in the DataFrame, we can use the DataFrame's *hist()* method rather than the Series *hist()* method associated with an individual column.

In [27]:
```
epa_subset.hist()
```

Out[27]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x10b7fb1d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10b836278>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10b898828>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x10b9834a8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10b9d8240>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10b9d8208>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x10ba1c518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10ba39278>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10ba60780>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x10ba77b70>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10ba9b0f0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10bac04e0>]],
      dtype=object)
```



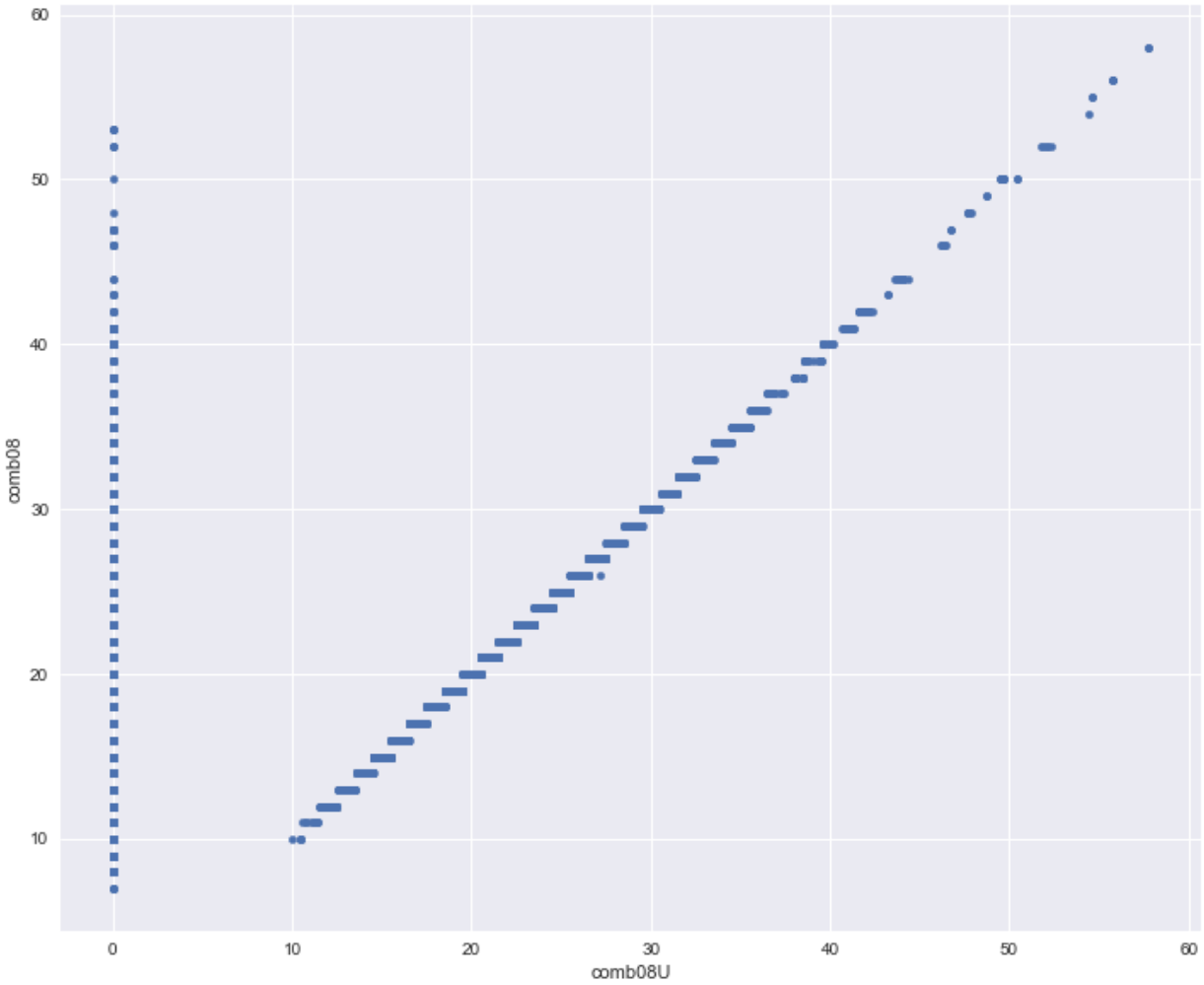
Viewing the histograms allow us to see the distrubution of data for each column more quickly than looking at results of *describe()* or similar methods. From a quick glance, we can see that there noticable differences between the values in `city08` and `city08U`. From the data documentation, we know that the `city08` column contains "unrounded data" but, when comparing the historgrams between the two columns, it apepars that the "unrounded" data contains more zero values.

Scatter Plots

While a histogram is useful to see how data is distributed for a signle column, we often would like to see if any relationships exist between two columns/variables. We can compare the values of two columns directly using a scatter plot.

In [28]:
```
epa_subset.plot.scatter(x="comb08U", y="comb08")
```

Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10bac6550>
```

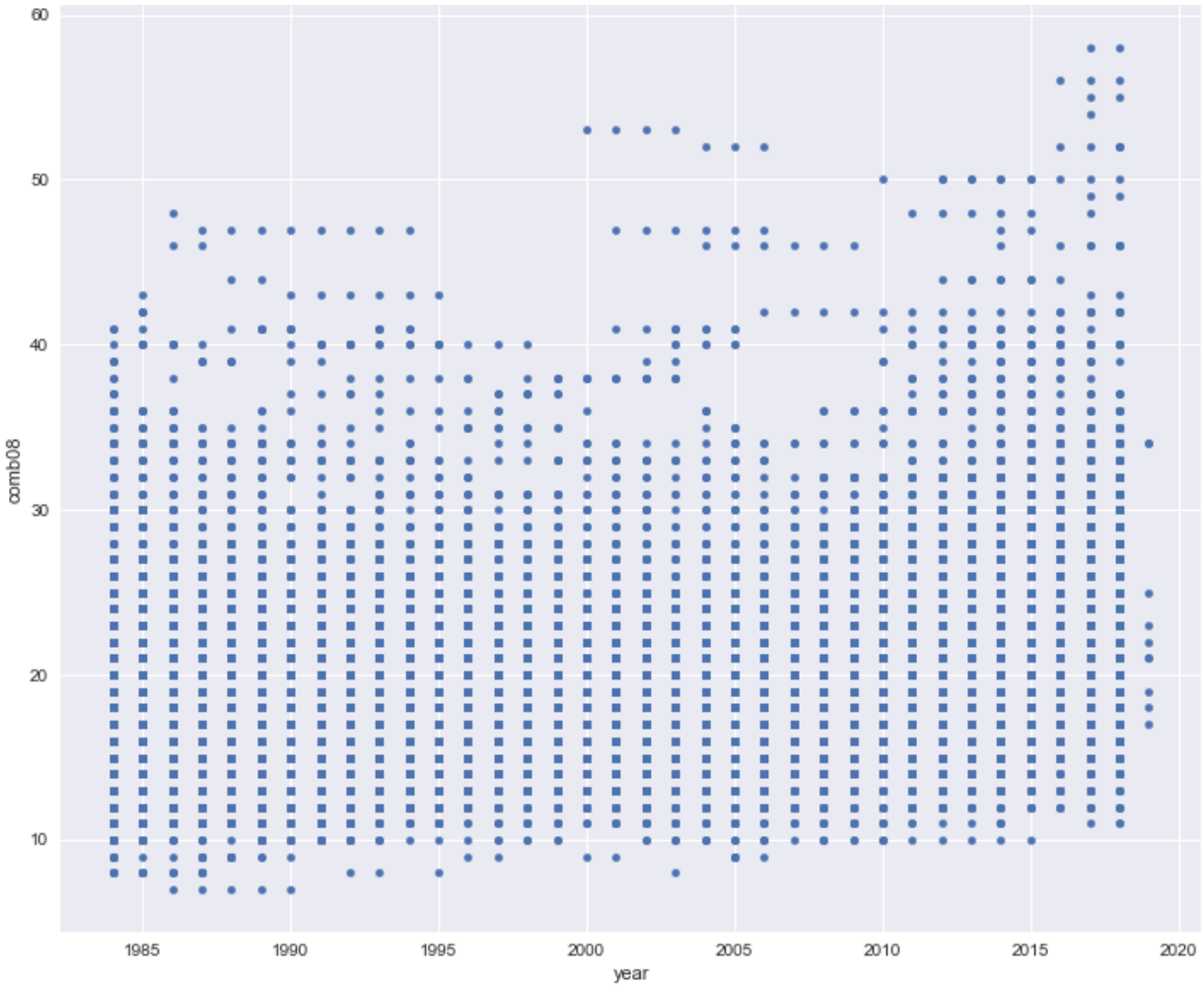


Notice that there are rows in which `comb08U` has a value of zero but the value of `comb08` is non-zero. Because it doesn't make sense to round zero to a non-zero value, it's reasonable to conclude that there is missing data for the unrounded values and zero was used as a placeholder. It might be the case that before some point in time only rounded values were stored. To verify this, we can compare the values of both columns against the `year` data assuming measurements were made around the the time the vehicles were manufactured.

Lab 6 In the cells below, verify that rounded values for combined fuel efficiency are available for earlier years compared to to unrounded values by creating two scatter plots.

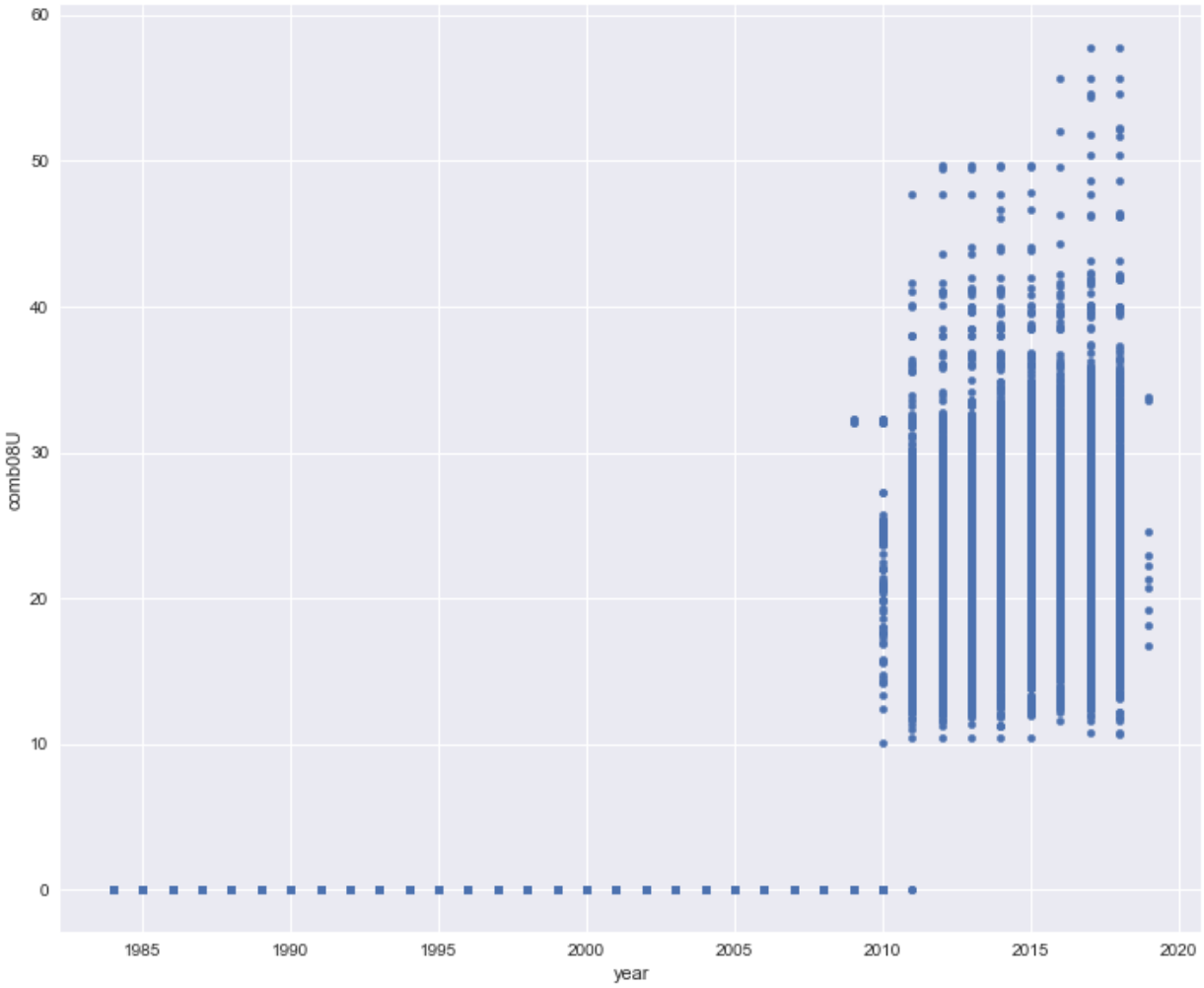
In [29]:
```
epa_subset.plot.scatter(x="year", y="comb08")
```

Out[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10be7cdd8>
```



In [30]:
```
epa_subset.plot.scatter(x="year", y="comb08U")
```

Out[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10b89e390>
```



We can see that the `city08U`, `comb08U`, and `highway08U` columns have the same number of zero-valued entries, which supports the idea that unrounded data from earlier years isn't available.

