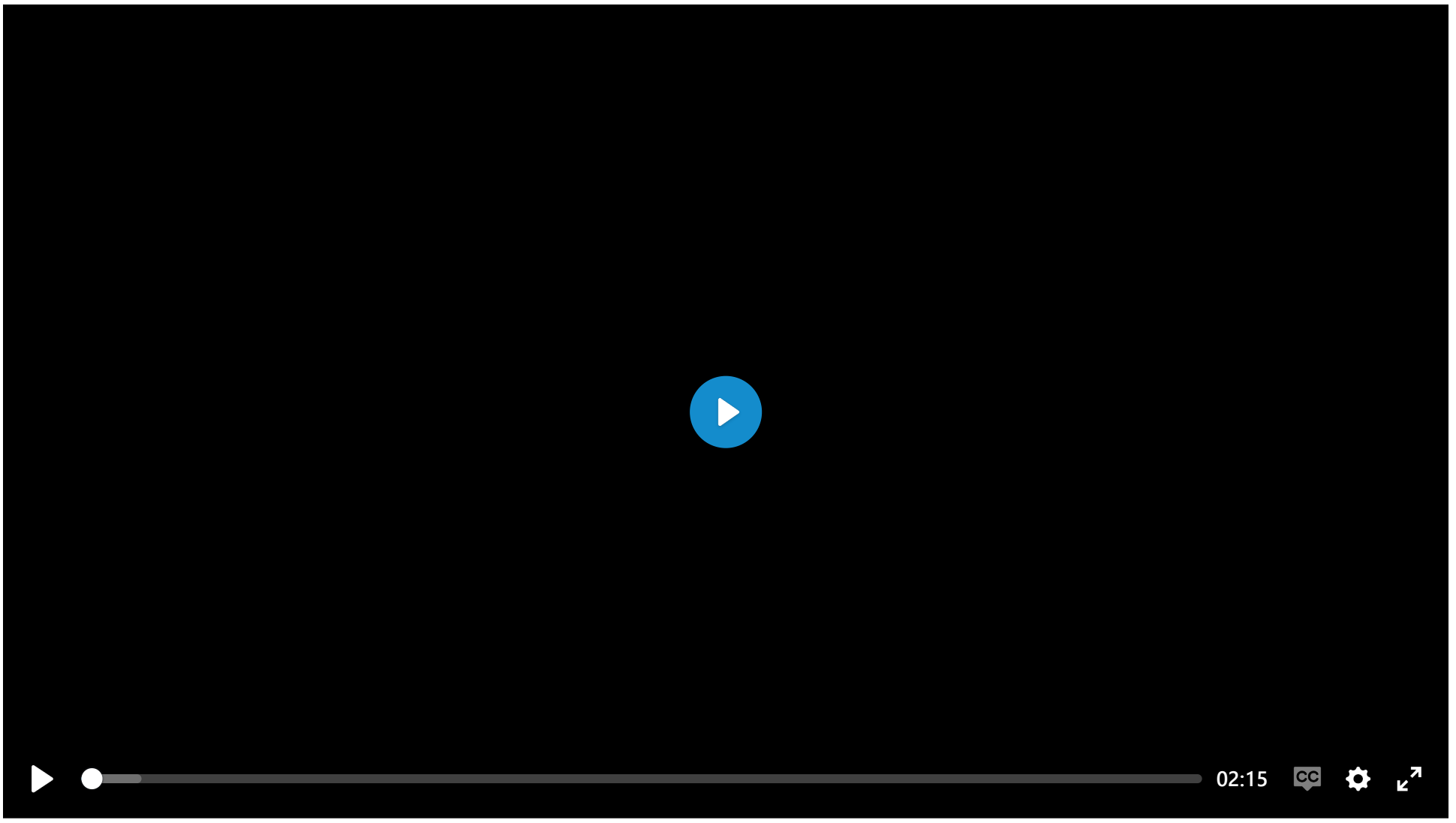
03. Overplotting, Transparency, and Jitter

L4 031 Overplotting Transparency And Jitter 1 V4

Data Vis L4 C03 V1



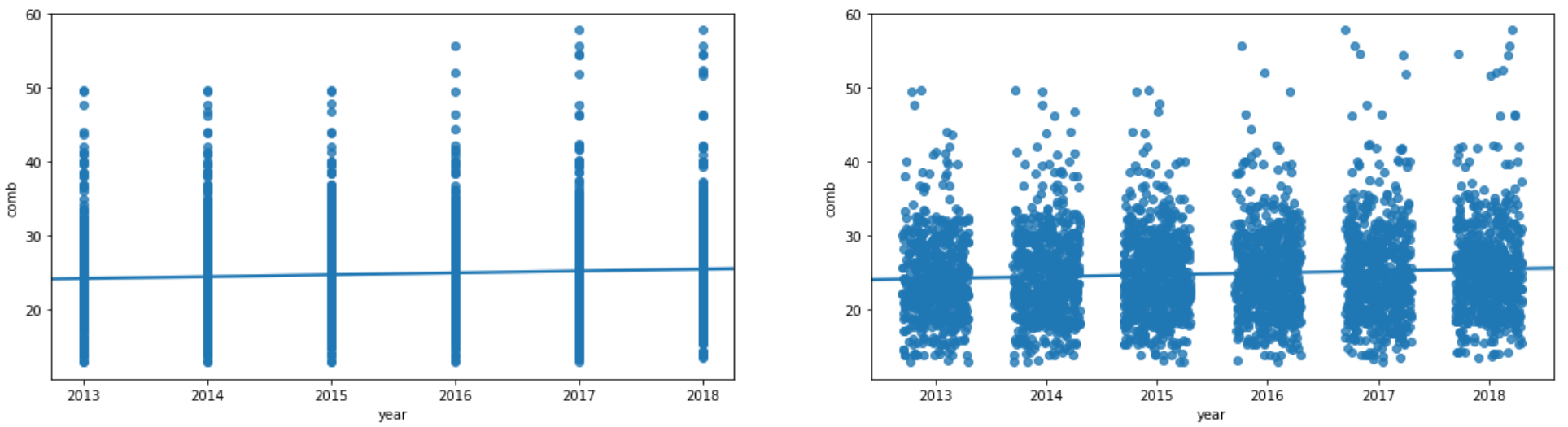
Overplotting, Transparency, and Jitter

If we have a very large number of points to plot or our numeric variables are discrete-valued, then it is possible that using a scatterplot straightforwardly will not be informative. The visualization will suffer from *overplotting*, where the high amount of overlap in points makes it difficult to see the actual relationship between the plotted variables.

Let's see an example below for each Jitter to move the position of each point slightly from its true value. Jitter is not a direct option in matplotlib's `scatter()` function, but is a built-in option with seaborn's regplot() function. The x- and y- jitter can be added independently, and won't affect the fit of any regression function, if made.

Example 1. Jitter - Randomly add/subtract a small value to each data point

```
# TO DO: Necessary import

# Read the CSV file
fuel_econ = pd.read_csv('fuel_econ.csv')
fuel_econ.head(10)

##########################################
# Resize figure to accommodate two plots
plt.figure(figsize = [20, 5])

# PLOT ON LEFT - SIMPLE SCATTER
plt.subplot(1, 2, 1)
sb.regplot(data = fuel_econ, x = 'year', y = 'comb', truncate=False);

##########################################
# PLOT ON RIGHT - SCATTER PLOT WITH JITTER
plt.subplot(1, 2, 2)

# In the sb.regplot() function below, the `truncate` argument accepts a boolean.
# If truncate=True, the regression line is bounded by the data limits.
# Else if truncate=False, it extends to the x axis limits.
# The x_jitter will make each x value will be adjusted randomly by +/-0.3
sb.regplot(data = fuel_econ, x = 'year', y = 'comb', truncate=False, x_jitter=0.3);
```



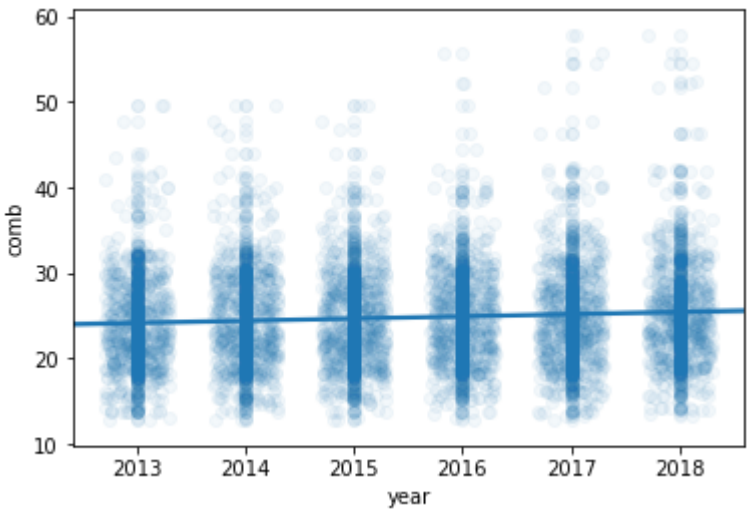
The scatter plot on left showing a simple scatter plot, while the right one presents with jitter.

In the left scatter plot above, the degree of variability in the data and strength of relationship are fairly unclear. In cases like this, we may want to employ *transparency* and *jitter* to make the scatterplot more informative. The right scatter plot has a jitter introduced to the data points.

You can add transparency to either scatter() or regplot() by adding the "alpha" parameter set to a value between 0 (fully transparent, not visible) and 1 (fully opaque). See the example below.

Example 2. Plot with both Jitter and Transparency

```
# The scatter_kws helps specifying the opaqueness of the data points.
# The alpha take a value between [0-1], where 0 represents transparent, and 1 is opaque.
sb.regplot(data = fuel_econ, x = 'year', y = 'comb', truncate=False, x_jitter=0.3, scatter_kws=
{'alpha':1/20});

# Alternative way to plot with the transparency.
# The scatter() function below does NOT have any argument to specify the Jitter
plt.scatter(data = fuel_econ, x = 'year', y = 'comb', alpha=1/20);
```



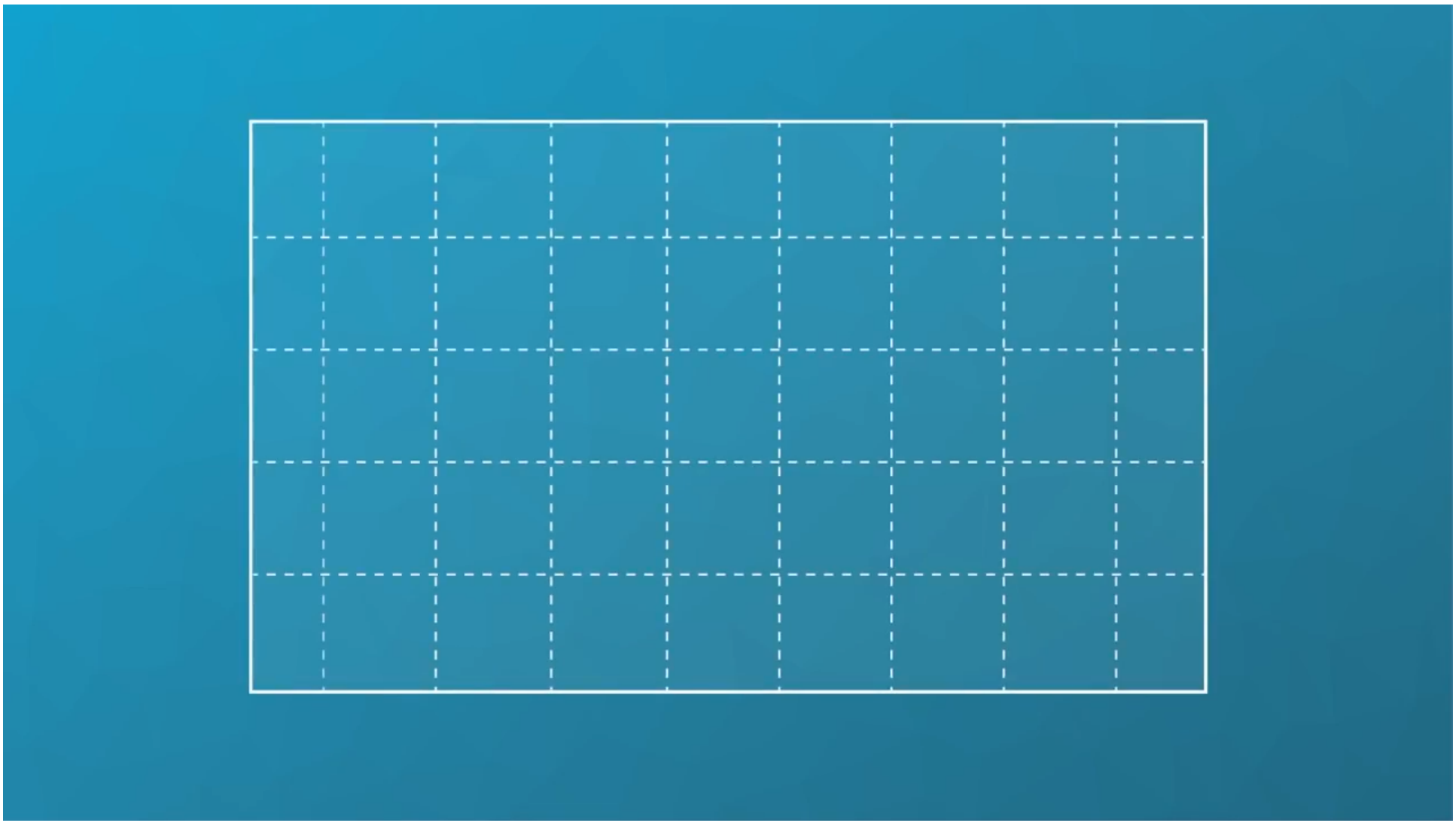
Plot all points with Jitter, and transparency.

In the plot above, the jitter settings will cause each point to be plotted in a uniform ±0.3 range of their true values. Note that transparency has been changed to be a dictionary assigned to the "*scatter_kws*" parameter.

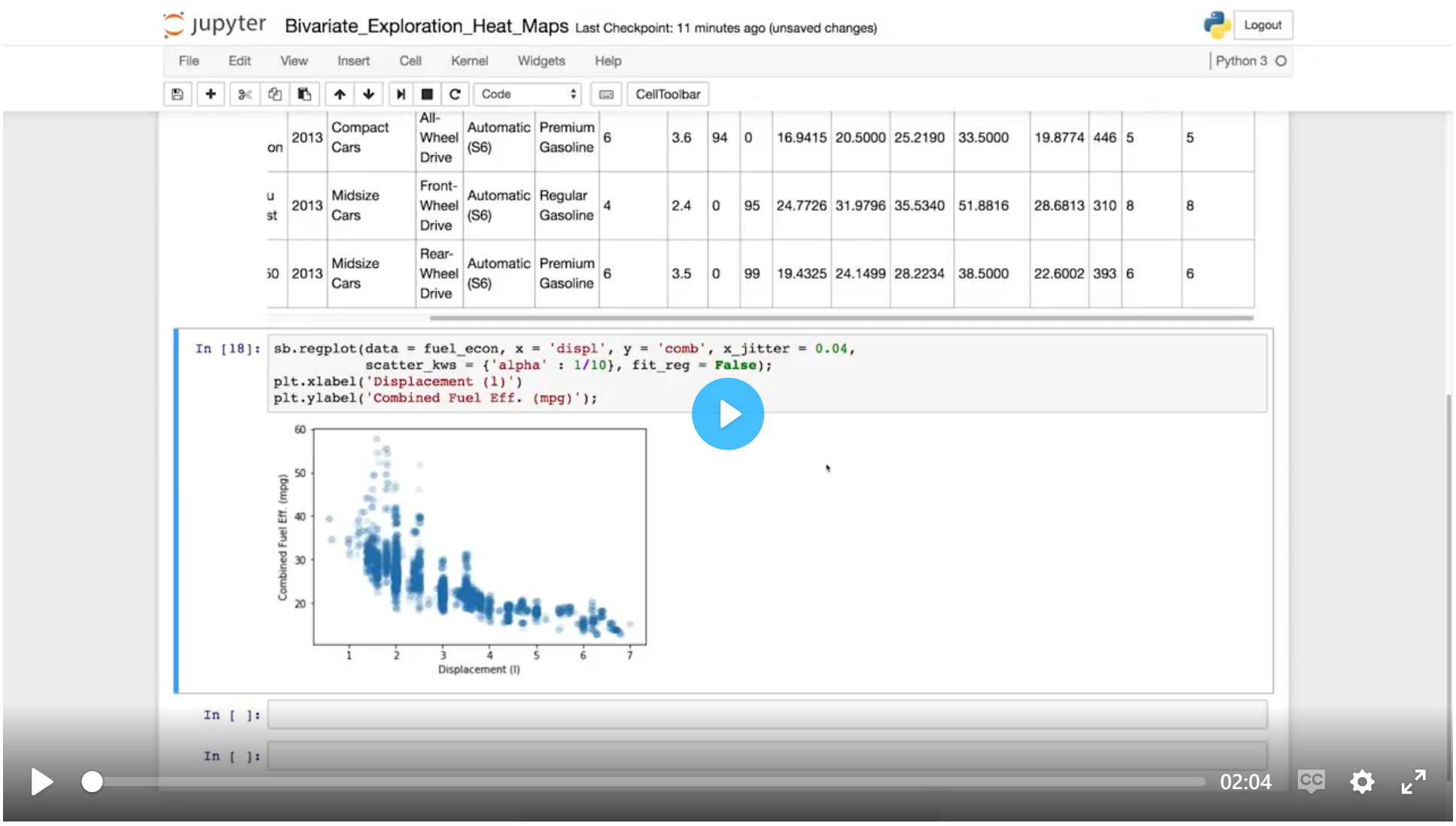
Next Concept

04. Heat Maps

L4 041 Heat Maps V4



Data Vis L4 C04 V1



Heat Maps

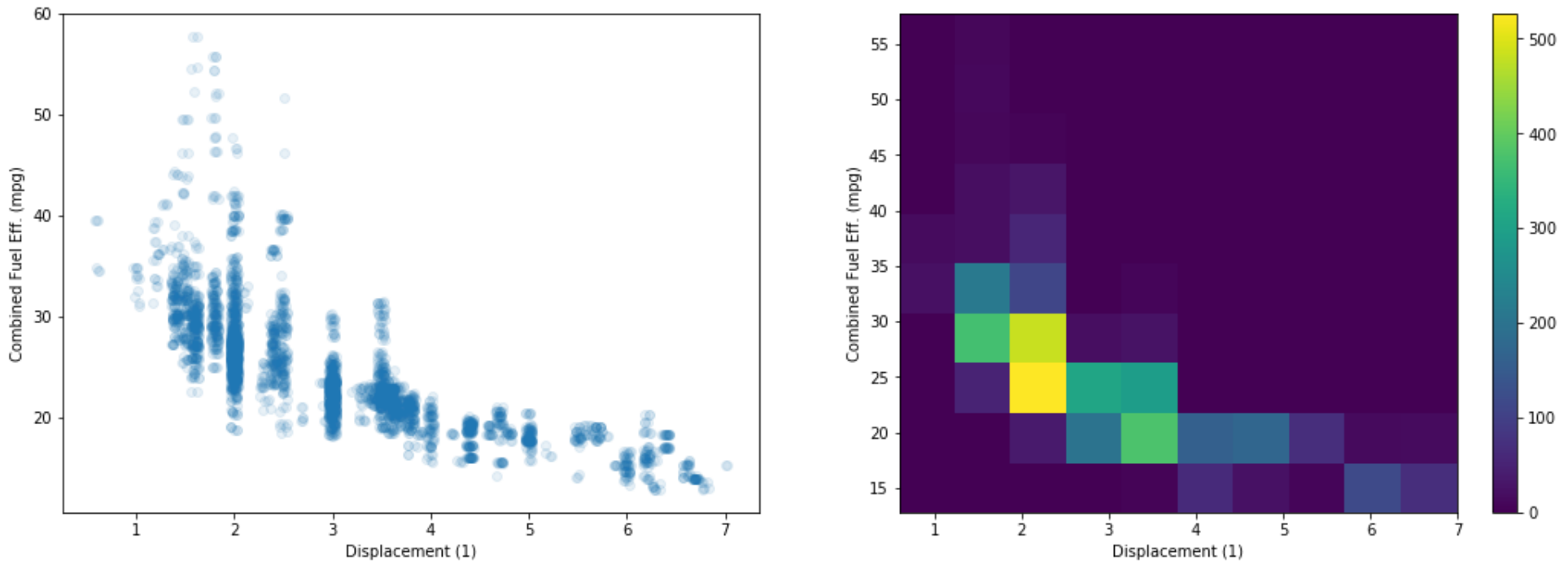
A **heat map** is a 2-d version of the histogram that can be used as an alternative to a scatterplot. Like a scatterplot, the values of the two numeric variables to be plotted are placed on the plot axes. Similar to a histogram, the plotting area is divided into a grid and the number of points in each grid rectangle is added up. Since there won't be room for bar heights, counts are indicated instead by grid cell color. A heat map can be implemented with Matplotlib's hist2d() function.

Heat maps are useful in the following cases:

1. To represent a plot for discrete vs. another discrete variable
2. As an alternative to transparency when the data points are enormous

Example 1. Default heat plot using Matplotlib.pyplot.hist2d() function

```
# TO DO: Necessary import

# Read the CSV file
fuel_econ = pd.read_csv('fuel_econ.csv')
fuel_econ.head(10)


plt.figure(figsize = [18, 6])

# PLOT ON LEFT
plt.subplot(1, 2, 1)
sb.regplot(data = fuel_econ, x = 'displ', y = 'comb', x_jitter=0.04, scatter_kws={'alpha':1/10},
fit_reg=False)
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');

# PLOT ON RIGHT
plt.subplot(1, 2, 2)
plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb')
plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');
```



A scatter plot vs heat plot based on the same data

In the example above, we added a colorbar() function call to add a colorbar to the side of the plot, showing the mapping from counts to colors.

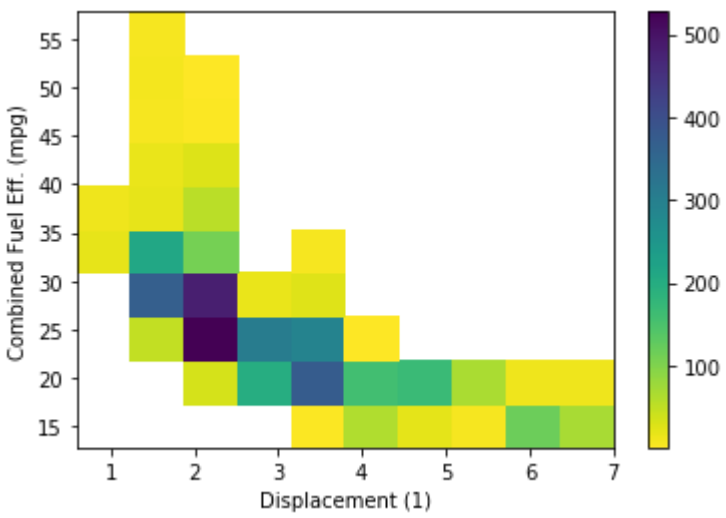
Additional Variations

To select a different color palette, you can set the "cmap" parameter in `hist2d`. The most convenient way of doing this is to set the "cmap" value as a string referencing a built-in Matplotlib palette. A list of valid strings can be found on this part of the Pyplot API documentation. A further discussion of color in plots will be left to the next lesson. For now, I will just show an example of reversing the default "viridis" color palette, by setting `cmap = 'viridis_r'`.

Furthermore, I would like to distinguish cells with zero counts from those with non-zero counts. The "cmin" parameter specifies the minimum value in a cell before it will be plotted. By adding a `cmin = 0.5` parameter to the `hist2d` call, this means that a cell will only get colored if it contains at least one point.

Example 2. Heat plot - Set a minimum bound on counts and a reverse color map

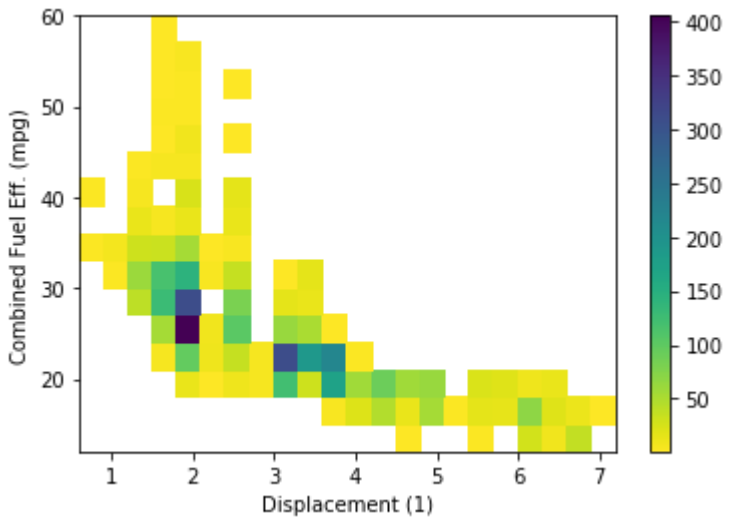
```
# Use cmin to set a minimum bound of counts
# Use cmap to reverse the color map.
plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb', cmin=0.5, cmap='viridis_r')
plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');
```



As the count of points in the cell increases, the color in the heatmap gets brighter and moves from blue to yellow.