In [31]:
```
for column in ["city08U", "comb08U", "highway08U"]:
    zeros = epa_subset[epa_subset[column] == 0]
    display(column, len(zeros))
```

```
'city08U'

27489

'comb08U'

27489

'highway08U'

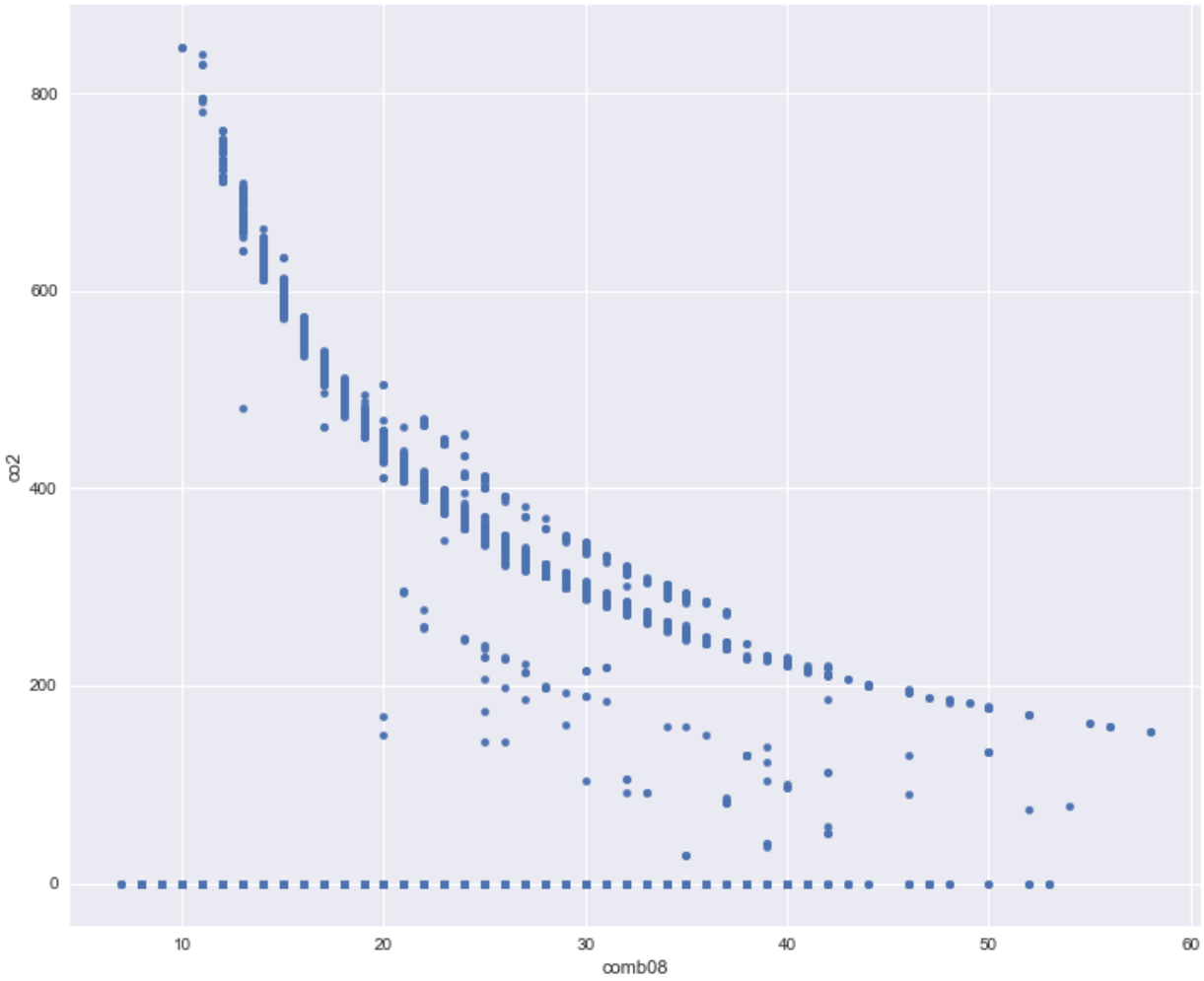
27489
```

With this in mind, we might choose to work with the rounded data if we wanted to work with a larger set of data including more historic data. If there is a desire to work with unrounded values or we wish to examine only recent data, we could work with the unrounded values. Having more historic data will be useful to us so we'll work with rounded data.

Having a sense of the distribution of a single column's data is important but we're often interested in how one or more columns influence another column. When working with two columns, we often use scatter plots to asses potential relationships. We can create a scatter plot for two columns as we did above when looking at rounded and unrounded data compared to years. As another example, let's compare the values of `comb08`, the combined highway and city fuel efficiency in miles/gallon, and `co2`, the tailpipe CO2 emissions in grams/mile.

In [32]:
```
epa_subset.plot.scatter(x="comb08", y="co2")
```

Out[32]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10bf03898>
```



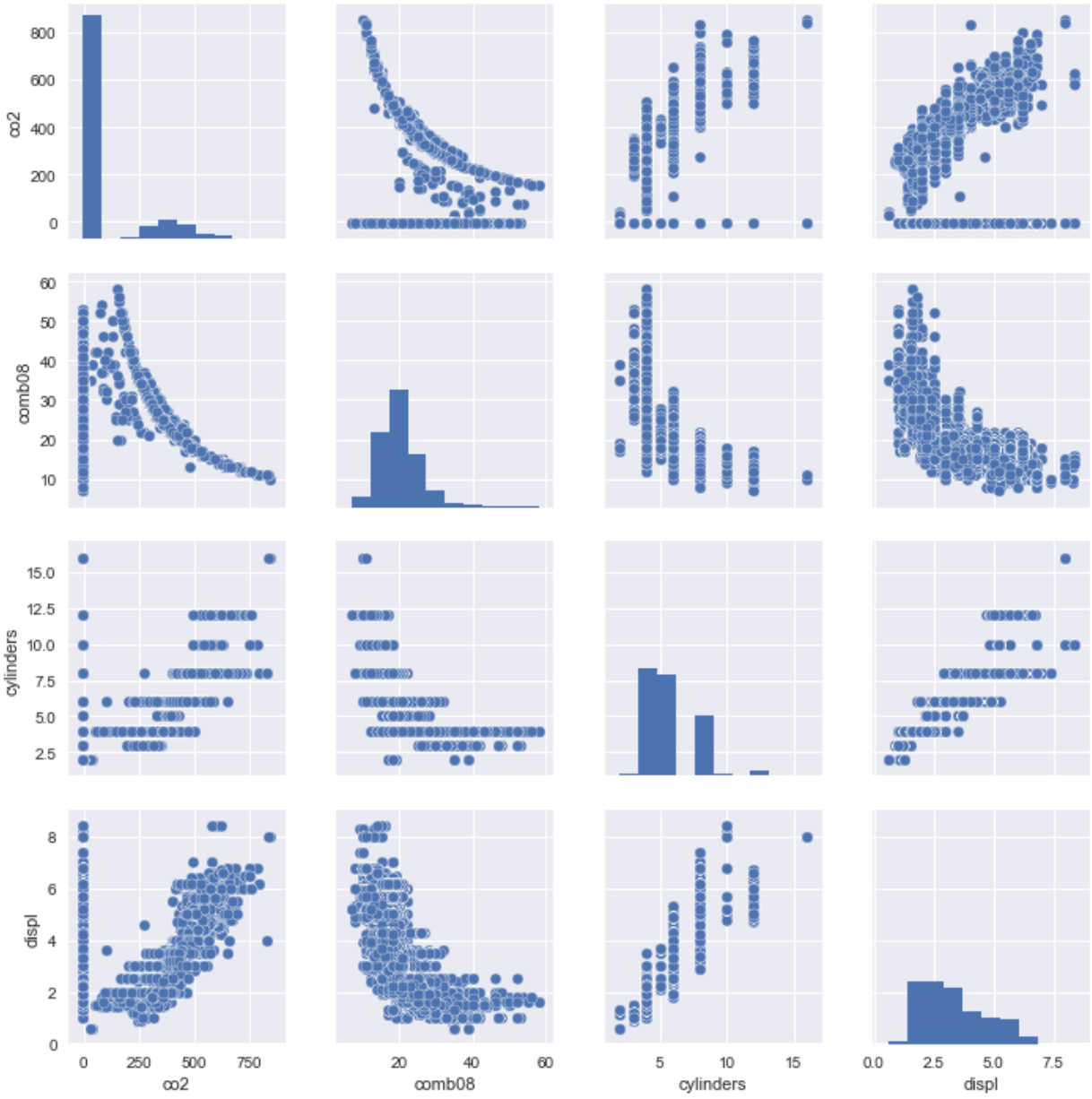
Here we can see that there appears to be a relationship between fuel efficiency and carbon dioxide emissions - as fuel efficiency improves, emissions decrease. We'll examine this relationship further in a later unit.

Pair Plots: Histograms and Scatter Plots

Just as it was helpful to be able to quickly visualize distribution data for each of the numeric columns in our dataset, we can use a *pairplot* (https://seaborn.pydata.org/generated/seaborn.pairplot.html) to visualize the pairwise relationships between columns. In the example below, we first further reduce the columns we'll examine then use the Seaborn *pairplot()* function to generate the pairwise scatter plots for those columns. Note that when a column is paired with itself, the column's histogram is displayed.

In [33]:
```
columns = [ "co2", "comb08", "cylinders", "displ"]
sns.pairplot(epa_subset[columns])
```

Out[33]:
```
<seaborn.axisgrid.PairGrid at 0x10c179908>
```



Notice that there is a form of symmetry to the plots with respect to the plots to the left and below the histograms and the plots to the right and above. For example, the the second plot in the first row and the first plot in the second column both show the relationship between `co2` and `comb08` - the axis to which each variable corresponds differs but the relationship is the same.

Using pair plots can help us quickly identify which columns or variables are dependent on other columns/variables.

The `co2` column contains quite a few values that appear to be zero but are -1 (as can be seen from the output of *describe()*. Let's see what those are.

In [34]:
```
epa_subset.query("co2 == -1").head()
```

Out[34]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highway08
0	19	0.0	-1	423.190476	21	0.0	4.0	2.0	Regular Gasoline	25
1	9	0.0	-1	807.909091	11	0.0	12.0	4.9	Regular Gasoline	14
2	23	0.0	-1	329.148148	27	0.0	4.0	2.2	Regular Gasoline	33
3	10	0.0	-1	807.909091	11	0.0	8.0	5.2	Regular Gasoline	12
4	17	0.0	-1	467.736842	19	0.0	4.0	2.2	Premium Gasoline	23

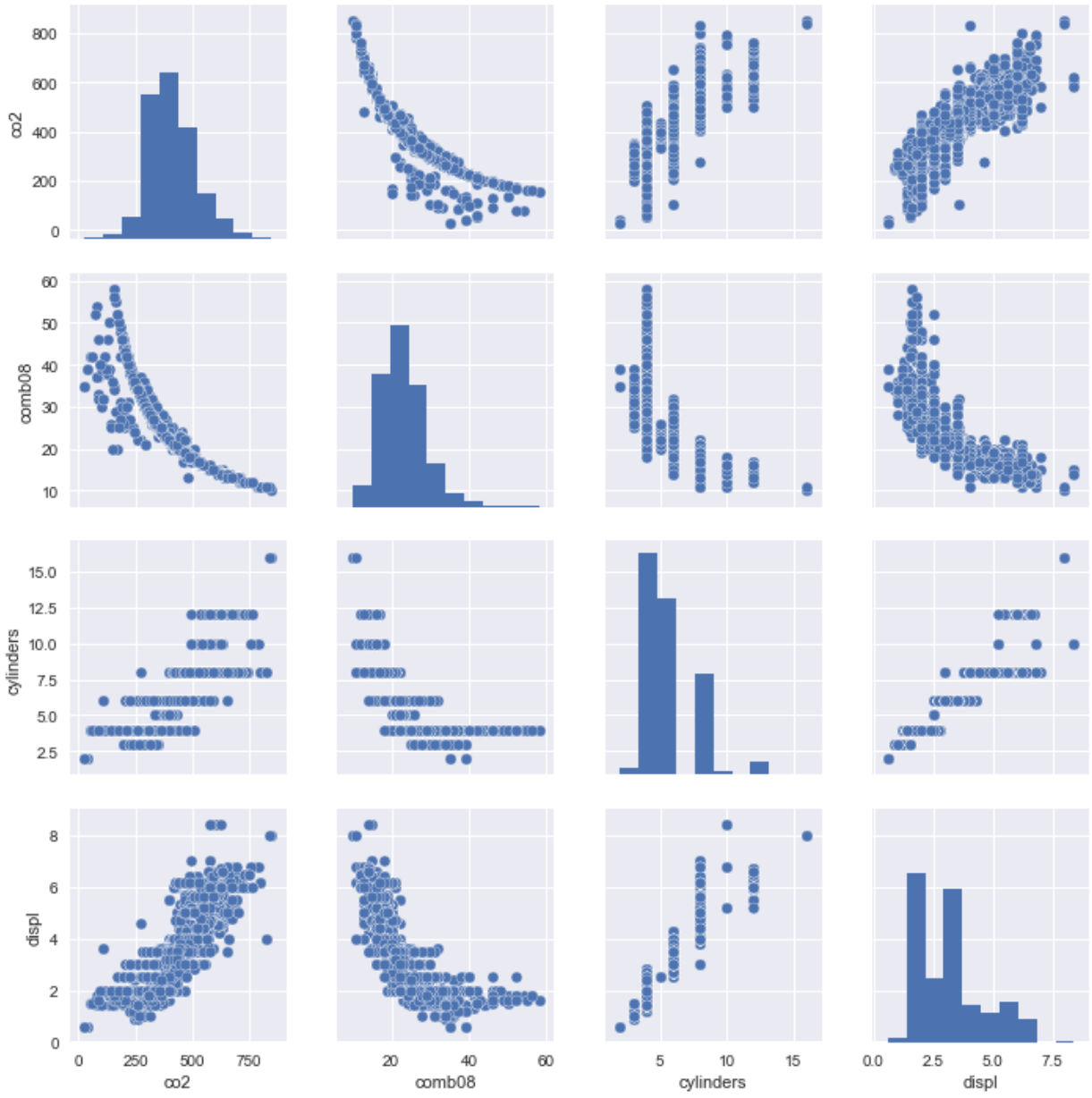
Its not immediately clear what might be the reason for the carbon dioxide emissions having a value of -1 but we'll remove them for the remainder of our work.

In [35]:
```
epa_subset = epa_subset.query('co2 >= 0')
```

With that, let's look at the pair plots again.

In [36]:
```
columns = [ "co2", "comb08", "cylinders", "displ"]
sns.pairplot(epa_subset[columns])
```

Out[36]:
```
<seaborn.axisgrid.PairGrid at 0x10cf406a0>
```



Categorical Data

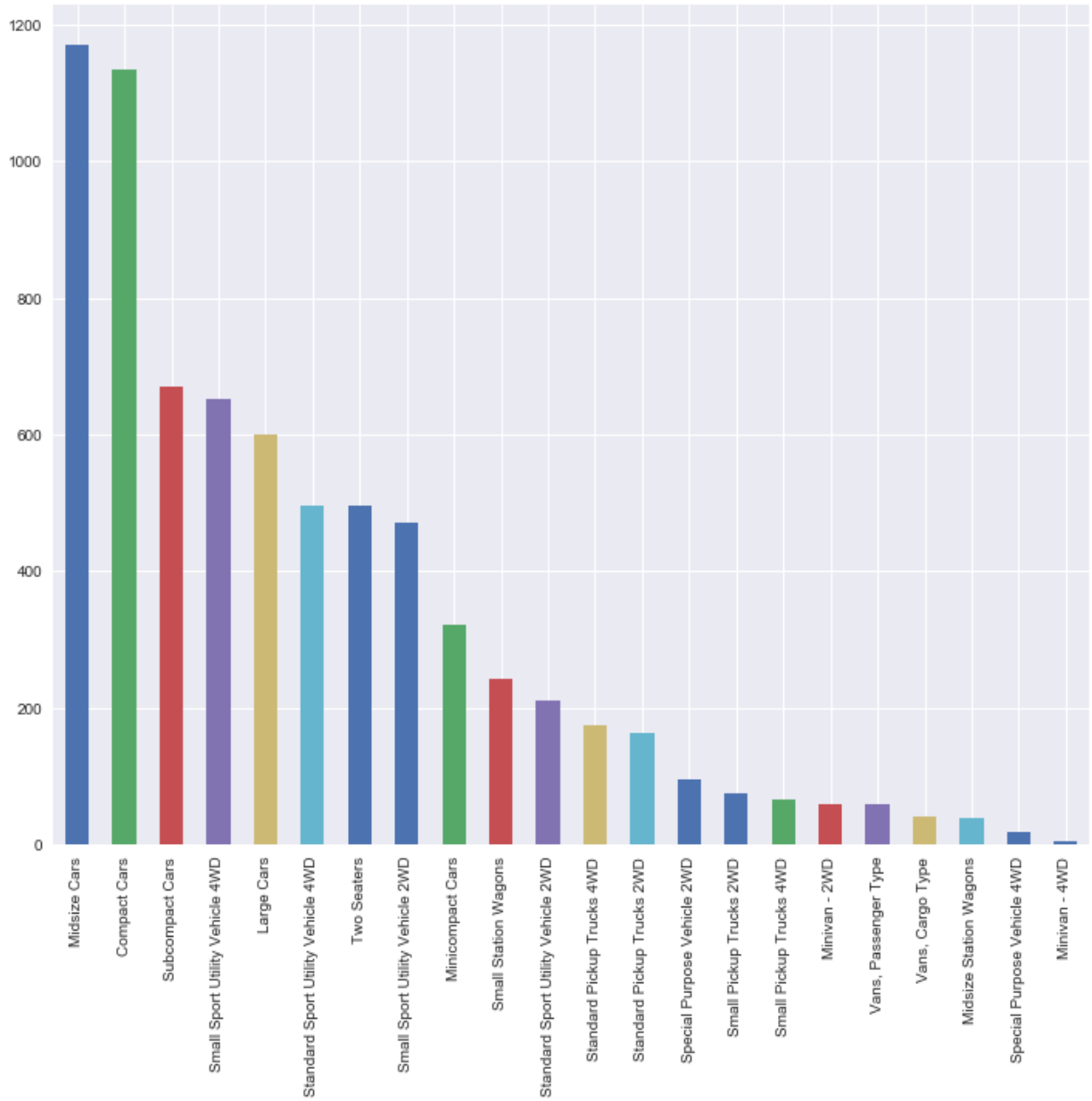
While the examples above provide insight into numeric data, they don't tell us about columns that contain categorical data such as `VClass`.

A typical method of visualizing categorical data is with a bar plot. Both pandas and Seaborn provide methods of generating bar plots.

For a given column in the DataFrame, the *value_counts()* method returns a Series. Series, as we've seen before, have a *plot()* method. We can explicitly create a bar chart using the `kind=bar` or `kind=barh` arguments to the *plot()* method for a vertical or horizontal bar chart, respectively.

In [37]:
```
epa_subset.VClass.value_counts().plot(kind="bar")
```

Out[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10e2ec860>
```