Example 3. Heat plot - Specify bin edges

```
# Specify bin edges
bins_x = np.arange(0.6, 7+0.3, 0.3)
bins_y = np.arange(12, 58+3, 3)

plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb', cmin=0.5, cmap='viridis_r', bins = [bins_x, bins_y])
plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');

# Notice the areas of high frequency in the middle of the negative trend in the plot.
```



A heat plot with a custom bin size

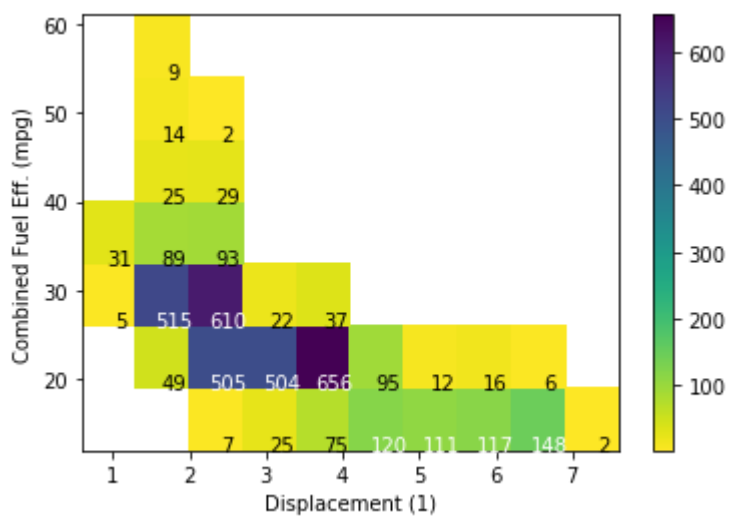
Notice that since we have two variables, the "bins" parameter takes a list of two bin edge specifications, one for each dimension. Choosing an appropriate bin size is just as important here as it was for the univariate histogram.

Annotations on each cell

If you have a lot of data, you might want to add annotations to cells in the plot indicating the count of points in each cell. From `hist2d`, this requires the addition of text elements one by one, much like how text annotations were added one by one to the bar plots in the previous lesson. We can get the counts to annotate directly from what is returned by `hist2d`, which includes not just the plotting object, but an array of counts and two vectors of bin edges.

Example 4. Add text annotation on each cell using pyplot.text() function

```
# Specify bin edges
bins_x = np.arange(0.6, 7+0.7, 0.7)
bins_y = np.arange(12, 58+7, 7)
# Use cmin to set a minimum bound of counts
# Use cmap to reverse the color map.
h2d = plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb', cmin=0.5, cmap='viridis_r', bins = [bins_x,
bins_y])

plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');

# Select the bi-dimensional histogram, a 2D array of samples x and y.
# Values in x are histogrammed along the first dimension and
# values in y are histogrammed along the second dimension.
counts = h2d[0]

# Add text annotation on each cell
# Loop through the cell counts and add text annotations for each
for i in range(counts.shape[0]):
    for j in range(counts.shape[1]):
        c = counts[i,j]
        if c >= 100: # increase visibility on darker cells
            plt.text(bins_x[i]+0.5, bins_y[j]+0.5, int(c),
                     ha = 'center', va = 'center', color = 'white')
        elif c > 0:
            plt.text(bins_x[i]+0.5, bins_y[j]+0.5, int(c),
                     ha = 'center', va = 'center', color = 'black')
```



A heat plot with annotation on each cell

If you have too many cells in your heat map, then the annotations will end up being too overwhelming, too much to attend to. In cases like that, it's best to leave off the annotations and let the data and colorbar speak for themselves.

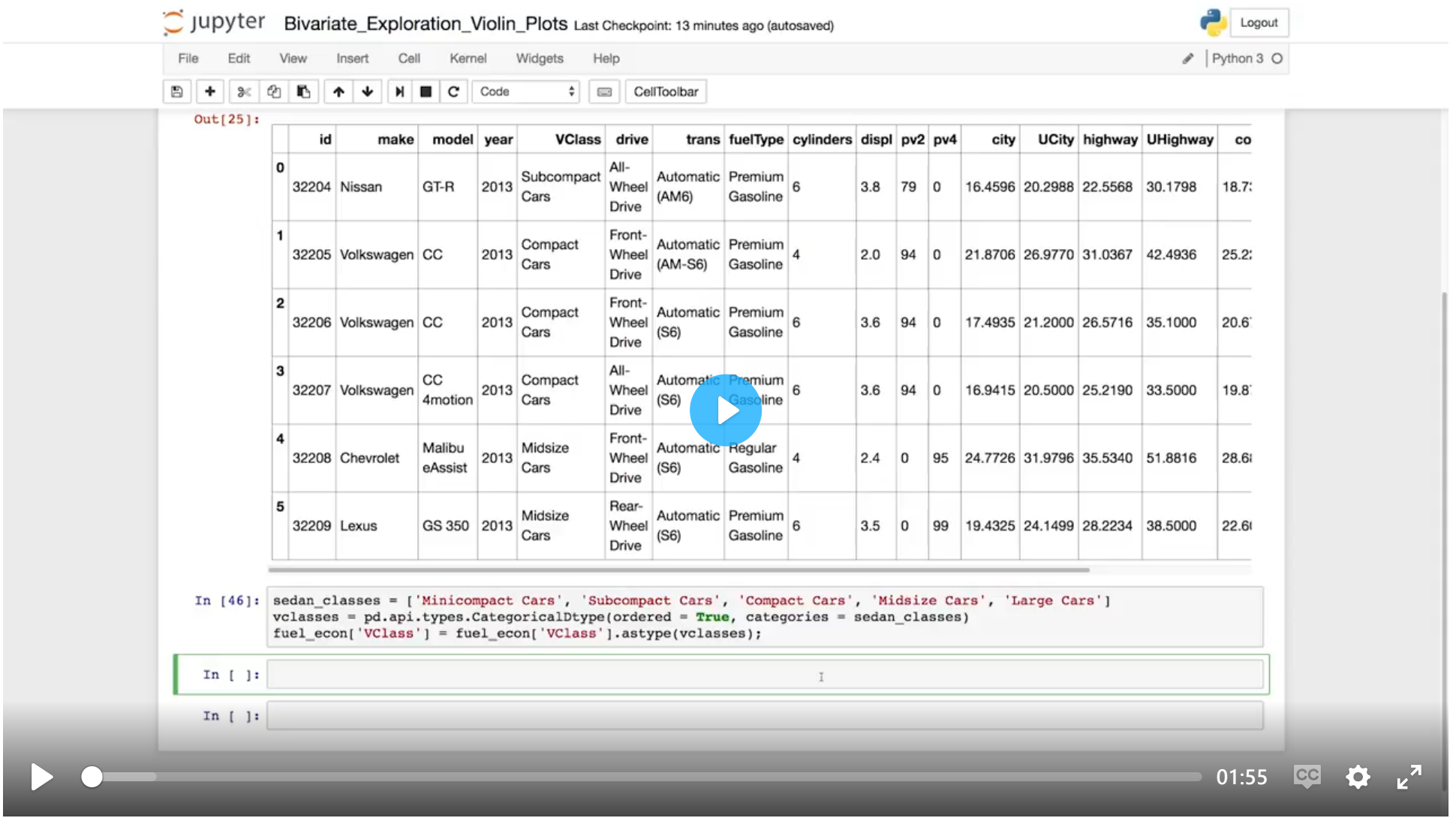
You're more likely to see annotations in a categorical heat map, where there are going to be fewer cells plotted. Indeed, there is a parameter built into seaborn's `heatmap()` function that is built for categorical heatmaps, as will be seen later.

Next Concept

06. Violin Plots

L4 061 Violin Plots 2 V3

Data Vis L4 C06 V2



	id	make	model	year	VClass	drive	trans	fuelType	cylinders	displ	pv2	pv4	city	UCity	highway	UHighway	co
0	32204	Nissan	GT-R	2013	Subcompact Cars	All-Wheel Drive	Automatic (AM6)	Premium Gasoline	6	3.8	79	0	16.4596	20.2988	22.5568	30.1798	18.7
1	32205	Volkswagen	CC	2013	Compact Cars	Front-Wheel Drive	Automatic (AM-S6)	Premium Gasoline	4	2.0	94	0	21.8706	26.9770	31.0367	42.4936	25.2
2	32206	Volkswagen	CC	2013	Compact Cars	Front-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	94	0	17.4935	21.2000	26.5716	35.1000	20.6
3	32207	Volkswagen	CC 4motion	2013	Compact Cars	All-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	94	0	16.9415	20.5000	25.2190	33.5000	19.8
4	32208	Chevrolet	Malibu eAssist	2013	Midsize Cars	Front-Wheel Drive	Automatic (S6)	Regular Gasoline	4	2.4	0	95	24.7726	31.9796	35.5340	51.8816	28.6
5	32209	Lexus	GS 350	2013	Midsize Cars	Rear-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.5	0	99	19.4325	24.1499	28.2234	38.5000	22.6

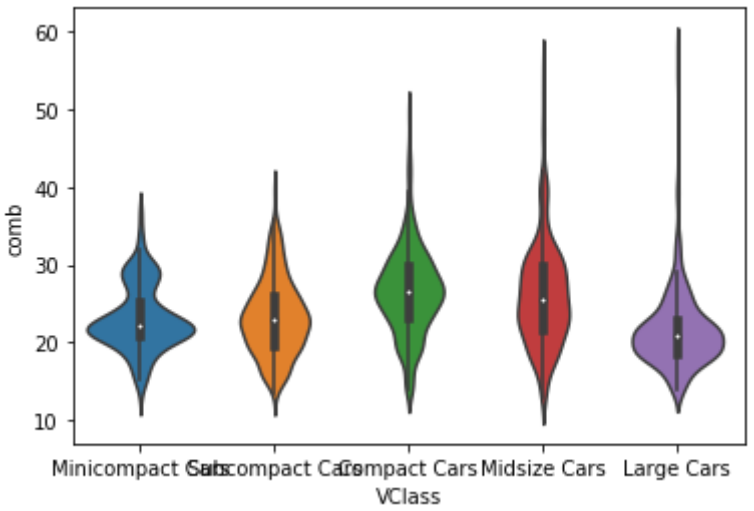
Violin Plots

There are a few ways of plotting the relationship between one quantitative and one qualitative variable, that demonstrate the data at different levels of abstraction. The **violin plot** is on the lower level of abstraction. For each level of the categorical variable, a distribution of the values on the numeric variable is plotted. The distribution is plotted as a kernel density estimate, something like a smoothed histogram. There is an extra section at the end of the previous lesson that provides more insight into kernel density estimates.

Seaborn's violinplot() function can be used to create violin plots.

Example 1. Violin plot for plotting a Quantitative variable (fuel efficiency) versus Qualitative variable (vehicle class)

```
# Types of sedan cars
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize Cars', 'Large Cars']

# Returns the types for sedan_classes with the categories and orderedness
# Refer - https://pandas.pydata.org/pandas-
docs/version/0.23.4/generated/pandas.api.types.CategoricalDtype.html
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)

# Use pandas.astype() to convert the "VClass" column from a plain object type into an ordered categorical
type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);

sb.violinplot(data=fuel_econ, x='VClass', y='comb');
```



A violin plot with default arguments in violinplot() function

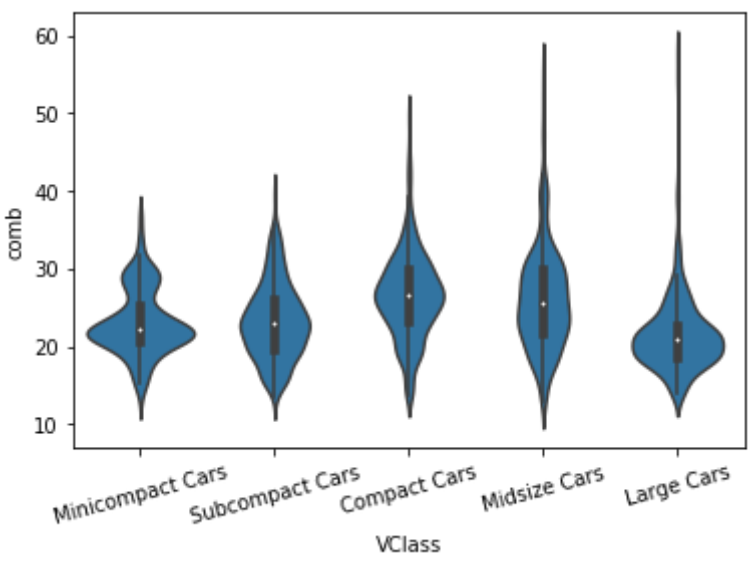
---

You can see that each level has been rendered in a different color, like how the plain `countplot()` was in the previous lesson. We can set the "color" parameter to make each curve the same color if it is not meaningful.

Inside each curve, there is a black shape with a white dot inside, a miniature *box plot*. A further discussion of box plots will be performed on the next page. If you'd like to remove the box plot, you can set the `inner = None` parameter in the `violinplot` call to simplify the look of the final visualization.

Example 2. Violin plot without datapoints in the violin interior

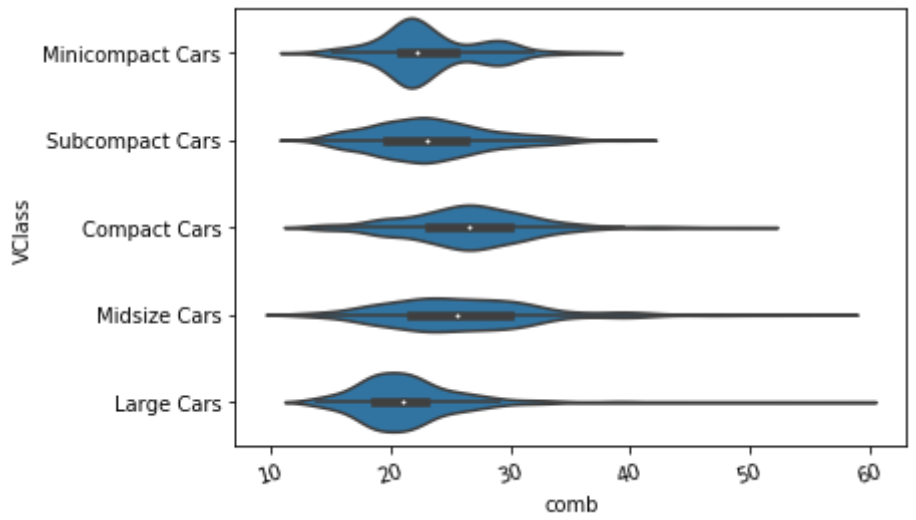
```
base_color = sb.color_palette()[0]

# The "inner" argument represents the datapoints in the violin interior.
# It can take any value from {“box”, “quartile”, “point”, “stick”, None}
# If "box", it draws a miniature boxplot.
sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color, innner=None)
plt.xticks(rotation=15);
```



Additional Variation

Much like how the bar chart could be rendered with horizontal bars, the violin plot can also be rendered horizontally. Seaborn is smart enough to make an appropriate inference on which orientation is requested, depending on whether "x" or "y" receives the categorical variable. But if both variables are numeric (e.g., one is discretely-valued) then the "orient" parameter can be used to specify the plot orientation.

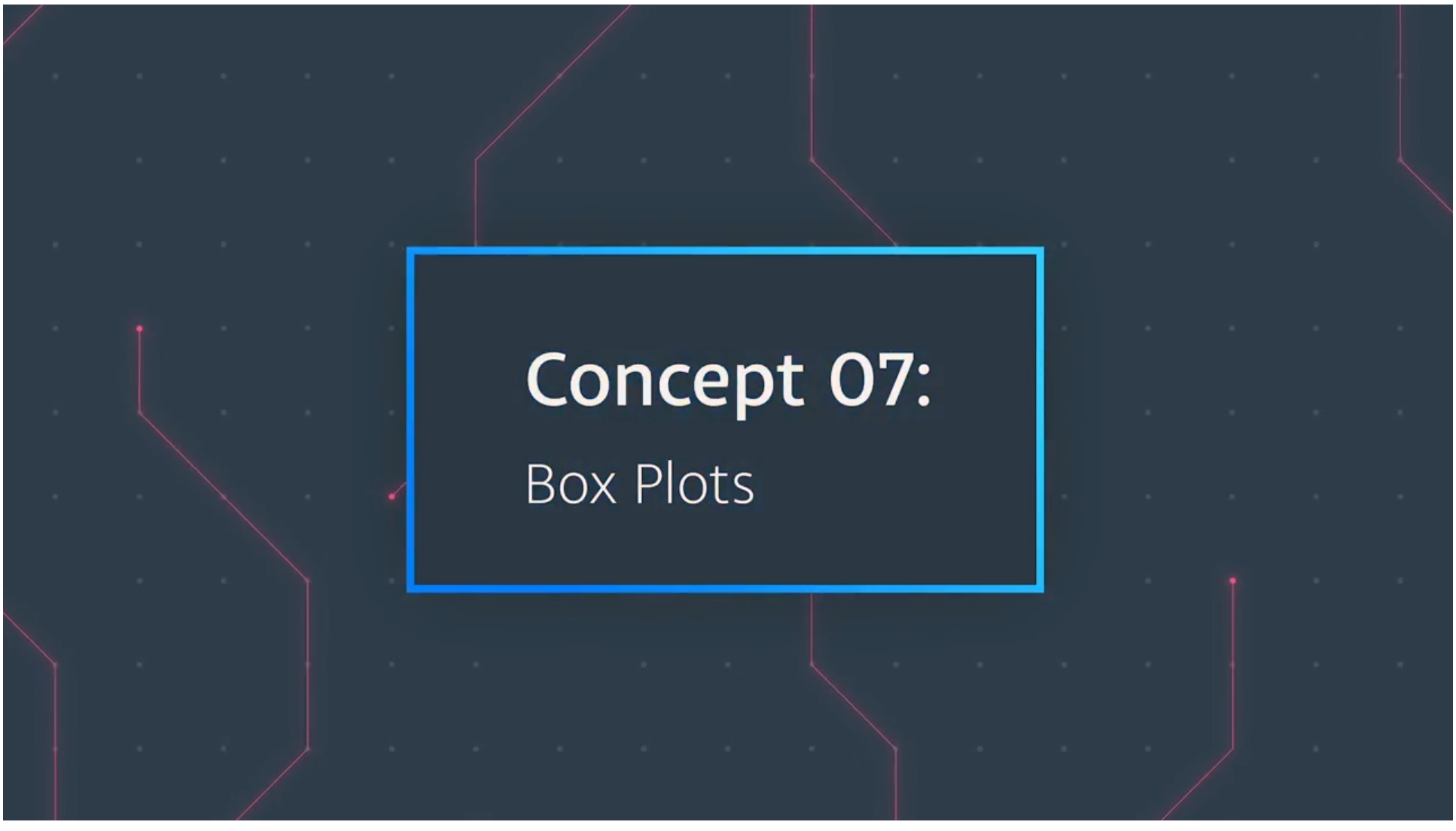
```
sb.violinplot(data=fuel_econ, y='VClass', x='comb', color=base_color, innner=None);
```