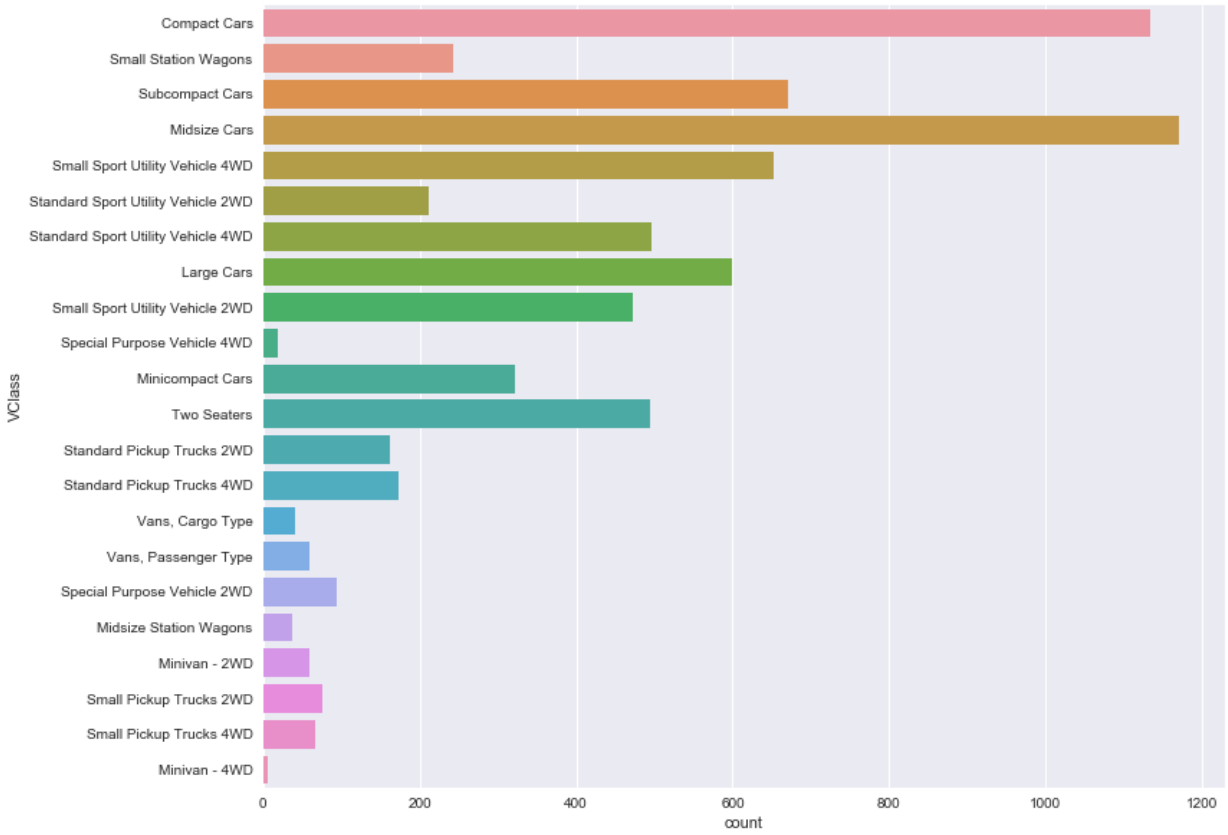


The order of the categories is determined by the Series used to create the chart; because the results of *value_counts()* are ordered, the bars in the resulting plot are ordered.

We can create a similar plot using the Seaborn [*countplot()*](https://seaborn.pydata.org/generated/seaborn.countplot.html) function. To create a vertical bar chart, we can use the `x` keyword argument to specify source data; to create a horizontal bar chart, we can use the `y` keyword argument to specify the source data. Below, we create a horizontal bar chart for the vehicle class data.

In [38]:
```
sns.countplot(y=epa_subset.VClass)
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10e56b7f0>
```

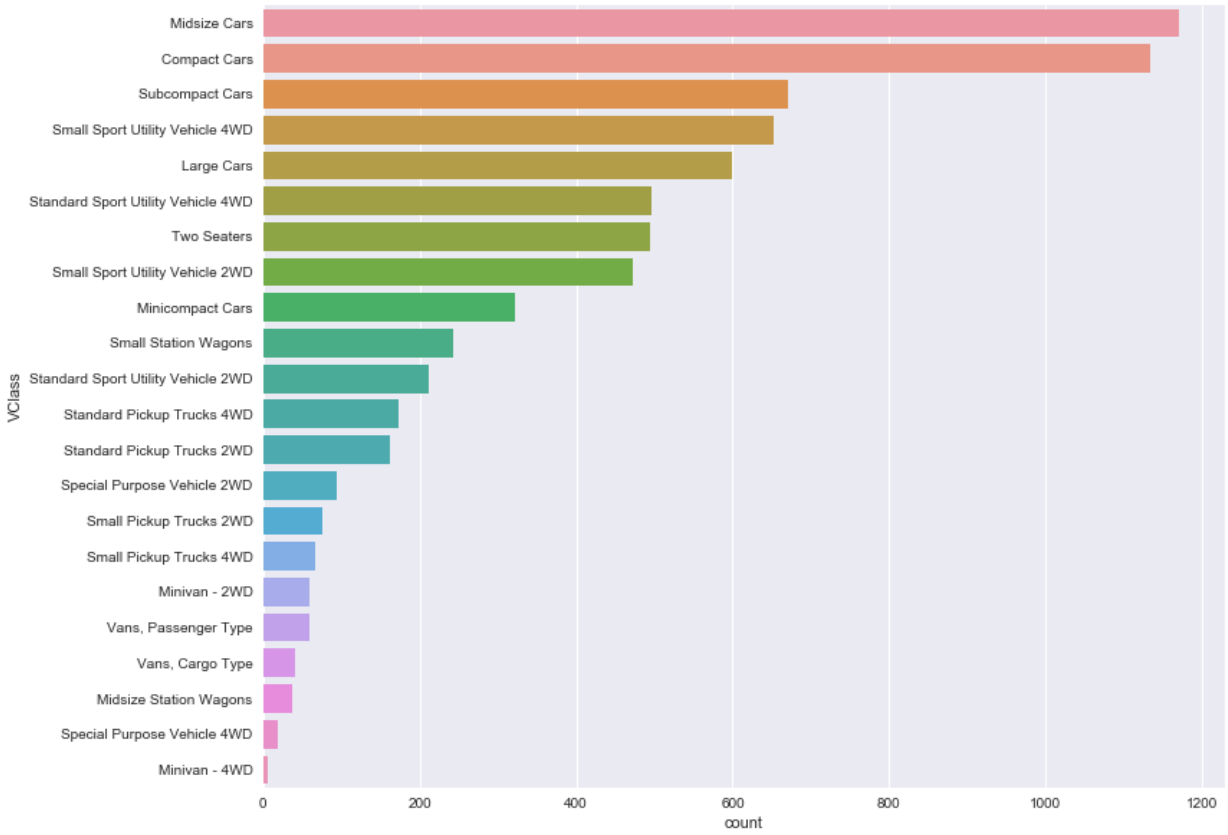


Notice that the data isn't ordered in the same way it was before; by default, categories are ordered based on when they appear in the source data. To impose a different ordering, we can use the `order` keyword argument with the *countplot()* function.

Lab 7 In the cell below, use the Seaborn *countplot()* function to generate a bar chart visualizing the counts for each of the vehicle classes in the `VClass` column. Use the `order` keyword argument to order the categories based on the number of entries for each class. Use the index from the series returned by the `value_counts()` method for ordering information.

In [39]:
```
sns.countplot(y=epa_subset.VClass, order=epa_subset.VClass.value_counts().ir
```

Out[39]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10d756278>
```



As you look at these bar charts, you might notice that there are several vehicle class names that are similar to other classes, for example, "Special Purpose Vehicles" and "Special Purpose Vehicle". Listing the distinct vehicles alphabetically helps to see this better.

In [40]:
```
sorted(epa_subset.VClass.unique().tolist())
```

Out[40]:
```
['Compact Cars',
 'Large Cars',
 'Midsize Cars',
 'Midsize Station Wagons',
 'Minicompact Cars',
 'Minivan - 2WD',
 'Minivan - 4WD',
 'Small Pickup Trucks 2WD',
 'Small Pickup Trucks 4WD',
 'Small Sport Utility Vehicle 2WD',
 'Small Sport Utility Vehicle 4WD',
 'Small Station Wagons',
 'Special Purpose Vehicle 2WD',
 'Special Purpose Vehicle 4WD',
 'Standard Pickup Trucks 2WD',
 'Standard Pickup Trucks 4WD',
 'Standard Sport Utility Vehicle 2WD',
 'Standard Sport Utility Vehicle 4WD',
 'Subcompact Cars',
 'Two Seaters',
 'Vans, Cargo Type',
 'Vans, Passenger Type']
```

Let's clean this data a bit by replacing similar values with one value. In addition to combining categories with similar names, we'll combine two- and four-wheel drive vehicles into the category without an indication of drive and the different types of vans into the general van category.

There are a variety of way of doing this. A for-loop would work but, as mentioned previously, for-loops should be avoided. An alternate way of replacing a column's values is through the use of the *apply()* method that we've used before. We could use the [*map()*] method and supply a dictionary where the keys are correspond to current values in the column and the associated values represent the replacement data, but we have to provide a mapping for every column - even those we don't need to alter.

In [41]:
```
vclass_map = {
    'Minivan - 2WD': 'Minivan',
    'Minivan - 4WD': 'Minivan',
    'Small Pickup Trucks 2WD': 'Small Pickup Trucks',
    'Small Pickup Trucks 4WD': 'Small Pickup Trucks',
    'Small Sport Utility Vehicle 2WD': 'Small Sport Utility Vehicle',
    'Small Sport Utility Vehicle 4WD': 'Small Sport Utility Vehicle',
    'Special Purpose Vehicle': 'Special Purpose Vehicles',
    'Special Purpose Vehicle 2WD': 'Special Purpose Vehicles',
    'Special Purpose Vehicle 4WD': 'Special Purpose Vehicles',
    'Special Purpose Vehicles/2wd': 'Special Purpose Vehicles',
    'Special Purpose Vehicles/4wd': 'Special Purpose Vehicles',
    'Sport Utility Vehicle - 2WD': 'Sport Utility Vehicle',
    'Sport Utility Vehicle - 4WD': 'Sport Utility Vehicle',
    'Standard Pickup Trucks 2WD': 'Standard Pickup Trucks',
    'Standard Pickup Trucks 4WD': 'Standard Pickup Trucks',
    'Standard Pickup Trucks/2wd': 'Standard Pickup Trucks',
    'Standard Sport Utility Vehicle 2WD': 'Standard Sport Utility Vehicle',
    'Standard Sport Utility Vehicle 4WD': 'Standard Sport Utility Vehicle',
    'Vans Passenger': 'Vans',
    'Vans, Cargo Type': 'Vans',
    'Vans, Passenger Type': 'Vans'
}