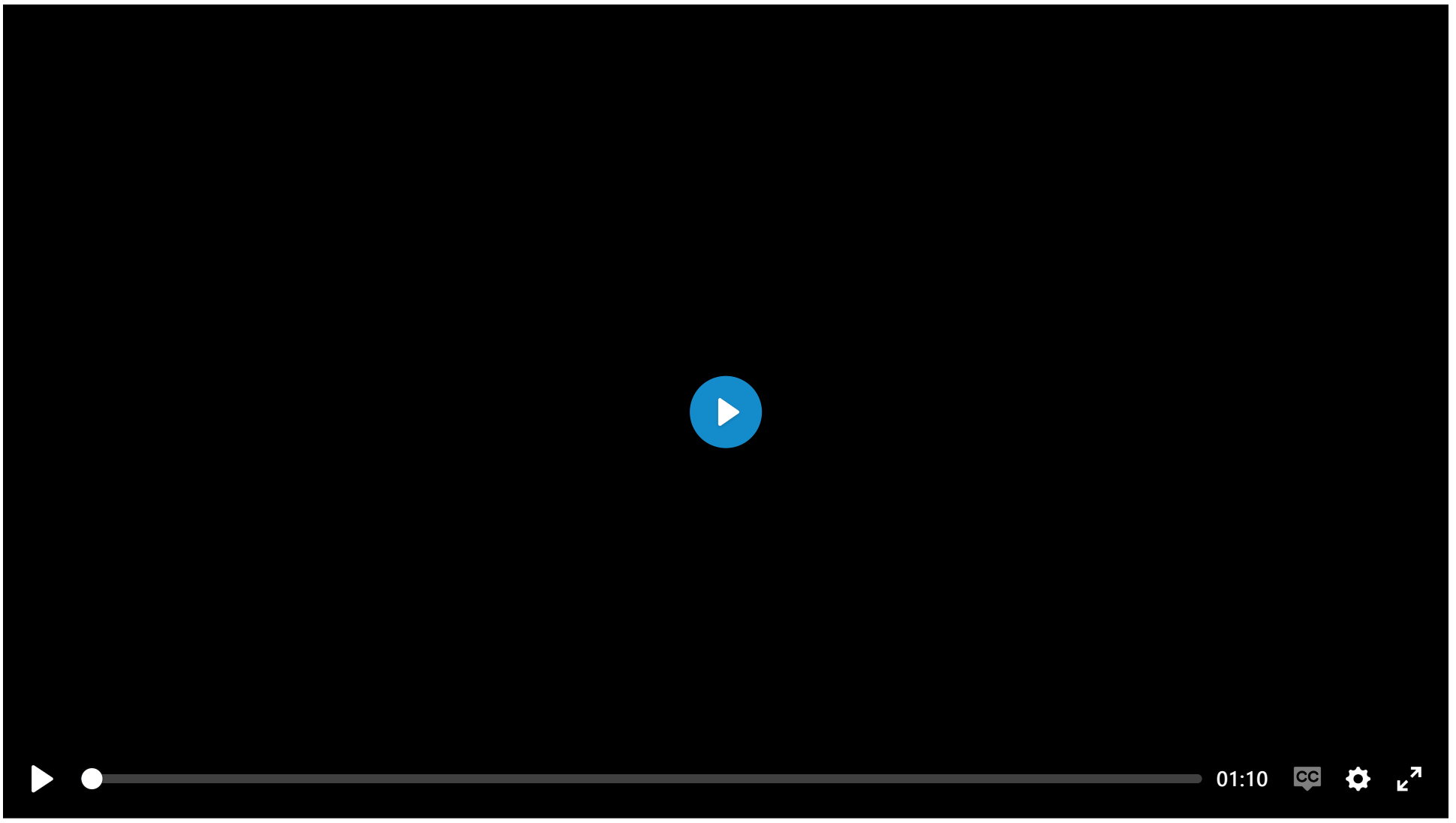
Next Concept

07. Box Plots

L4 071 Box Plots V4



Data Vis L4 C07 V1



Box Plots

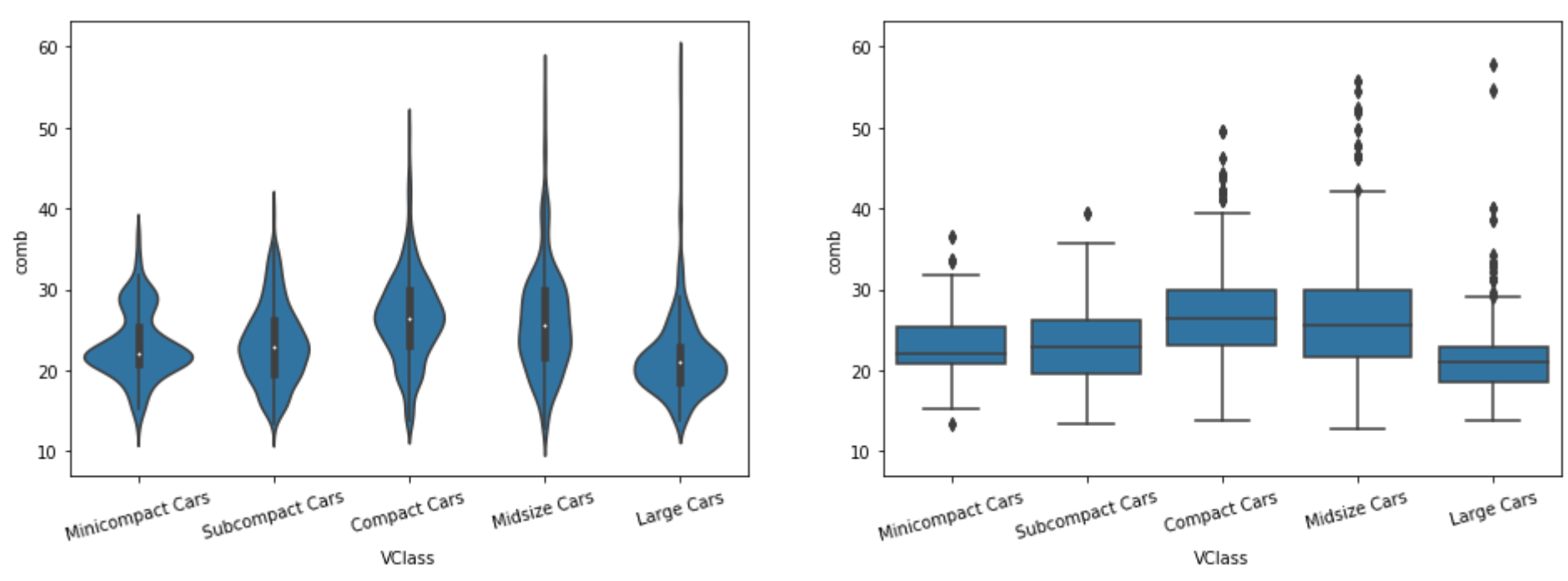
A **box plot** is another way of showing the relationship between a numeric variable and a categorical variable. Compared to the violin plot, the box plot leans more on the summarization of the data, primarily just reporting a set of descriptive statistics for the numeric values on each categorical level. A box plot can be created using seaborn's boxplot() function.

Example 1. Violin versus Box plot

```
# Step 1. Import packages

# Step 2. Load data

# Step 3. Convert the "VClass" column from a plain object type into an ordered categorical type
# Types of sedan cars
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize Cars', 'Large Cars']

# Returns the types for sedan_classes with the categories and orderedness
# Refer - https://pandas.pydata.org/pandas-
docs/version/0.23.4/generated/pandas.api.types.CategoricalDtype.html
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)

# Use pandas.astype() to convert the "VClass" column from a plain object type into an ordered categorical
type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);

# Step 4.  TWO PLOTS IN ONE FIGURE
plt.figure(figsize = [16, 5])
base_color = sb.color_palette()[0]

# LEFT plot: violin plot
plt.subplot(1, 2, 1)
#Let's return the axes object
ax1 = sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color, innner='quartile')
plt.xticks(rotation=15);

# RIGHT plot: box plot
plt.subplot(1, 2, 2)
sb.boxplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=15);
plt.ylim(ax1.get_ylim()) # set y-axis limits to be same as left plot
```

Note that the "color" parameter is being used here to make each box the same color. In order to provide a better comparison of the violin and box plots, a `ylim` expression has been added to the second plot to match the two plots' y-axis limits. The Axes object returned by `violinplot()` is assigned to a variable, `ax1` is used to programmatically obtain those limit values.

Documentation: Axes objects



Comparison between violin and box plot

The inner boxes and lines in the violin plot match up with the boxes and whiskers in the box plot. In a box plot, the central line in the box indicates the median of the distribution, while the top and bottom of the box represent the third and first quartiles of the data, respectively. Thus, the height of the box is the interquartile range (IQR). From the top and bottom of the box, the whiskers indicate the range from the first or third quartiles to the minimum or maximum value in the distribution. Typically, a maximum range is set on whisker length; by default, this is 1.5 times the IQR. For the Gamma level, there are points below the lower whisker that indicate individual outlier points that are more than 1.5 times the IQR below the first quartile.

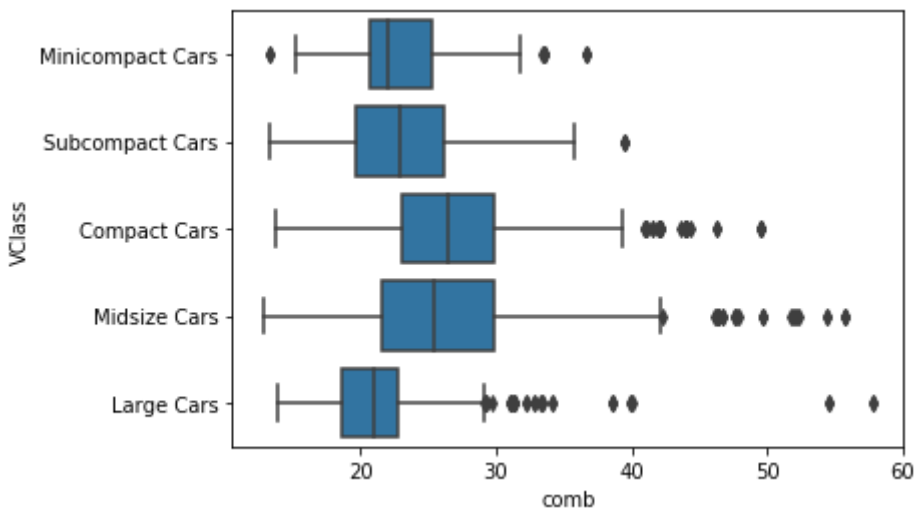
Comparing the two plots, the box plot is a cleaner summary of the data than the violin plot. It's easier to compare statistics between the groups with a box plot. This makes a box plot worth more consideration if you have a lot of groups to compare, or if you are building explanatory plots. You can clearly see from the box plot that the Delta group has the lowest median. On the other hand, the box plot lacks as nuanced a depiction of distributions as the violin plot: you can't see the slight bimodality present in the Alpha level values. The violin plot may be a better option for exploration, especially since seaborn's implementation also includes the box plot by default.

Additional Variations

As with `violinplot`, `boxplot` can also render horizontal box plots by setting the numeric and categorical features to the appropriate arguments.

Example 2. Horizontal box plot

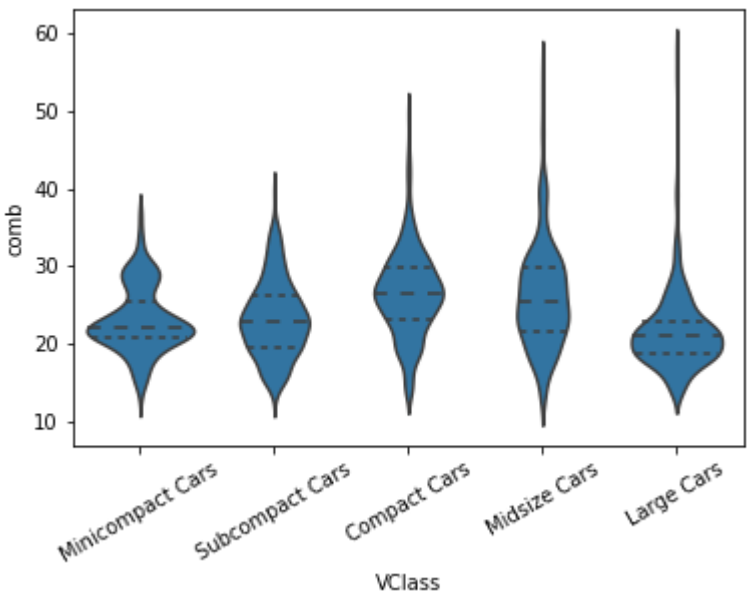
```
sb.boxplot(data=fuel_econ, y='VClass', x='comb', color=base_color)
```



In `violinplot`, there is an additional option for plotting summary statistics in the violin, beyond the default mini box plot. By setting `inner = 'quartile'`, three lines will be plotted within each violin area for the three middle quartiles. The line with thick dashes indicates the median, and the two lines with shorter dashes on either side the first and third quartiles.

Example 3. Violin plot with quartile information in the middle

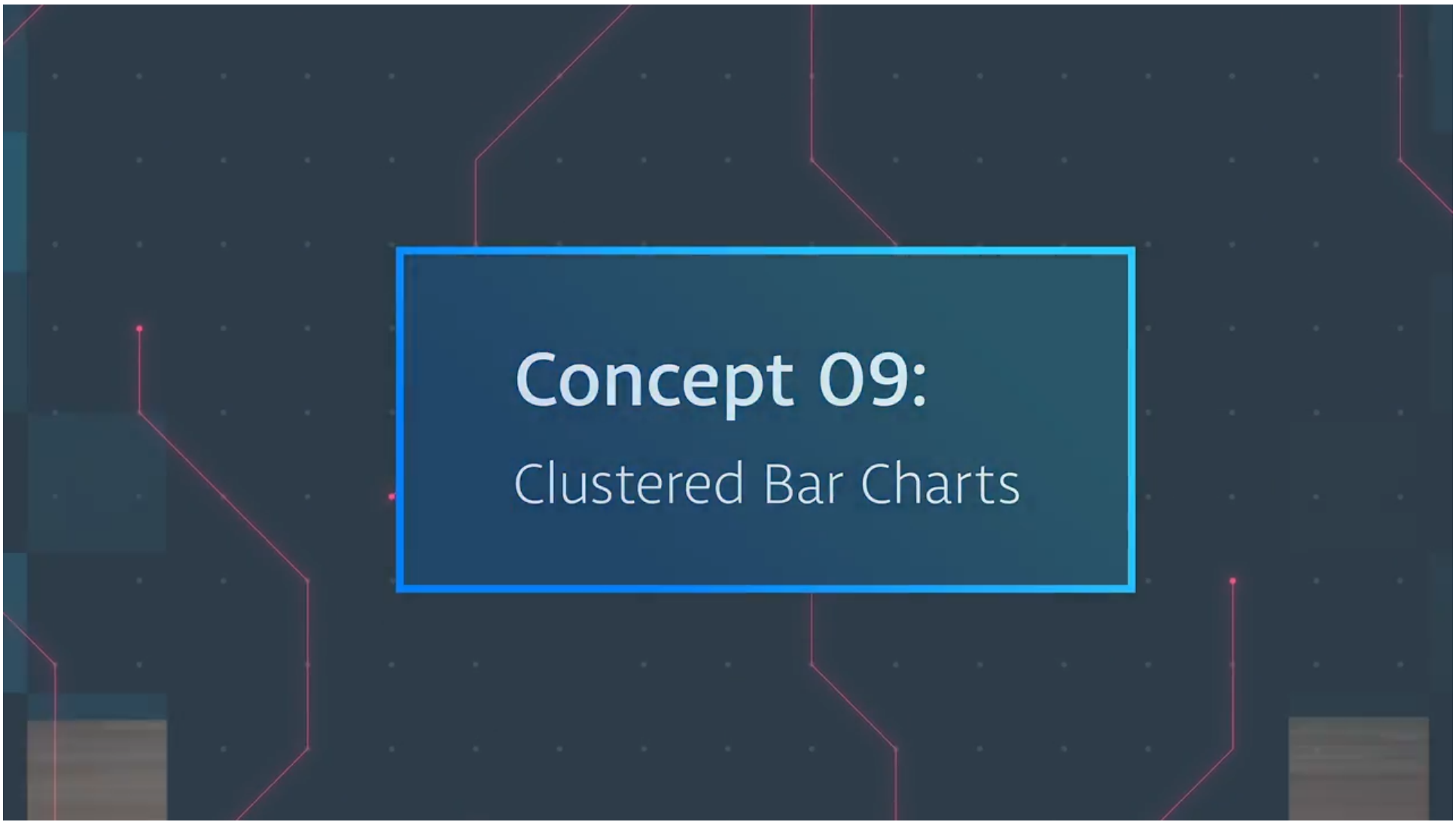
```
base_color = sb.color_palette()[0]
sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color, inner='quartile')
plt.xticks(rotation=30);
```



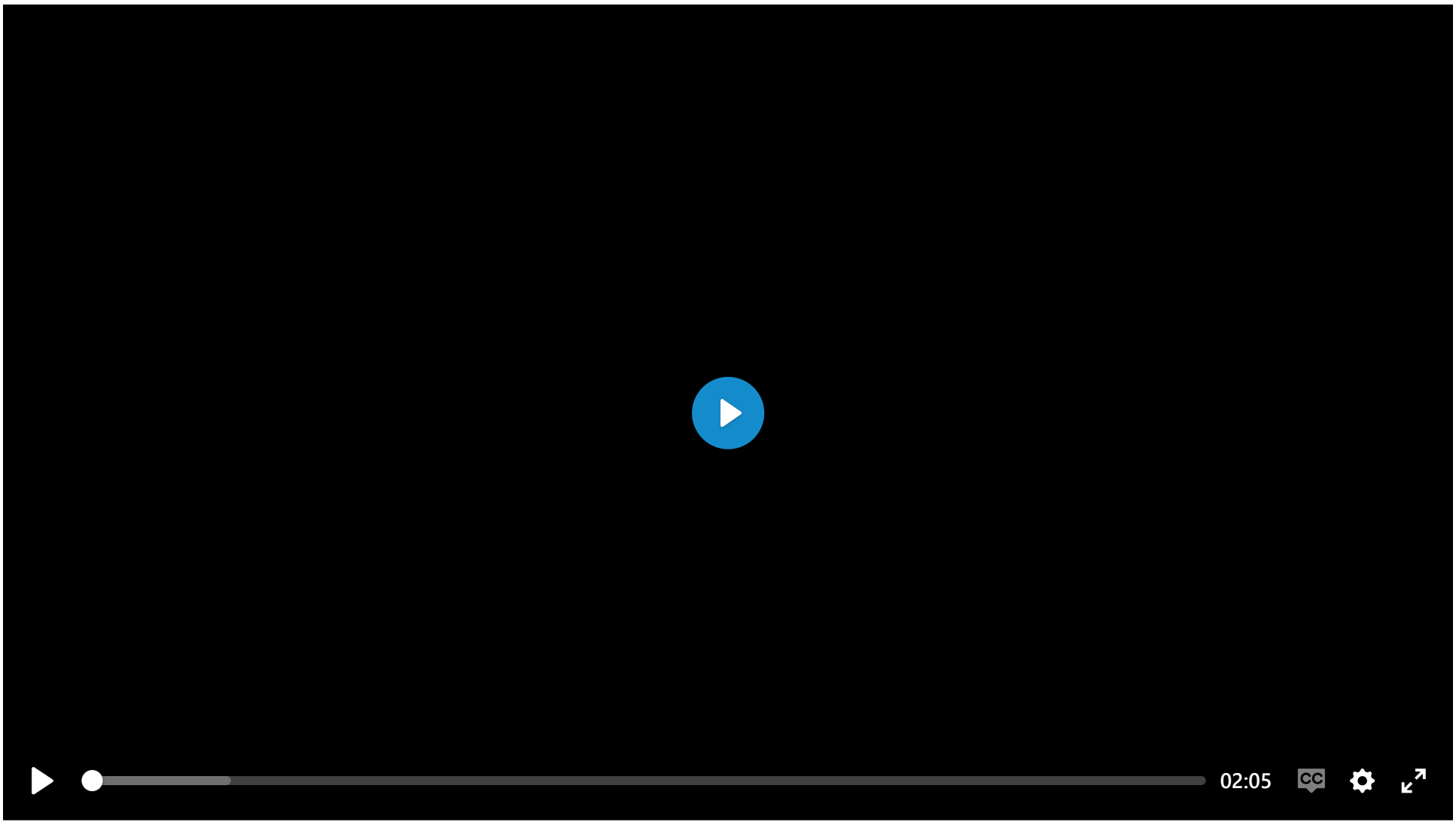
Next Concept

09. Clustered Bar Charts

L4 091 Clustered Bar Charts V4



Data Vis L4 C09 V1



Clustered Bar Charts

To depict the relationship between two categorical variables, we can extend the univariate bar chart seen in the previous lesson into a **clustered bar chart**. Like a standard bar chart, we still want to depict the count of data points in each group, but each group is now a combination of labels on two variables. So we want to organize the bars into an order that makes the plot easy to interpret. In a clustered bar chart, bars are organized into clusters based on levels of the first variable, and then bars are ordered consistently across the second variable within each cluster. This is easiest to see with an example, using seaborn's `countplot` function. To take the plot from univariate to bivariate, we add the second variable to be plotted under the "hue" argument:

Example 1. Plot a Bar chart between two qualitative variables

Preparatory Step 1 - Convert the "VClass" column from a plain object type into an ordered categorical type

```
# Types of sedan cars
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize Cars', 'Large Cars']

# Returns the types for sedan_classes with the categories and orderedness
# Refer - https://pandas.pydata.org/pandas-
docs/version/0.23.4/generated/pandas.api.types.CategoricalDtype.html
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)

# Use pandas.astype() to convert the "VClass" column from a plain object type into an ordered categorical
type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);
```

Preparatory Step 2 - Add a new column for transmission type - Automatic or Manual

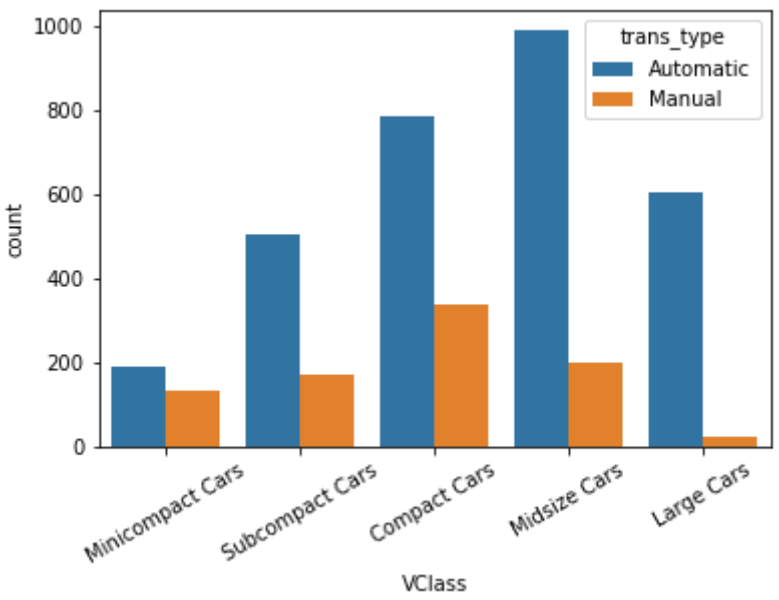
```
# The existing `trans` column has multiple sub-types of Automatic and Manual.
# But, we need plain two types, either Automatic or Manual. Therefore, add a new column.

# The Series.apply() method invokes the `lambda` function on each value of `trans` column.
# In python, a `lambda` function is an anonymous function that can have only one expression.
fuel_econ['trans_type'] = fuel_econ['trans'].apply(lambda x:x.split()[0])
fuel_econ.head()
```

year	VClass	drive	trans	fuelType	cylinders	displ	...	pv4	city	UCity	highway	UHighway	comb	co2	feScore	ghgScore	trans_type
2013	Subcompact Cars	All-Wheel Drive	Automatic (AM6)	Premium Gasoline	6	3.8	...	0	16.4596	20.2988	22.5568	30.1798	18.7389	471	4	4	Automatic
2013	Compact Cars	Front-Wheel Drive	Automatic (AM-S6)	Premium Gasoline	4	2.0	...	0	21.8706	26.9770	31.0367	42.4936	25.2227	349	6	6	Automatic
2013	Compact Cars	Front-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	...	0	17.4935	21.2000	26.5716	35.1000	20.6716	429	5	5	Automatic
2013	Compact Cars	All-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	...	0	16.9415	20.5000	25.2190	33.5000	19.8774	446	5	5	Automatic
2013	Midsize Cars	Front-Wheel Drive	Automatic (S6)	Regular Gasoline	4	2.4	...	95	24.7726	31.9796	35.5340	51.8816	28.6813	310	8	8	Automatic

DataFrame after adding a new column `trans_type`

Step 3. Plot the bar chart

```
sb.countplot(data = fuel_econ, x = 'VClass', hue = 'trans_type')
```



Bar chart between two qualitative variables, and one of them is ordered.

Alternative Approach

Example 2. Plot a Heat Map between two qualitative variables

One alternative way of depicting the relationship between two categorical variables is through a heat map. Heat maps were introduced earlier as the 2-D version of a histogram; here, we're using them as the 2-D version of a bar chart. The seaborn function heatmap() is at home with this type of heat map implementation, but the input arguments are unlike most of the visualization functions that have been introduced in this course. Instead of providing the original dataframe, we need to summarize the counts into a matrix that will then be plotted.

Step 1 - Get the data into desirable format - a DataFrame

```
# Use group_by() and size() to get the number of cars and each combination of the two variable levels as a
pandas Series
ct_counts = fuel_econ.groupby(['VClass', 'trans_type']).size()
```

```
VClass              trans_type
Minicompact Cars    Automatic      188
                    Manual         133
Subcompact Cars     Automatic      502
                    Manual         171
Compact Cars        Automatic      784
                    Manual         338
Midsize Cars        Automatic      989
                    Manual         199
Large Cars          Automatic      605
                    Manual          20
dtype: int64
```

Number of cars in each vehicle type and transmission combination

```
# Use Series.reset_index() to convert a series into a dataframe object
ct_counts = ct_counts.reset_index(name='count')
```

	VClass	trans_type	count
0	Minicompact Cars	Automatic	188
1	Minicompact Cars	Manual	133
2	Subcompact Cars	Automatic	502
3	Subcompact Cars	Manual	171
4	Compact Cars	Automatic	784
5	Compact Cars	Manual	338
6	Midsize Cars	Automatic	989
7	Midsize Cars	Manual	199
8	Large Cars	Automatic	605
9	Large Cars	Manual	20

A DataFrame object created from the Series generated in the step above

trans_type VClass	Automatic	Manual
Minicompact Cars	188	133
Subcompact Cars	502	171
Compact Cars	784	338
Midsize Cars	989	199
Large Cars	605	20

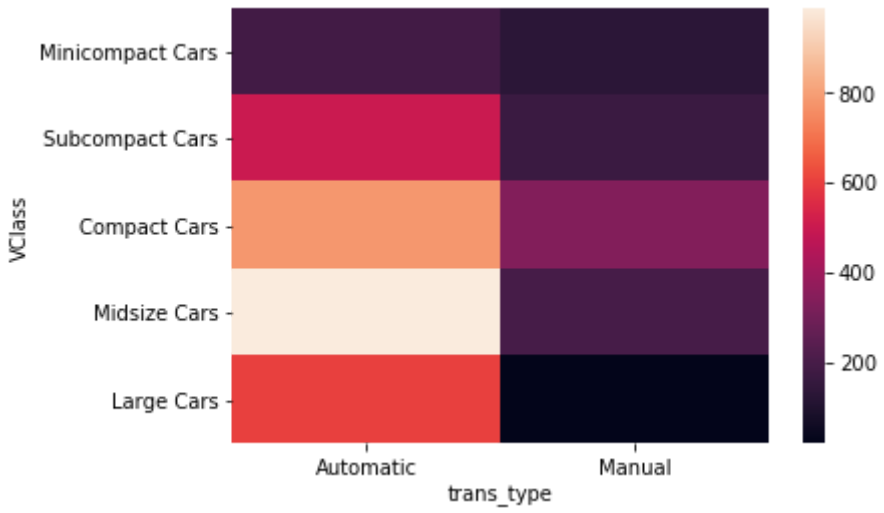
The DataFrame to plot on heatmap

```
# Use DataFrame.pivot() to rearrange the data, to have vehicle class on rows
ct_counts = ct_counts.pivot(index = 'VClass', columns = 'trans_type', values = 'count')
```

Documentation: Series `reset_index`, DataFrame `pivot`

Step 2 - Plot the heatmap

```
sb.heatmap(ct_counts)
```

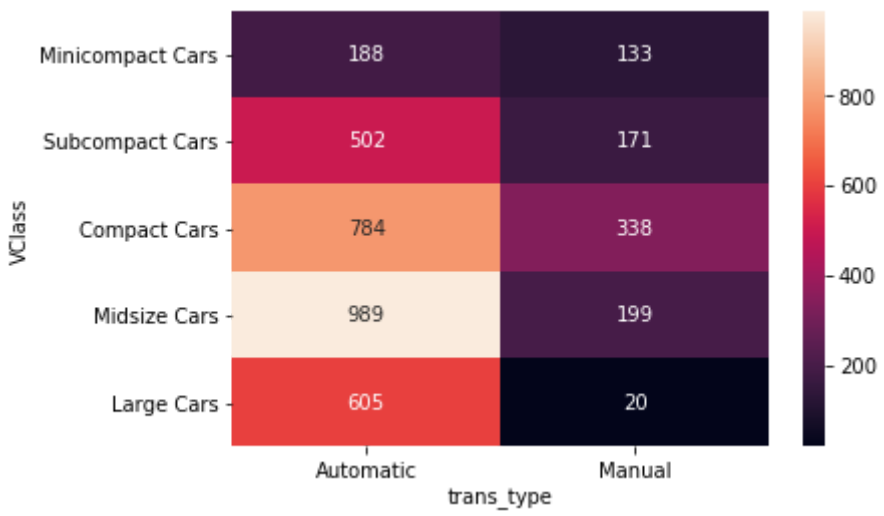


The heat map tells the same story as the clustered bar chart.