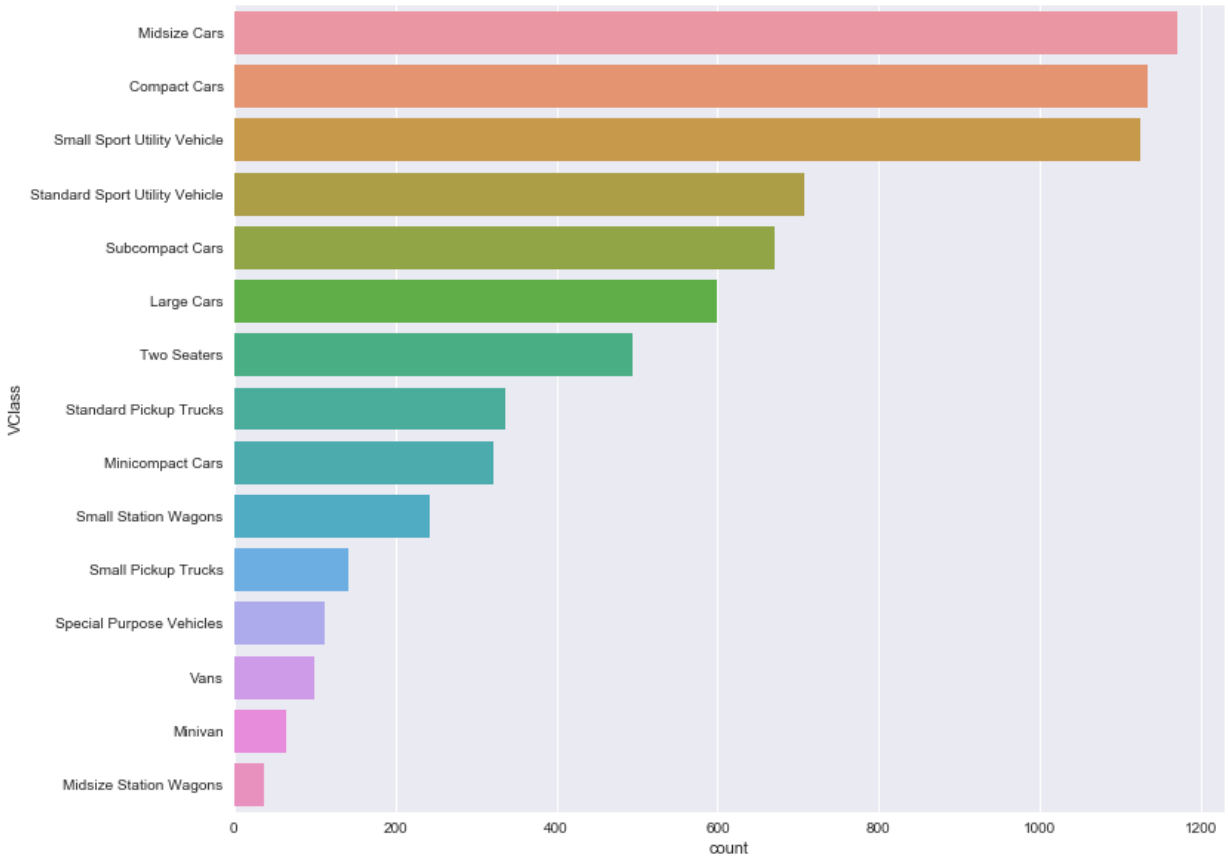
def replace(value):
    if value in vclass_map:
        return vclass_map[value]
    else:
        return value

epa_subset.VClass = epa_subset.VClass.apply(replace)
```

With the similar values replaced, let's look at the value count bar chart again.

In [42]:
```
sns.countplot(y=epa_subset.VClass,
              order=epa_subset.VClass.value_counts().index)
```

Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10e7c2198>
```



As an alternative to how often a given vehicle class appears in the data, we might be interested in knowing the mean fuel economy for each class. In order to this, we can use panda's *group by* functionality (https://pandas.pydata.org/pandas-docs/stable/groupby.html) that allows us to creates groups of data within the dataset and calculate an aggregate value for each group; this is similar to the standard SQL *group by* (https://www.w3schools.com/sql/sql_groupby.asp) statement.

To create a grouping, we can use the DataFrame's *groupby()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html) method specifying at least one column whose values should be used for grouping. The code below groups the data in `epa_subset` by values in the `VClass` column then computes the mean across the other columns for each group.

In [43]:
```
epa_subset.groupby(['VClass']).mean()
```

Out[43]:

VClass	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	di
Compact Cars	23.560459	23.522102	348.318623	348.318623	26.623124	26.583078	4.858782	2.
Large Cars	18.555927	18.555019	429.794658	429.794658	21.494157	21.498619	6.631052	3.
Midsize Cars	23.251067	23.241370	360.021349	360.021349	26.175918	26.174252	5.227156	2.
Midsize Station Wagons	23.289474	23.214797	372.394737	372.394737	25.289474	25.256176	5.473684	3.
Minicompact Cars	20.451713	20.448277	395.666667	395.666667	23.161994	23.152877	5.775701	3.
Minivan	18.584615	18.581455	421.492308	421.492308	21.261538	21.194100	5.815385	3.
Small Pickup Trucks	17.971831	17.924173	453.408451	453.408451	20.035211	19.977349	5.014085	3.
Small Sport Utility Vehicle	20.697509	20.688013	391.183274	391.183274	23.096085	23.106241	4.767794	2.
Small Station Wagons	24.921811	24.881690	337.045267	337.045267	27.530864	27.528605	4.275720	2.
Special Purpose Vehicles	18.176991	18.136774	452.548673	452.548673	20.212389	20.184398	5.274336	3.
Standard Pickup Trucks	15.563798	15.536920	513.646884	513.646884	17.643917	17.597813	7.293769	4.
Standard Sport Utility Vehicle	16.247175	16.199211	498.697740	498.697740	18.312147	18.309386	6.841808	4.
Subcompact Cars	20.358209	20.354097	393.407463	393.407463	23.265672	23.305571	5.685075	3.
Two Seaters	17.983838	18.010229	464.676768	464.676768	20.583838	20.590016	7.030303	3.
Vans	12.150000	12.110525	652.840000	652.840000	13.690000	13.762976	7.780000	5.

If we'd like the aggregated values for a single column, we can specify that column after the call to the the *groupby()* method or after the call to the aggregation function. It's generally better to reduce the size of the dataset over which a calculation is applied so we should select the column of interest before doing the aggregation calculation. We can select the `comb08` column after grouping the data and then calculate the mean.

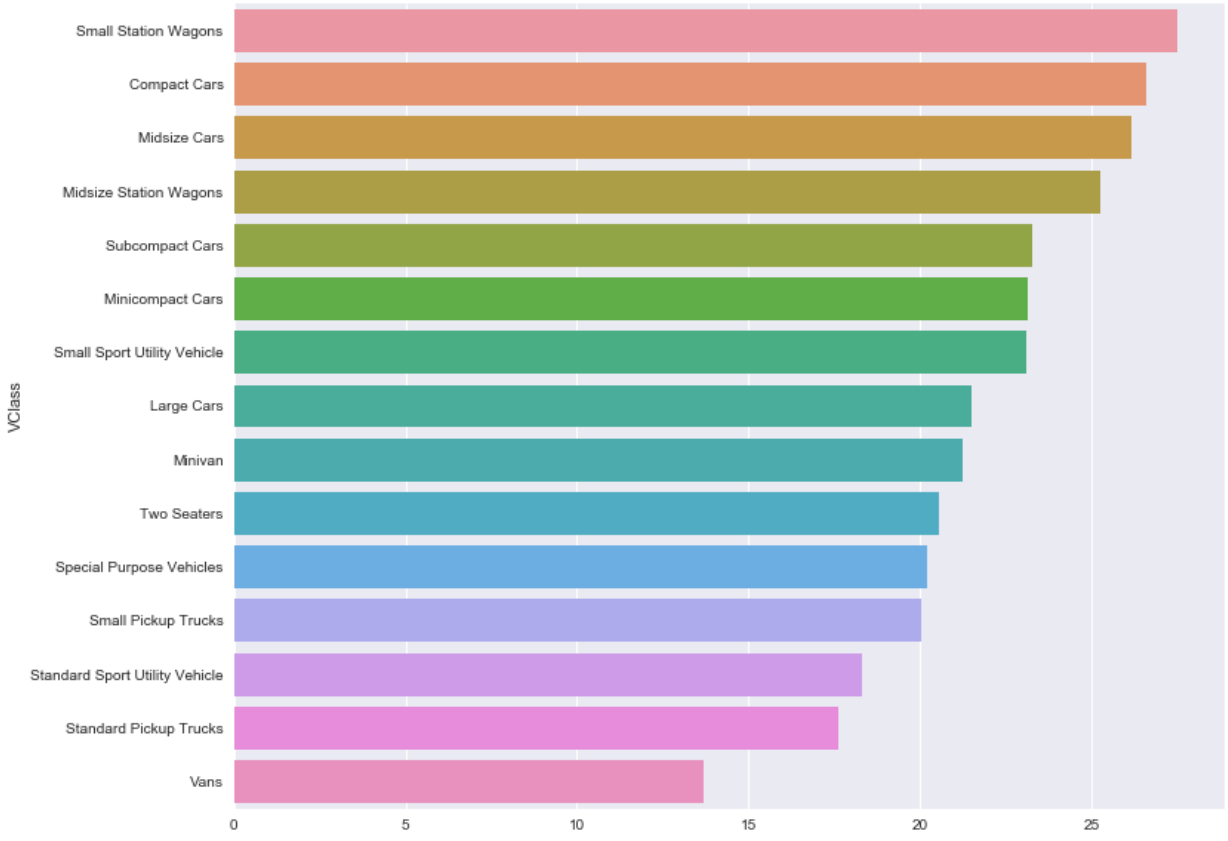
In [44]:
```
mean_mpg_by_vclass = epa_subset.groupby(['VClass'])['comb08'].mean()
display(mean_mpg_by_vclass)
```

```
VClass
Compact Cars                        26.623124
Large Cars                          21.494157
Midsize Cars                        26.175918
Midsize Station Wagons              25.289474
Minicompact Cars                    23.161994
Minivan                             21.261538
Small Pickup Trucks                 20.035211
Small Sport Utility Vehicle         23.096085
Small Station Wagons                27.530864
Special Purpose Vehicles            20.212389
Standard Pickup Trucks              17.643917
Standard Sport Utility Vehicle      18.312147
Subcompact Cars                     23.265672
Two Seaters                         20.583838
Vans                                13.690000
Name: comb08, dtype: float64
```

We can use this aggregated data, stored in a pandas Series, to create a bar chart. Because we are no longer interested in the number of times a `VClass` value appears, we cannot use the Seaborn *countplot()* function. Instead, we can use [*barplot()*]. Before plotting, we sort the Series using the *sort_values()* method to indicate that we'd like to sort the Series by its values; we specify `inplace=True` to alter the existing Series object and `ascending=False` to sort the values in descending order.

In [45]:
```
mean_mpg_by_vclass.sort_values(inplace=True, ascending=False)
sns.barplot(x=mean_mpg_by_vclass.values, y=mean_mpg_by_vclass.index)
```

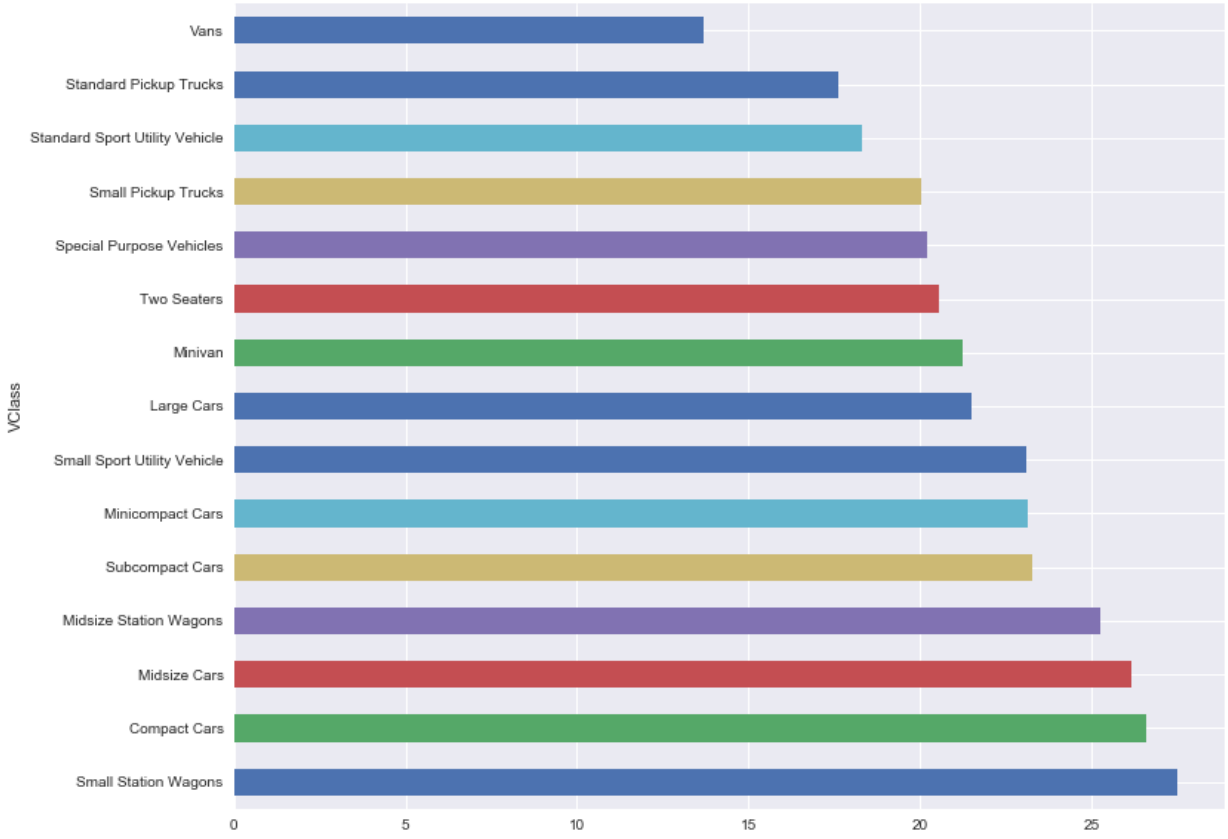
Out[45]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10e7fea90>
```



Alternatively, we could have used the *plot()* method on the Series itself.

In [46]:
```
mean_mpg_by_vclass.plot(kind='barh')
```

Out[46]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10e98a8d0>
```



Plotting Numerical Data with Categorical Data

With the vehicle classes sorted by combined city and highway fuel economy, let's see how some the relationships we looked at in the pair plot above differ based on classes. Let's select three vehicle classes, the one with the greatest mean fuel economy, the one with the least mean fuel economy, and the class corresponding to median of the aggregated values. We can use the *min()*, *max()*, and *median()* methods to find the corresponding values within the Series and use a filter with those values to get the index values (the vehicle class names). The following identifies the vehicle class with the median value for mean combined fuel economy. Note that the filter returns a collection of rows that match the given criteria; to get the first and only result, we use bracket notation.

In [47]:
```
mean_mpg_by_vclass[mean_mpg_by_vclass == mean_mpg_by_vclass.median()].index[
```

Out[47]:
```
'Large Cars'
```

Let's collect the three classes we're interested in into one list.

In [48]:
```
vclass_sample = [
    mean_mpg_by_vclass[mean_mpg_by_vclass == mean_mpg_by_vclass.min()].index
    mean_mpg_by_vclass[mean_mpg_by_vclass == mean_mpg_by_vclass.median()].in
    mean_mpg_by_vclass[mean_mpg_by_vclass == mean_mpg_by_vclass.max()].index
]

vclass_sample
```

Out[48]:
```
['Vans', 'Large Cars', 'Small Station Wagons']
```

We can use this list to reduce the size of our `epa_subset` DataFrame. The mask we'll use to filter the data will rely on the *isin()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.isin.html) method that tests whether a value is in a specified list or not; this is similar to the Python *in* keyword.