Example 3. Additional Variation

```
sb.heatmap(ct_counts, annot = True, fmt = 'd')
```

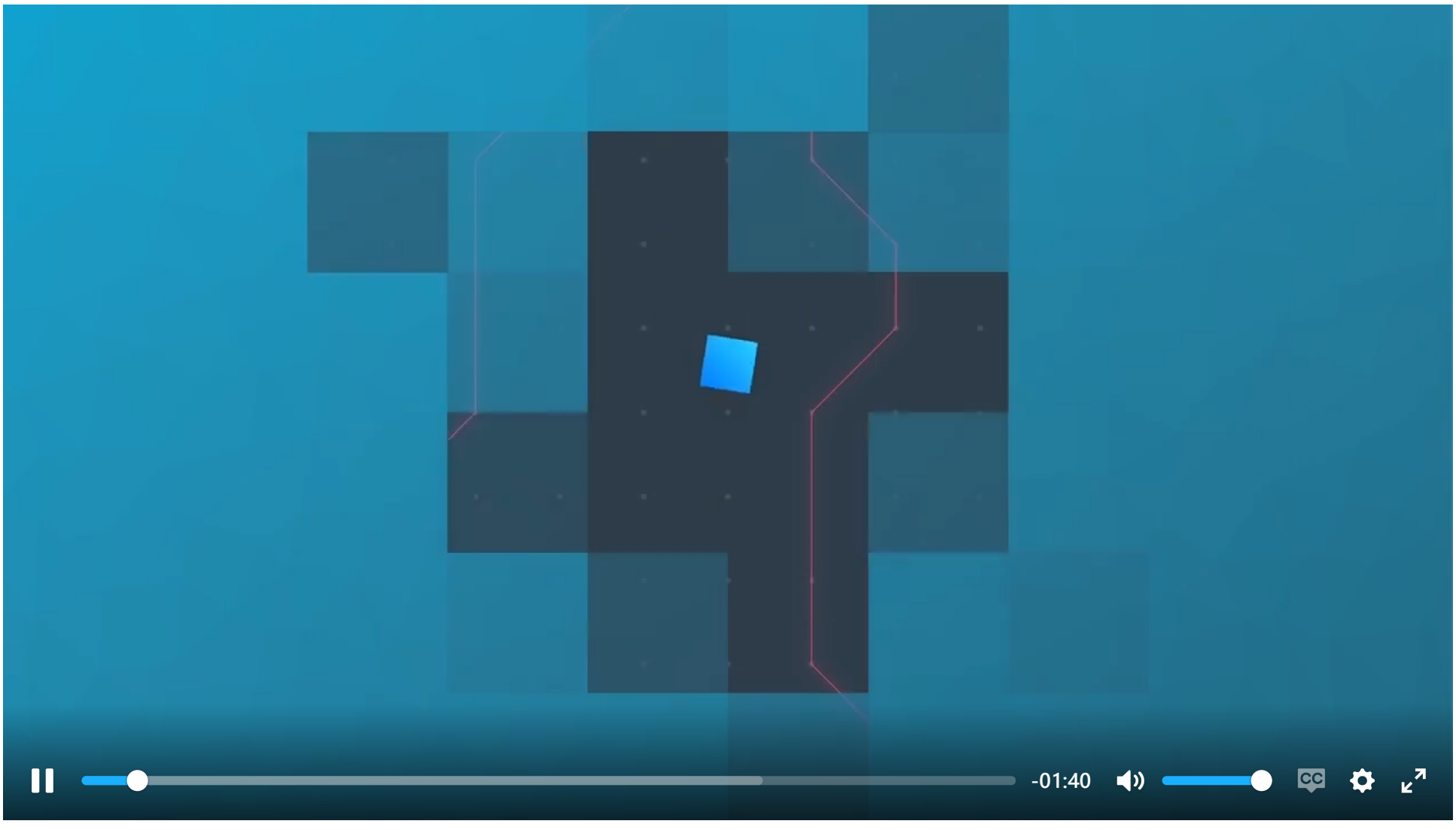
`annot = True` makes it so annotations show up in each cell, but the default string formatting only goes to two digits of precision. Adding `fmt = 'd'` means that annotations will all be formatted as integers instead. You can use `fmt = '.0f'` if you have any cells with no counts, in order to account for NaNs.



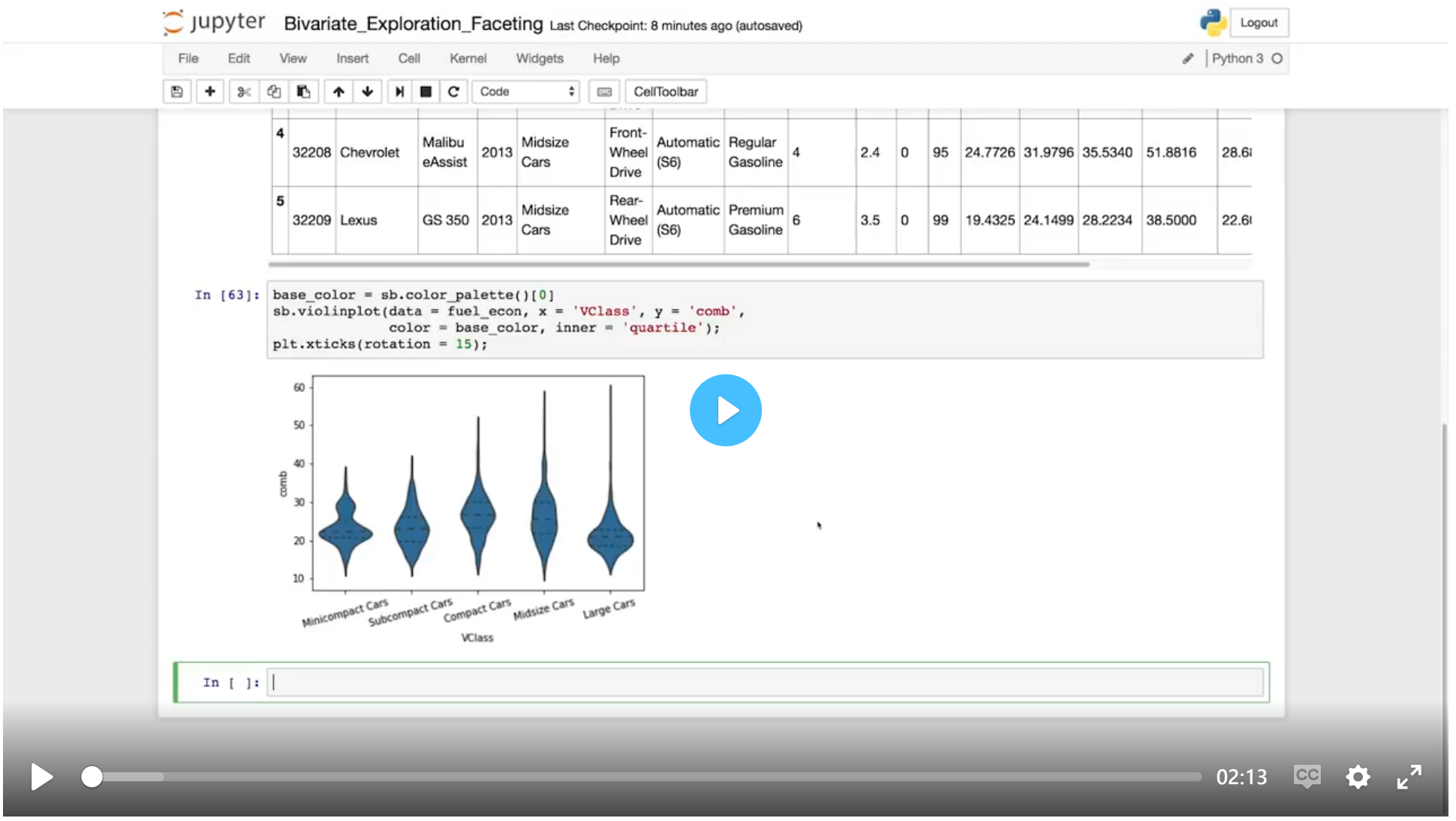
Next Concept

11. Faceting

L4 111 Faceting V2



Data Vis L4 C11 V1



Faceting

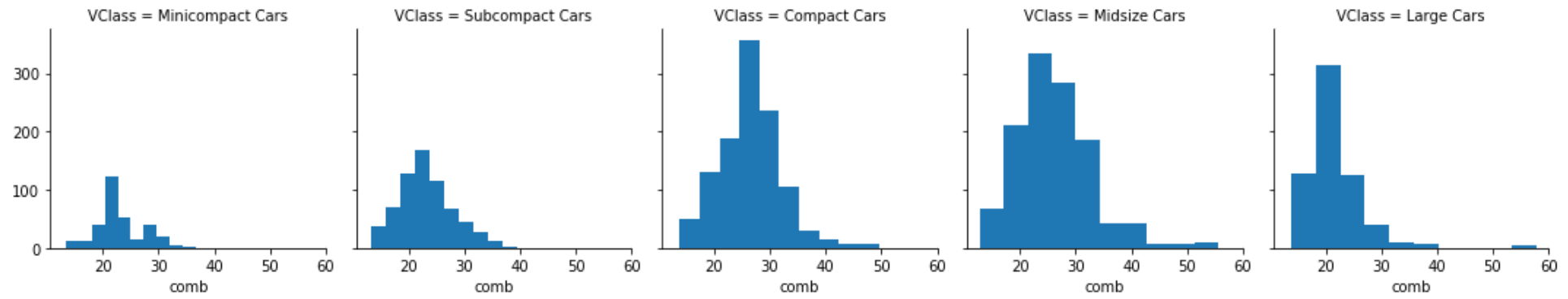
One general visualization technique that will be useful for you to know about to handle plots of two or more variables is **faceting**. In faceting, the data is divided into disjoint subsets, most often by different levels of a categorical variable. For each of these subsets of the data, the same plot type is rendered on other variables. Faceting is a way of comparing distributions or relationships across levels of additional variables, especially when there are three or more variables of interest overall. While faceting is most useful in multivariate visualization, it is still valuable to introduce the technique here in our discussion of bivariate plots.

For example, rather than depicting the relationship between one numeric variable and one categorical variable using a violin plot or box plot, we could use faceting to look at a histogram of the numeric variable for subsets of the data divided by categorical variable levels. Seaborn's FacetGrid class facilitates the creation of faceted plots. There are two steps involved in creating a faceted plot. First, we need to create an instance of the FacetGrid object and specify the feature we want to facet by (*vehicle class, "VClass" in our example*). Then we use the `map` method on the FacetGrid object to specify the plot type and variable(s) that will be plotted in each subset (*in this case, the histogram on combined fuel efficiency "comb"*).