In [49]:
```
epa_vclass_sample = epa_subset[epa_subset.VClass.isin(vclass_sample)]
epa_vclass_sample.head()
```

Out[49]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highwa
16839	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	
16840	29	28.5	315	315.0	32	32.3	4.0	2.0	Diesel	
21346	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	
21347	29	28.5	315	315.0	32	32.3	4.0	2.0	Diesel	
21348	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	

We could achieve the same result using *query()* and a more Python-like syntax.

In [50]:
```
epa_vclass_sample = epa_subset.query('VClass in @vclass_sample')
epa_vclass_sample.head()
```

Out[50]:

	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	fuelType1	highwa
16839	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	
16840	29	28.5	315	315.0	32	32.3	4.0	2.0	Diesel	
21346	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	
21347	29	28.5	315	315.0	32	32.3	4.0	2.0	Diesel	
21348	29	29.0	318	318.0	32	32.0	4.0	2.0	Diesel	

With these three vehicle classes, let's look at the pair plot from earlier. We can use the values in the `VClass` column to determine the coloring of markers in the various plots. To to this, we specify the column name with the `hue` argument when we call *pairplot()*.

In [51]:
```
columns = [ "co2", "comb08", "cylinders", "displ", "VClass"]
sns.pairplot(epa_vclass_sample[columns], hue="VClass")
```

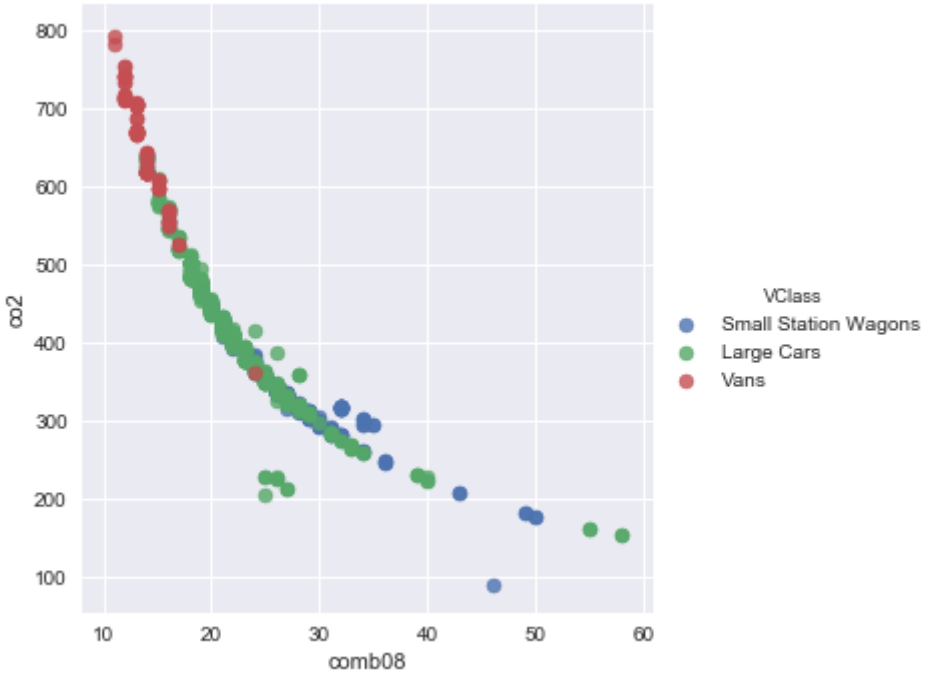
Out[51]: <seaborn.axisgrid.PairGrid at 0x10ec57cc0>

Looking at the data with how we chose values for `VClass` in mind, we can begin to see some trends. For example, the more fuel-efficient vehicles, like small station wagons, have engines with a smaller displacement and emit less carbon dioxide. Similarly, vans, which represent the least fuel-efficient vehicles, have engines with a greater displacement and emit more carbon dioxide.

We can look at pairwise relationships one at a time if we would like. Let's look at the scatter plot of `comb08` and `co2`. While we could use the *scatter()* method we used earlier, it would take some work to add the marker coloring based on `VClass` that we have in the pair plot above. Instead, we'll use the Seaborn *lmplot()* (https://seaborn.pydata.org/generated/seaborn.lmplot.html) function. By default, this plot type will also show the linear regression that fits the data. We'll disable this feature for now but will use it later.

In [52]:
```
sns.lmplot(x='comb08', y="co2", hue='VClass',
           data=epa_vclass_sample, fit_reg=False)
```

Out[52]:
```
<seaborn.axisgrid.FacetGrid at 0x110466128>
```

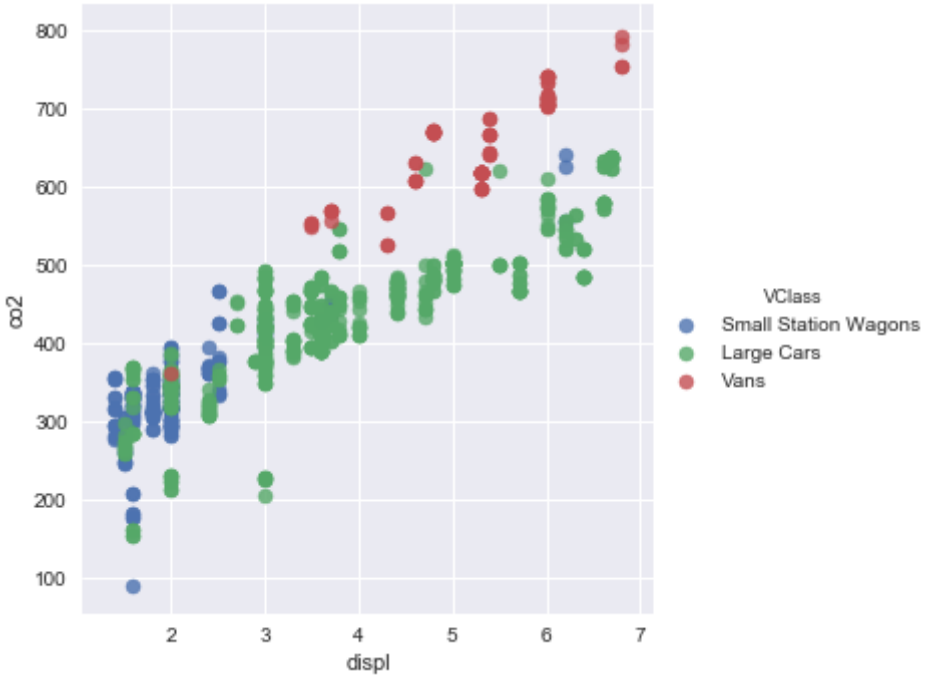


We can see that there appears to be an indirect relationship between the two variables: as fuel economy increases, carbon dioxide emissions decrease.

Lab 8 In the cell below, create a similar scatter plot for `displ` and `co2` with marker coloring determined by vehicle class.

In [53]:
```
sns.lmplot(x="displ", y="co2", hue='VClass', data=epa_vclass_sample, fit_reg
```

Out[53]:
```
<seaborn.axisgrid.FacetGrid at 0x110459588>
```



Box Plots

We'll look at modeling these relationships in the next unit, but for now, let's return to the larger subset of data with all the vehicle classes. So far, we've used the *describe()* method, histograms, and bar charts to get a sense of the distribution and other properties of some of the numeric data in our dataset. Another common way to explore numeric data is through the use of box plots (https://en.wikipedia.org/wiki/Box_plot).

To understand the components of a box plot, consider the following example Series.

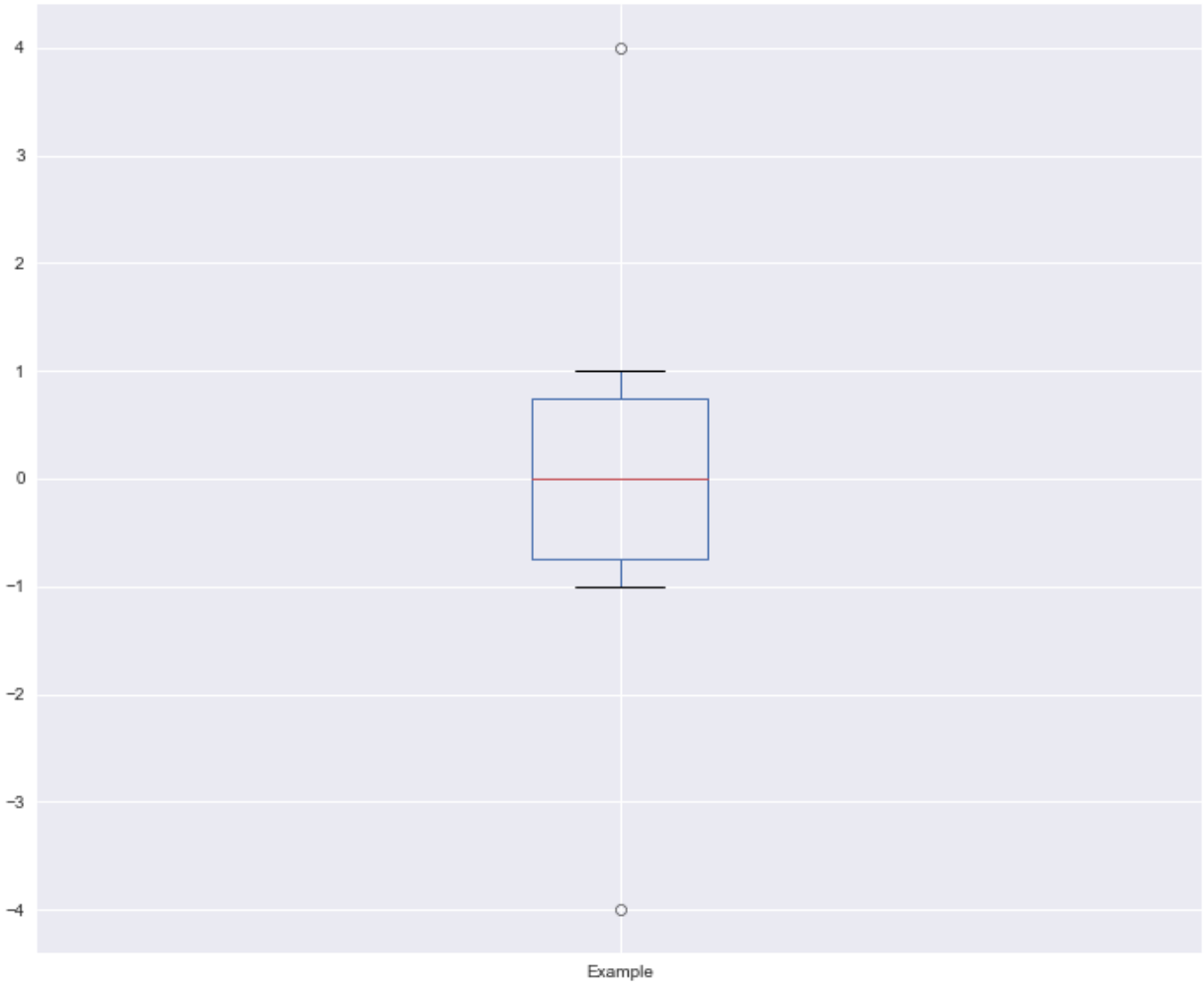
In [54]:
```
example = pd.Series([-4, -1, -0.5, 0, 0.5, 1, 4], name="Example")
example.describe()
```

Out[54]:
```
count    7.000000
mean     0.000000
std      2.397916
min     -4.000000
25%     -0.750000
50%      0.000000
75%      0.750000
max      4.000000
Name: Example, dtype: float64
```

Let's create a the box plot for the same Series and compare it to the output above.

```
In [55]: example.plot(kind='box')
```

```
Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x110690c50>
```

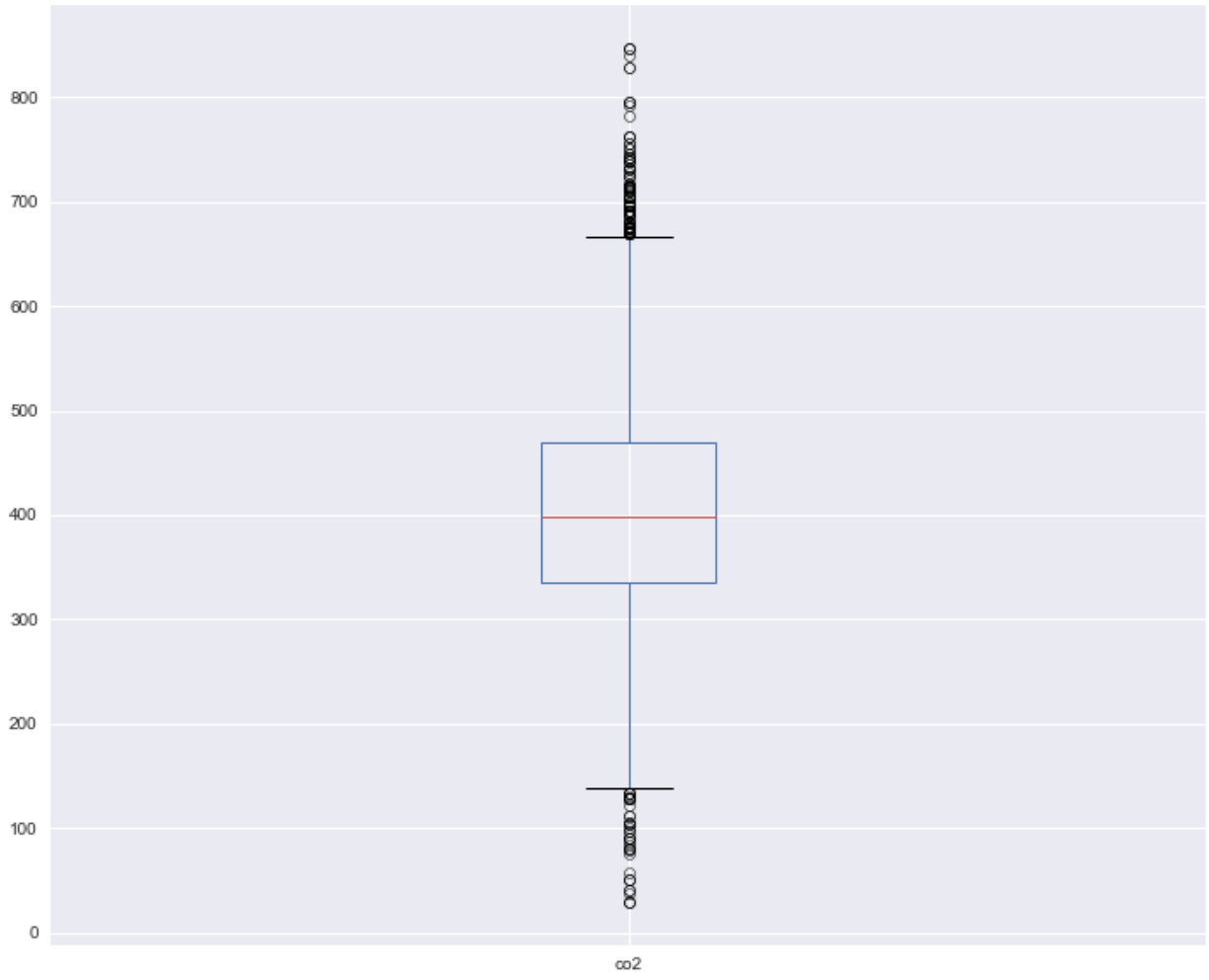


The red line in the box corresponds to the median or 50th percentile value. The lines above and below the median correspond to the 75th and 25th percentile values, respectively. The narrower lines above and below these lines correspond to the maximum and minimum values excluding outliers. Outliers are symbolized by the small circular markers. Outliers represent values that can be considered "distant" from other values. There are various ways of defining what constitutes an outlier but a common method relies on interquartile range, *IQR*, the difference between the third and first quartile, or in terms of percentiles, the difference between the 75th and 25th percentiles. Using interquartile range a value can be considered an outlier if it satisfies one of the following:

- it is greater than the sum of the 75th percentile and 1.5 times the interquartile range
- it is less than the difference of 25th percentile and 1.5 times the interquartile range.

In [56]:
```
epa_subset.co2.plot(kind='box')
```

Out[56]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1106accc0>
```

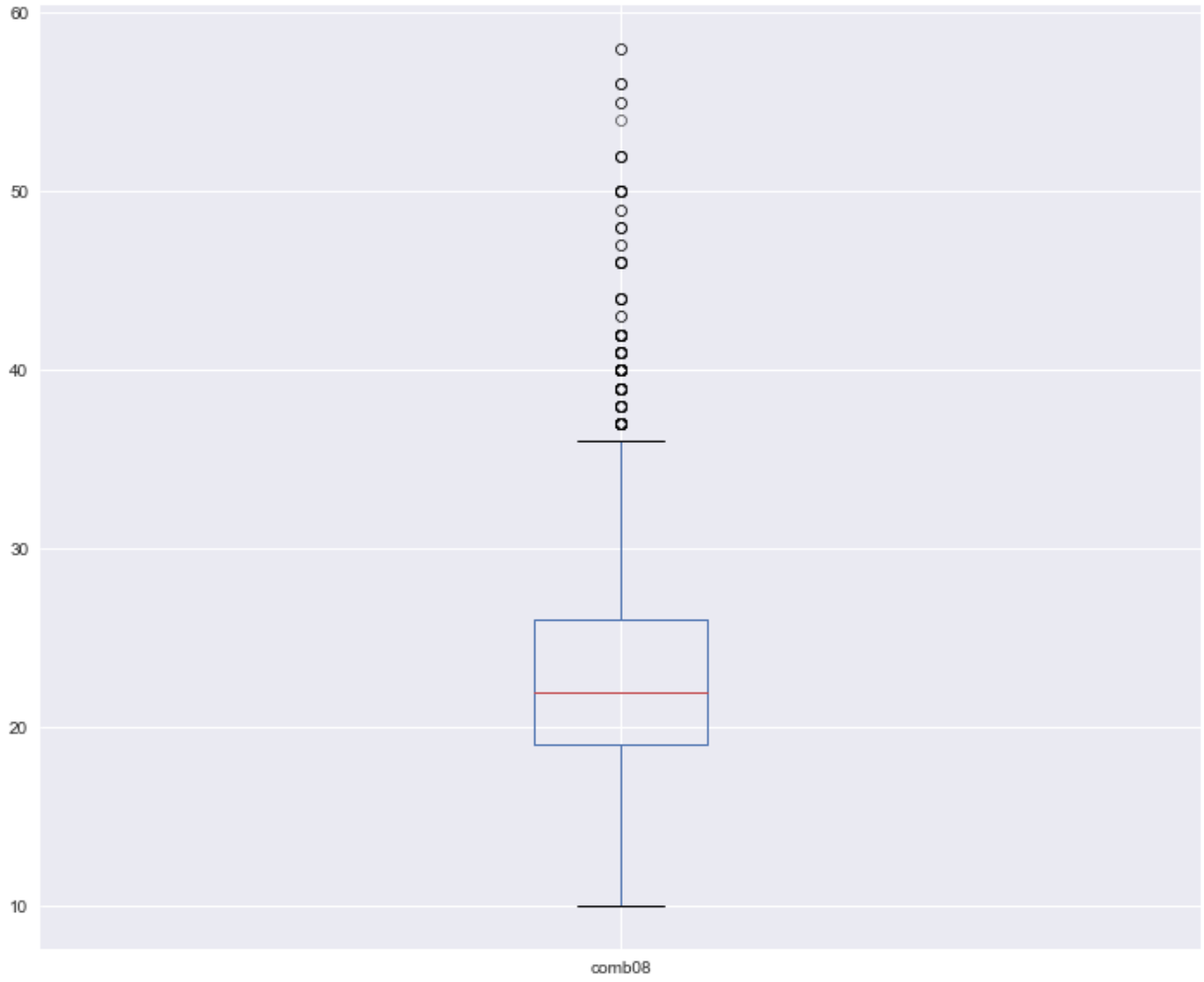


We can see that the median is about 400 g/mile and that there are quite a few outliers.

Next, let's look at the `comb08` and `cylinders` columns.

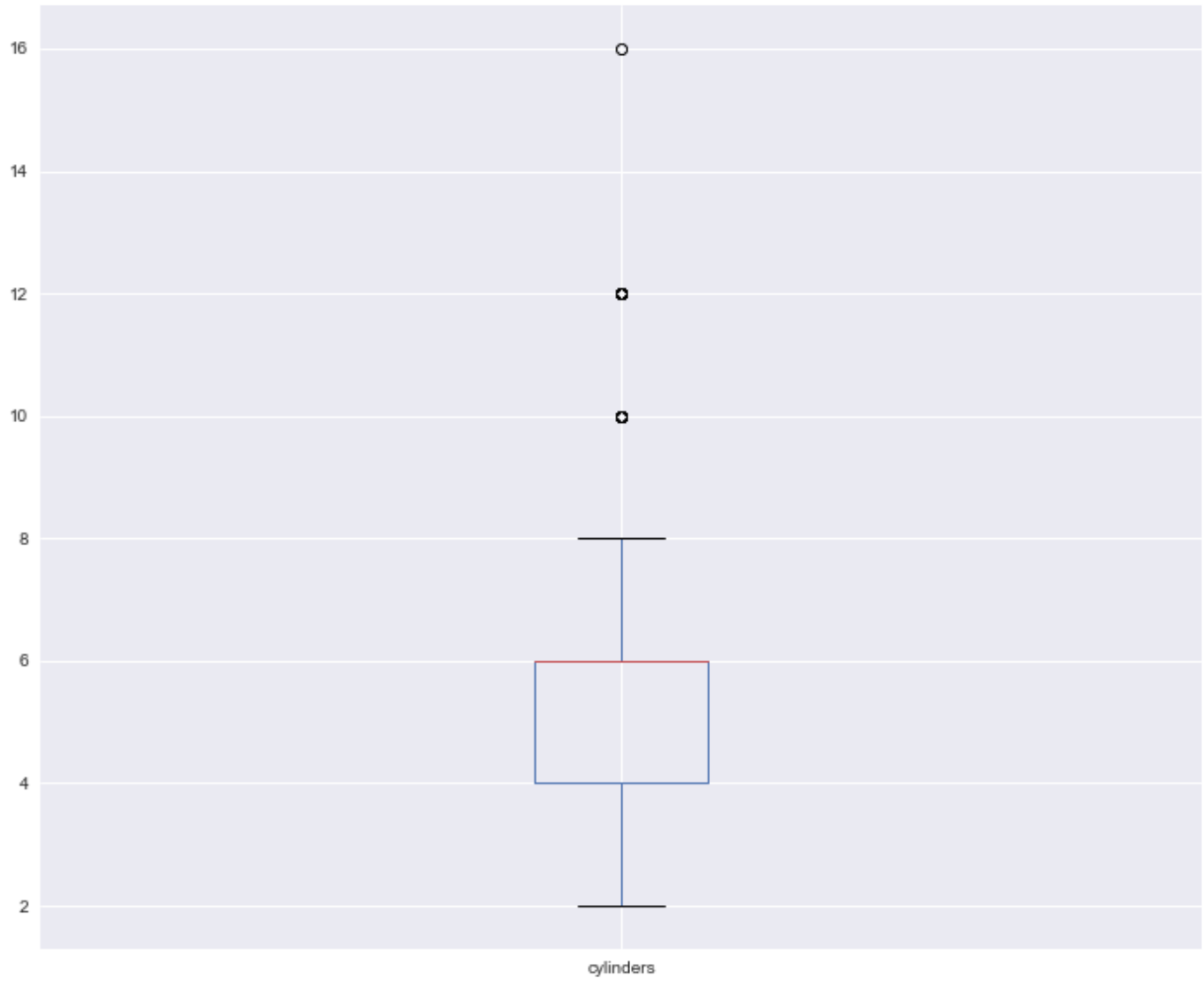
In [57]:
```
epa_subset.comb08.plot(kind='box')
```

Out[57]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1107968d0>
```



In [58]:
```
epa_subset.cylinders.plot(kind='box')
```

Out[58]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1107b4630>
```



The box plot for `cylinders` is a bit unusual. We see that the median and the 75th percentile values are the same. This is due to the fact that there are few different values and the distribution of those values as shown by *describe()* and *value_counts()* below.

In [59]:
```
display(epa_subset.cylinders.describe())
display(epa_subset.cylinders.value_counts())
```

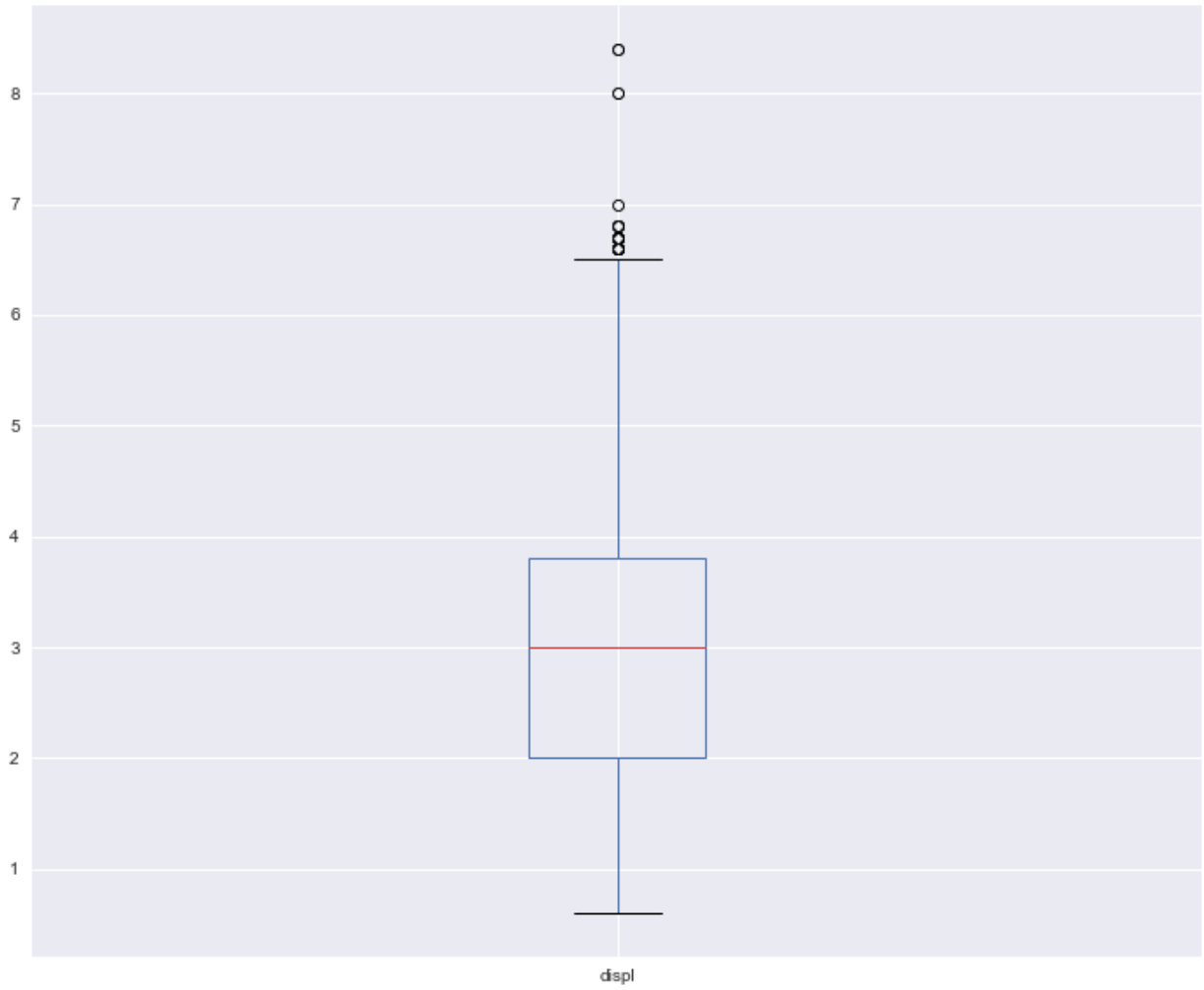
```
count    7259.000000
mean        5.663728
std         1.872197
min         2.000000
25%         4.000000
50%         6.000000
75%         6.000000
max        16.000000
Name: cylinders, dtype: float64
```