Example 1.

```
# Preparatory Step
fuel_econ = pd.read_csv('fuel_econ.csv')

# Convert the "VClass" column from a plain object type into an ordered categorical type
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize Cars', 'Large Cars']
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);

# Plot the Seaborn's FacetGrid
g = sb.FacetGrid(data = fuel_econ, col = 'VClass')
g.map(plt.hist, "comb")
```

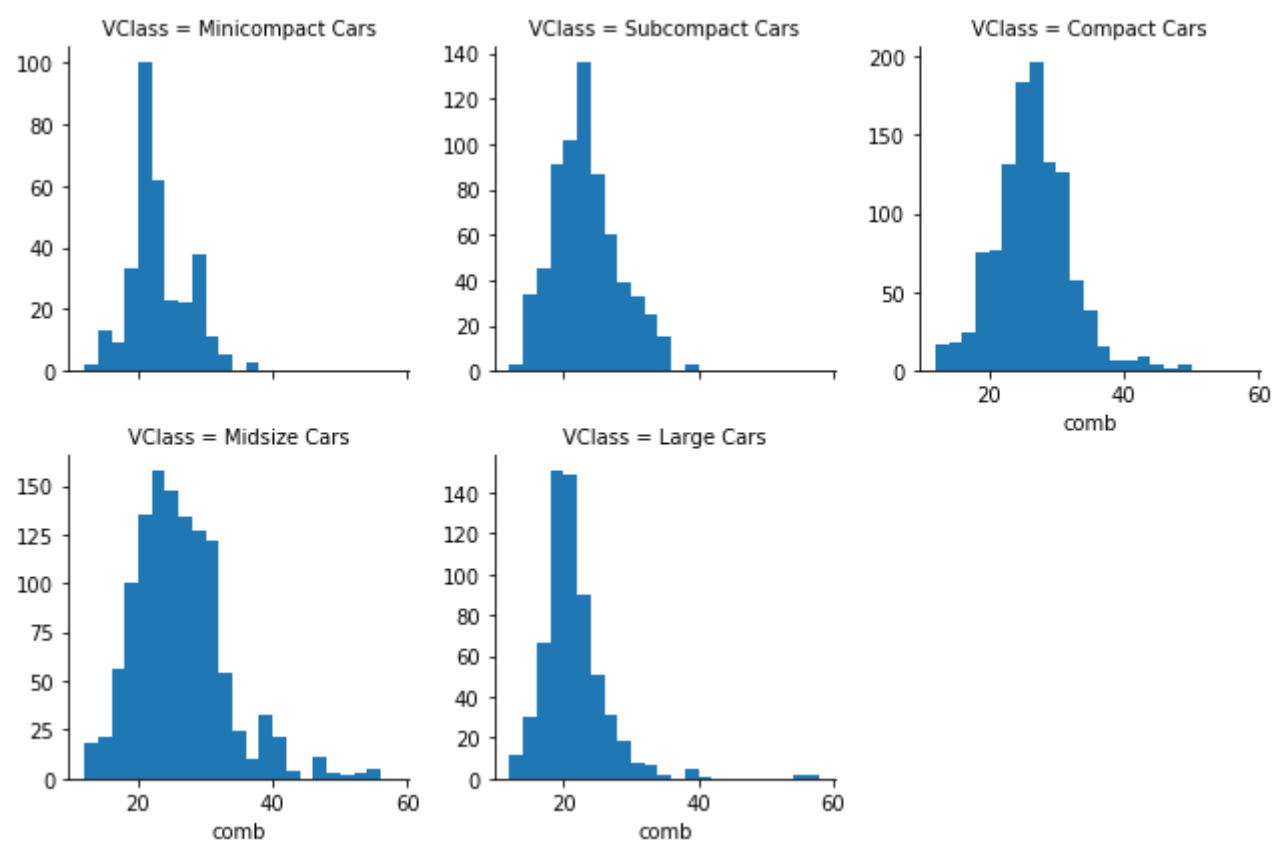
In the `map` call, just set the plotting function and variable to be plotted as positional arguments. Don't set them as keyword arguments, like `x = "comb"`, or the mapping won't work properly.



Notice that each subset of the data is being plotted independently. Each uses the default of ten bins from `hist` to bin together the data, and each plot has a different bin size. Despite that, the axis limits on each facet are the same to allow clear and direct comparisons between groups. It's still worth cleaning things a little bit more by setting the same bin edges on all facets. Extra visualization parameters can be set as additional keyword arguments to the `map` function.

Example 2.

```
bin_edges = np.arange(12, 58+2, 2)

# Try experimenting with dynamic bin edges
# bin_edges = np.arange(-3, fuel_econ['comb'].max()+1/3, 1/3)

g = sb.FacetGrid(data = fuel_econ, col = 'VClass', col_wrap=3, sharey=False)
g.map(plt.hist, 'comb', bins = bin_edges);
```



Additional Variation

If you have many categorical levels to plot, then you might want to add more arguments to the FacetGrid object's initialization to facilitate clarity in the conveyance of information. The example below includes a categorical variable, "trans", that has 27 different transmission types. Setting `col_wrap = 7` means that the plots will be organized into rows of 7 facets each, rather than a single long row of 27 plots.

Also, we want to have the facets for each transmission type in the decreasing order of combined fuel efficiency.

Example 3.

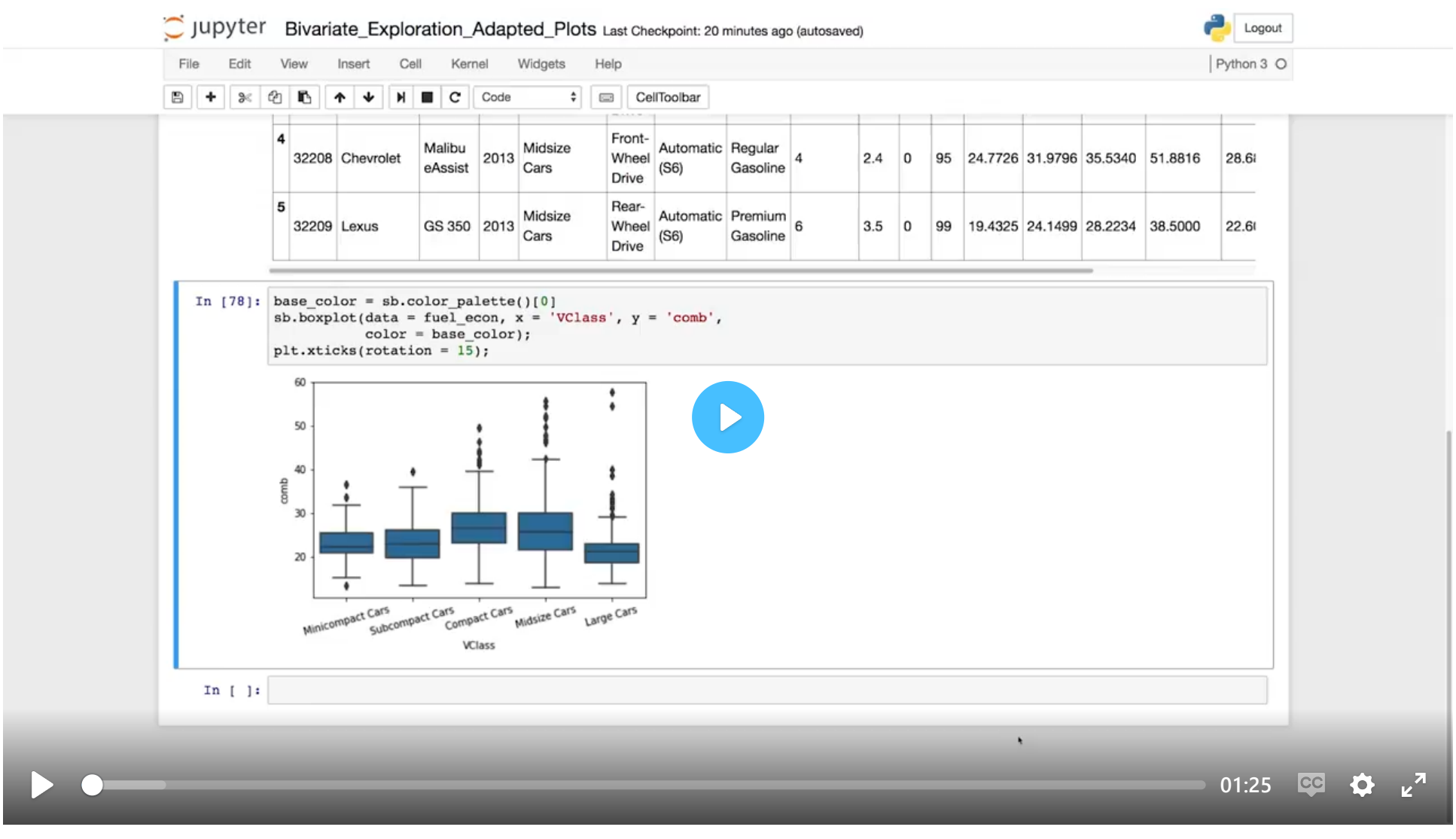
```
# Find the order in which you want to display the Facets
# For each transmission type, find the combined fuel efficiency
group_means = fuel_econ[['trans', 'comb']].groupby(['trans']).mean()

# Select only the list of transmission type in the decreasing order of combined fuel efficiency
group_order = group_means.sort_values(['comb'], ascending = False).index

# Use the argument col_order to display the FacetGrid in the desirable group_order
g = sb.FacetGrid(data = fuel_econ, col = 'trans', col_wrap = 7, col_order = group_order)
g.map(plt.hist, 'comb')
```

Next Concept

12. Adaptation of Univariate Plots

L4 121 Adaptations Of Univariate Plots V3

Data Vis L4 C12 V2



Adapted Bar Charts

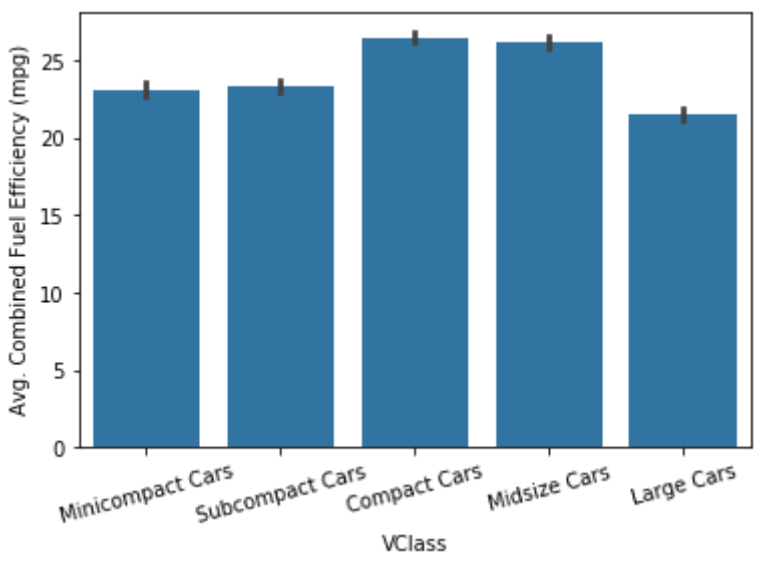
Histograms and bar charts were introduced in the previous lesson as depicting the distribution of numeric and categorical variables, respectively, with the height (or length) of bars indicating the number of data points that fell within each bar's range of values. These plots can be adapted for use as bivariate plots by, instead of indicating count by height, indicating a mean or other statistic on a second variable.

For example, we could plot a numeric variable against a categorical variable by adapting a bar chart so that its bar heights indicate the mean of the numeric variable. This is the purpose of seaborn's `barplot` function:

Example 1.

```
base_color = sb.color_palette()[0]
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=15);
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
```

Different hues are automatically assigned to each category level unless a fixed color is set in the "color" parameter, like in `countplot` and `violinplot`.



The bar heights indicate the mean value on the numeric variable, with error bars plotted to show the uncertainty in the mean based on variance and sample size.

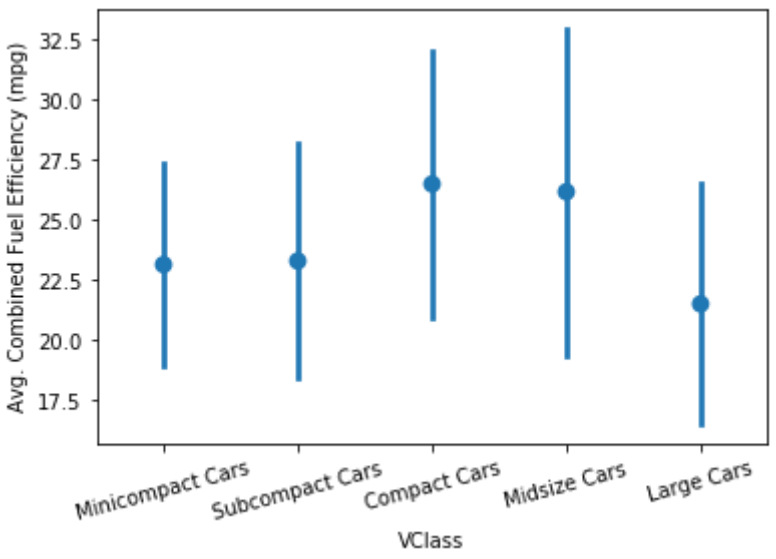
```
# Try these additional arguments
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color, errwidth=0)
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color, ci='sd')
```

As an alternative, the pointplot() function can be used to plot the averages as points rather than bars. This can be useful if having bars in reference to a 0 baseline aren't important or would be confusing.

Example 2.

```
sb.pointplot(data=fuel_econ, x='VClass', y='comb', color=base_color, ci='sd', linestyles="")
plt.xticks(rotation=15);
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
```