```
4.0     3092
6.0     2416
8.0     1398
12.0     176
3.0       84
10.0      42
5.0       41
2.0        6
16.0       4
Name: cylinders, dtype: int64
```

Lab 9 In the cell below, create a box plot for the `displ` column of the `epa_subset` DataFrame.

In [60]:
```
epa_subset.displ.plot(kind='box')
```

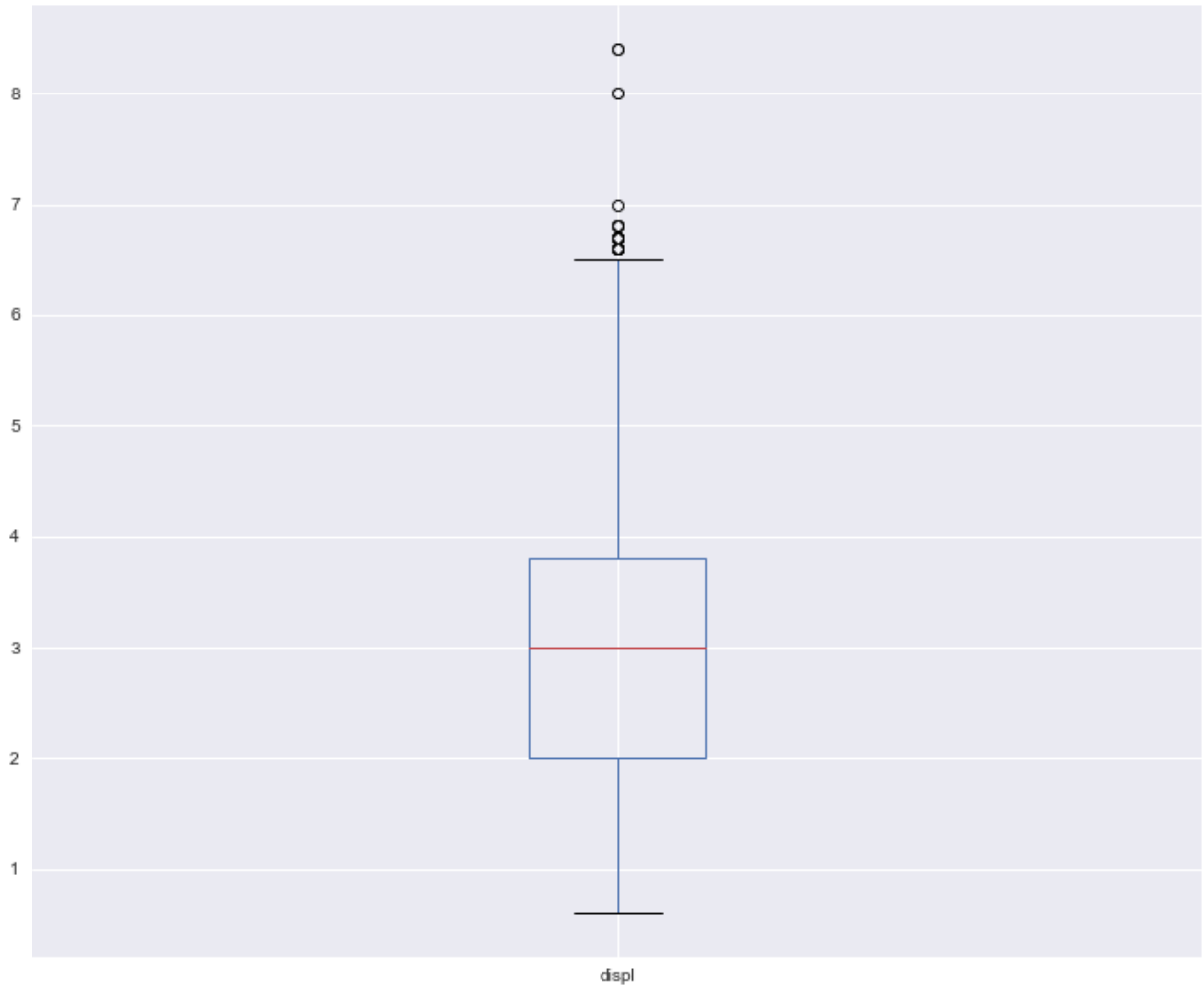
Out[60]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x110ac5898>
```



Let's look at the box plot for the `displ` column again.

In [61]:
```
epa_subset.displ.plot(kind='box')
```

Out[61]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x110b2e320>
```



Let's calculate thresholds for the outliers.

In [62]:
```
q1 = epa_subset.displ.quantile(0.25)  # first quartile
q3 = epa_subset.displ.quantile(0.75)  # third quartile
IQR = q3 - q1  # IQR
lower_threshold = q1 - 1.5 * IQR  # lower threshold
upper_threshold = q3 + 1.5 * IQR  # upper threshold
display(lower_threshold, upper_threshold)
```

```
-0.6999999999999997
```

```
6.5
```

From these calculations, an outlier for engine displacement is anything greater than 6.5 or less than about -0.7; for this data there are no lower outliers as negative displacement is meaningless.

Lab 10 In the cell below, calculate the lower and upper thresholds for outliers for the `co2` column in the `epa_subset` DataFrame.

In [63]:
```
q1 = epa_subset.co2.quantile(0.25)
q3 = epa_subset.co2.quantile(0.75)
IQR = q3 - q1
lower_threshold = q1 - 1.5 * IQR
upper_threshold = q3 + 1.5 * IQR
display(lower_threshold, upper_threshold)
```

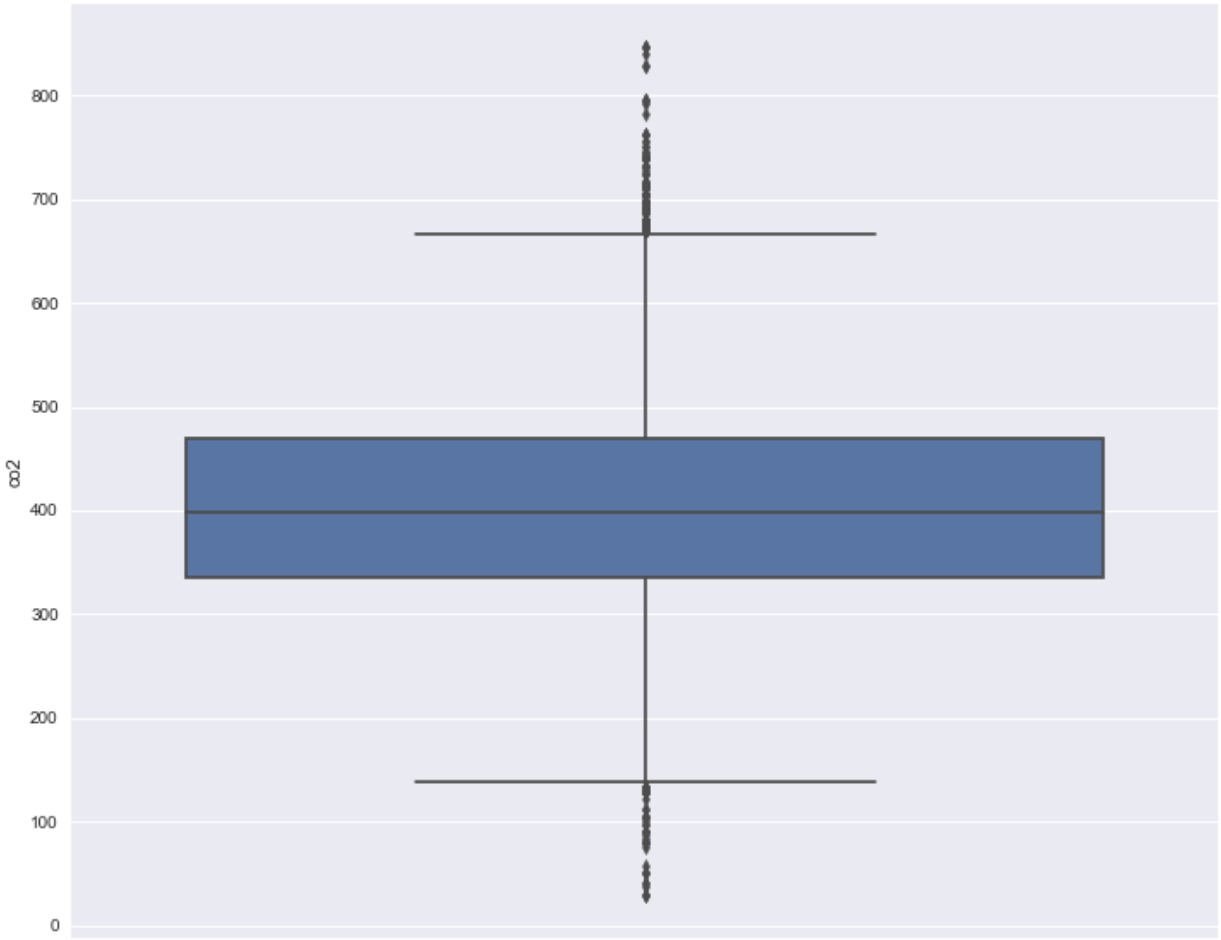
```
136.5
```

```
668.5
```

Seaborn also includes functionality to create box plots using the *boxplot()* (https://seaborn.pydata.org/generated/seaborn.boxplot.html) function but the pandas DataFrame *plot()* method tends to produce easier-to-read plots.

In [64]:
```
sns.boxplot(y=epa_subset.co2)
```

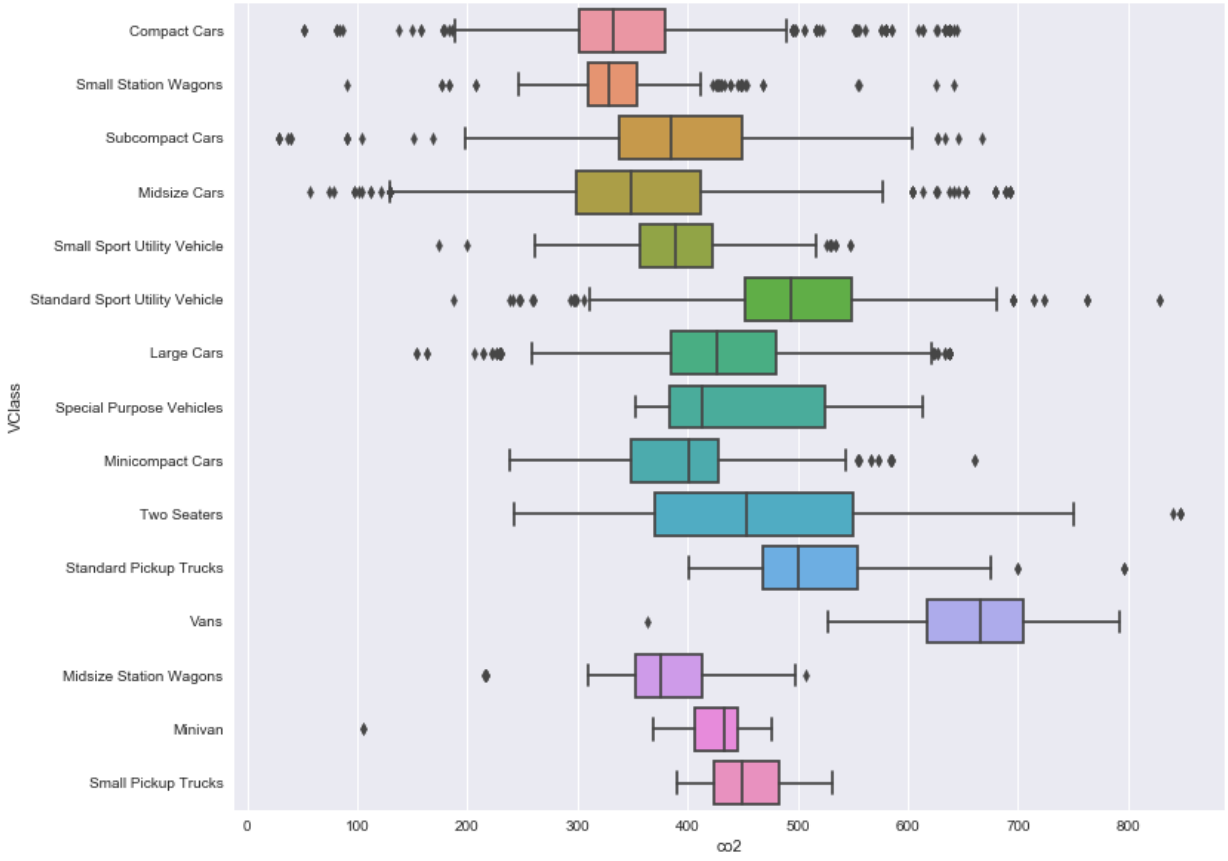
Out[64]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x110cc9e48>
```



Seaborn's *boxplot()* function does work well when we want to show the box plots from one column separated by categorical values.

In [65]:
```
sns.boxplot(x="co2", y="VClass", data=epa_subset)
```

Out[65]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x110e42550>
```



Pivot Tables

Pandas provides the ability to create pivot tables (https://en.wikipedia.org/wiki/Pivot_table) to aggregate and summarize data. To crate a pivot table, we can use the DataFrame's *pivot_table()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.pivot_table.html) method and, at a minimum, specifying which column should be used as the index of the new table; the index serves as the column by which values are grouped and aggregated. By default, the mean is used as the aggregation function but we can specify any appropriate function using the `aggfunc` keyword argument.

Let's create a pivot table that groups the data by `year` and aggregates data by calculating the median of values.

In [66]:
```
epa_subset.pivot_table(index="year", aggfunc=pd.np.median)
```

Out[66]:

year	city08	city08U	co2	co2TailpipeGpm	comb08	comb08U	cylinders	displ	highway08	high
2009	29.0	28.75000	316.5	316.5	32.0	32.15000	4.0	2.00	38.0	
2010	29.0	29.00000	318.0	318.0	32.0	32.00000	4.0	2.00	37.0	
2011	29.0	28.75000	316.5	316.5	32.0	32.00000	4.0	2.00	38.0	
2012	29.0	29.00000	315.0	315.0	32.0	32.15000	4.0	2.00	38.0	
2013	19.0	18.84565	415.0	415.0	21.0	21.42490	6.0	3.35	26.0	
2014	19.0	19.00030	407.0	407.0	22.0	21.85650	6.0	3.30	27.0	
2015	19.0	19.43940	398.0	398.0	22.0	22.28780	6.0	3.00	27.0	
2016	20.0	19.75200	396.0	396.0	23.0	22.55075	6.0	3.00	27.0	
2017	20.0	19.90000	391.0	391.0	23.0	22.67035	6.0	3.00	27.0	
2018	20.0	19.94125	389.5	389.5	23.0	22.75800	6.0	3.00	27.0	
2019	18.5	18.81610	408.0	408.0	21.5	21.72625	6.0	3.20	26.5	

We can choose a subset of columns by listing them using the `values` keyword argument.

In [67]:
```
epa_subset.pivot_table(index="year",
                       values=["city08", "comb08", "highway08"],
                       aggfunc=pd.np.median)
```

Out[67]:

year	city08	comb08	highway08
2009	29.0	32.0	38.0
2010	29.0	32.0	37.0
2011	29.0	32.0	38.0
2012	29.0	32.0	38.0
2013	19.0	21.0	26.0
2014	19.0	22.0	27.0
2015	19.0	22.0	27.0
2016	20.0	23.0	27.0
2017	20.0	23.0	27.0
2018	20.0	23.0	27.0
2019	18.5	21.5	26.5

We fan also further divide the data by specifying additional indexes or through the `columns` keyword argument.

In [68]:
```
epa_subset.pivot_table(index=["year", "fuelType1"],
                       values=["city08", "comb08", "highway08"],
                       aggfunc=pd.np.median)
```

Out[68]:

		city08	comb08	highway08
year	**fuelType1**			
2009	**Diesel**	29.0	32.0	38.0
2010	**Diesel**	29.0	32.0	37.0
2011	**Diesel**	29.0	32.0	37.0
	Premium Gasoline	35.0	37.0	40.0
2012	**Diesel**	29.0	32.0	37.0
	Premium Gasoline	27.5	28.5	30.5
	Regular Gasoline	51.0	50.0	49.0
2013	**Diesel**	27.0	30.0	37.0
	Midgrade Gasoline	15.0	17.0	21.0
	Natural Gas	27.0	31.0	38.0
	Premium Gasoline	18.0	21.0	26.0
	Regular Gasoline	20.0	22.0	27.0
2014	**Diesel**	27.0	30.5	37.5
	Midgrade Gasoline	15.0	17.0	22.0
	Natural Gas	19.5	22.0	27.0
	Premium Gasoline	18.0	21.0	26.0
	Regular Gasoline	20.0	23.0	28.0
2015	**Diesel**	27.5	32.0	38.5
	Midgrade Gasoline	14.0	17.0	22.0
	Natural Gas	27.0	31.0	38.0
	Premium Gasoline	19.0	22.0	26.0
	Regular Gasoline	20.0	23.0	28.0
2016	**Diesel**	23.0	25.5	30.0
	Midgrade Gasoline	14.5	17.0	22.0
	Premium Gasoline	19.0	22.0	26.0
	Regular Gasoline	21.0	24.0	28.0
2017	**Diesel**	23.0	25.0	30.0
	Midgrade Gasoline	16.0	19.0	24.0
	Premium Gasoline	19.0	22.0	27.0
	Regular Gasoline	21.0	24.0	28.0
2018	**Diesel**	28.0	32.0	38.5

		city08	comb08	highway08
year	fuelType1			
	Midgrade Gasoline	16.0	19.0	24.0
	Premium Gasoline	20.0	22.0	27.0
	Regular Gasoline	21.0	23.0	28.0
2019	Midgrade Gasoline	18.5	21.5	26.5
	Premium Gasoline	15.0	18.0	25.0
	Regular Gasoline	30.0	34.0	40.0

In [69]:
```
epa_subset.pivot_table(index="year",
                       columns="fuelType1",
                       values=["city08", "comb08", "highway08"],
                       aggfunc=pd.np.median)
```

Out[69]:

	city08					comb08				
fuelType1	Diesel	Midgrade Gasoline	Natural Gas	Premium Gasoline	Regular Gasoline	Diesel	Midgrade Gasoline	Natural Gas	Premium Gasoline	Reg Gas
year										
2009	29.0	NaN	NaN	NaN	NaN	32.0	NaN	NaN	NaN	
2010	29.0	NaN	NaN	NaN	NaN	32.0	NaN	NaN	NaN	
2011	29.0	NaN	NaN	35.0	NaN	32.0	NaN	NaN	37.0	
2012	29.0	NaN	NaN	27.5	51.0	32.0	NaN	NaN	28.5	
2013	27.0	15.0	27.0	18.0	20.0	30.0	17.0	31.0	21.0	
2014	27.0	15.0	19.5	18.0	20.0	30.5	17.0	22.0	21.0	
2015	27.5	14.0	27.0	19.0	20.0	32.0	17.0	31.0	22.0	
2016	23.0	14.5	NaN	19.0	21.0	25.5	17.0	NaN	22.0	
2017	23.0	16.0	NaN	19.0	21.0	25.0	19.0	NaN	22.0	
2018	28.0	16.0	NaN	20.0	21.0	32.0	19.0	NaN	22.0	
2019	NaN	18.5	NaN	15.0	30.0	NaN	21.5	NaN	18.0	

We an also use pivot tables as inputs for visualizations. Let's compare city and highway fuel economy for the various vehicle classes.

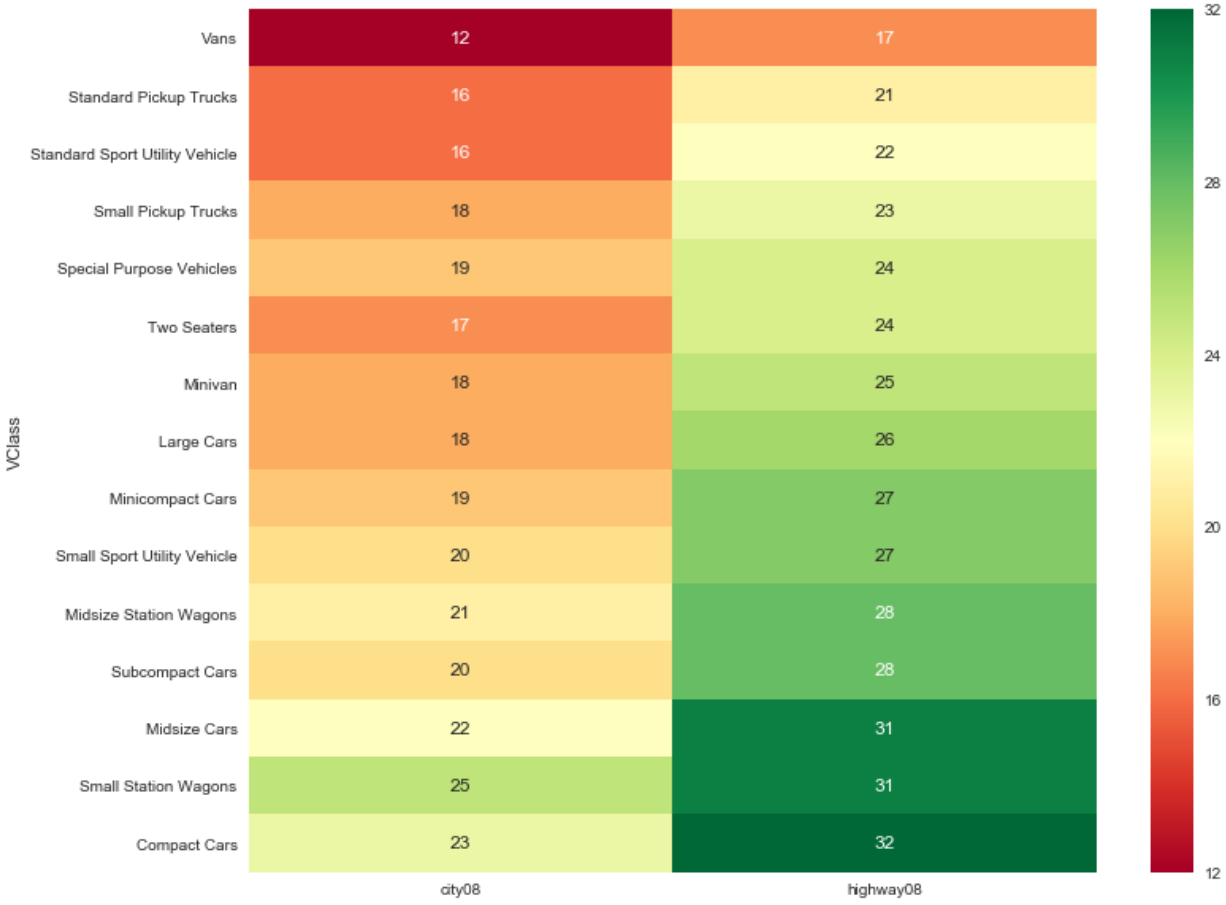
In [70]:
```
pivot = (
    epa_subset.pivot_table(
        index=["VClass"],
        values=["city08", "highway08"],
        aggfunc=pd.np.median)
    .sort_values(by="highway08"))
display(pivot)
```

VClass	city08	highway08
Vans	12	17
Standard Pickup Trucks	16	21
Standard Sport Utility Vehicle	16	22
Small Pickup Trucks	18	23
Special Purpose Vehicles	19	24
Two Seaters	17	24
Minivan	18	25
Large Cars	18	26
Minicompact Cars	19	27
Small Sport Utility Vehicle	20	27
Midsize Station Wagons	21	28
Subcompact Cars	20	28
Midsize Cars	22	31
Small Station Wagons	25	31
Compact Cars	23	32

With this we can create a heatmap (https://en.wikipedia.org/wiki/Heat_map) using Seaborn's *heatmap()* (https://seaborn.pydata.org/generated/seaborn.heatmap.html) function. In addition to specifying the pivot table as the data source, we can show the values for each rectangle using the `annot` keyword argument and specify the colormap using the `cmap` keyword argument with a matplotlib colormap name (https://matplotlib.org/users/colormaps.html).

In [71]:
```
sns.heatmap(pivot, annot=True, cmap="RdYlGn")
```

Out[71]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1112200b8>
```



Lab Answers

1.
```
employees.query(
    "city == 'Columbus' and "
    "department == 'HR'"
)
```

2.
```
columns = ["city08", "city08U", "co2", "co2TailpipeGpm", "comb08", "comb08U",
           "cylinders", "displ", "fuelType1", "highway08", "highway08U",
           "make", "model", "VClass", "year"]
epa_subset = epa_data[columns].copy()
epa_subset.head()
```

3.
```
epa_subset.describe()
```

4.
```
epa_subset.drop_duplicates(inplace=True)
```

```
display(epa_subset.city08.skew())
display(epa_subset.city08.kurtosis())
```

6. ```
   epa_subset.plot.scatter(x="year", y="comb08")
   ```

   and

   ```
   epa_subset.plot.scatter(x="year", y="comb08U")
   ```

7. ```
   sns.countplot(y=epa_subset.VClass, order=epa_subset.VClass.value_counts().index)
   ```

8. ```
   sns.lmplot(x="displ", y="co2", hue='VClass', data=epa_vclass_sample, fit_reg=False)
   ```

9. ```
   epa_subset.displ.plot(kind='box')
   ```

10. ```
    q1 = epa_subset.co2.quantile(0.25)
    q3 = epa_subset.co2.quantile(0.75)
    IQR = q3 - q1
    lower_threshold = q1 - 1.5 * IQR
    upper_threshold = q3 + 1.5 * IQR
    display(lower_threshold, upper_threshold)
    ```

Next Steps

We've identified some potential relationships among columns within the fuel economy dataset. In the next unit we'll create mathematical models for some of these relationships and see how well the model fits the existing data.

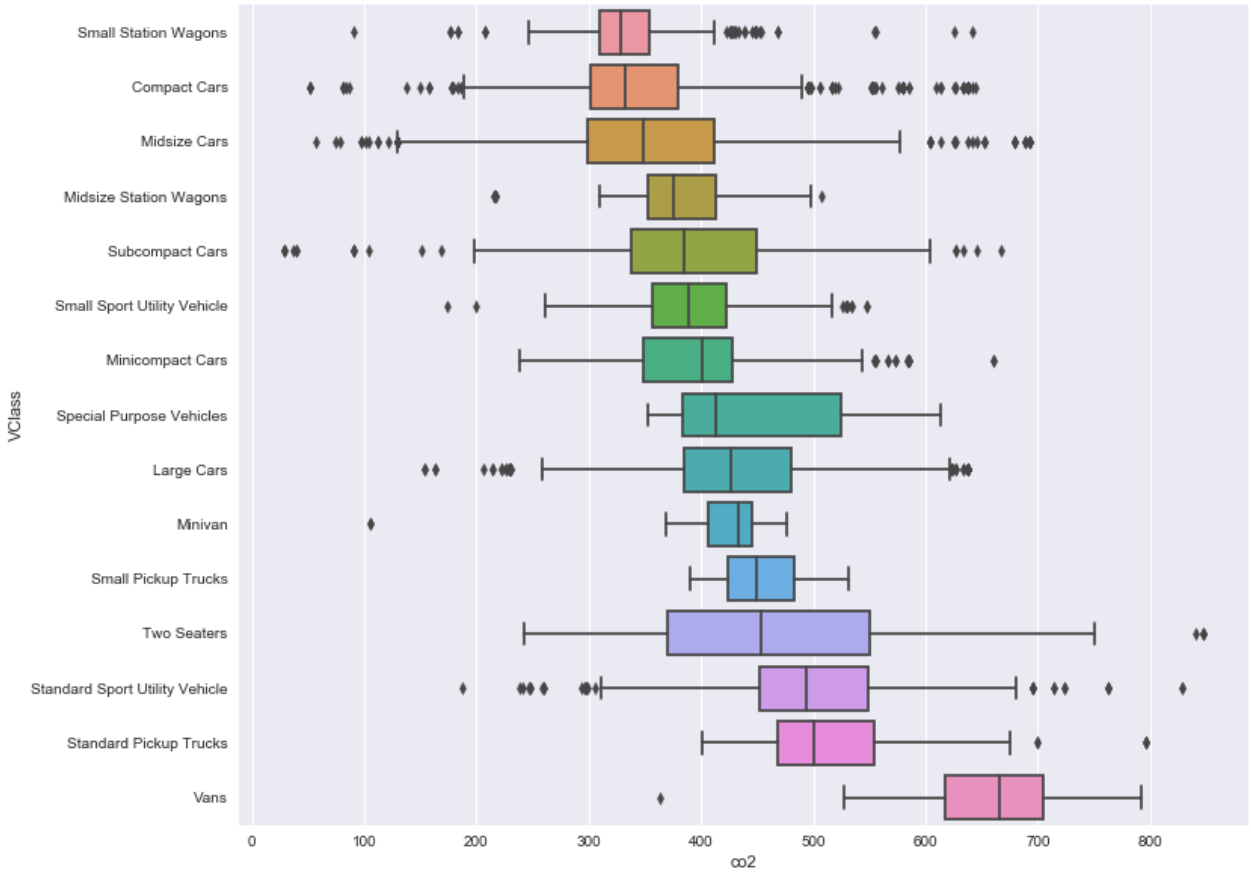
We'll continue to use plots and charts to understand the data but mostly for exploratory purposes. Visualizations also serve as great tools for conveying information; we'll explore explanatory visualizations later.

Resources and Further Reading

- An Introduction to Seaborn (https://seaborn.pydata.org/introduction.html)
- Seaborn Gallery (https://seaborn.pydata.org/examples/index.html)
- *Practical Statistics for Data Scientists* by Bruce and Bruce, Chapter 1: Exploratory Data Analysis (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/databases/9781491952955/1dot-exploratory-data-analysis/eda_html?uicode=ohlink)
- *Python: Data Analytics and Visualization* by Phuong, et. al., Data Exploration (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781788290 logistic-regression-with-python/ch06lvl2sec00083_html?uicode=ohlink)

Exercises
    ```

1. We created a set of box plots for the values in the `co2` column separated by `VClass` from the `epa_data` DataFrame using `sns.boxplot(x="co2", y="VClass", data=epa_subset)`. Prior to that, when working with bar charts, we specified an ordering for the categorical data using the `order` keyword argument. Modify the box plot so categories are ordered by median carbon dioxide emissions from least to greatest. See the image below for the desired plot.
2. Load county auditor data, either one county's data from one of the data sources we've used or from the combined data we saved to a database, and create a pair plot that compares sales price, area, number of bedrooms, and bathrooms. Are there any potential relationships between any of these variables?



Exercise 1 - Box plots for carbon dioxide emission by vehicle class sorted by median emission

In [ ]: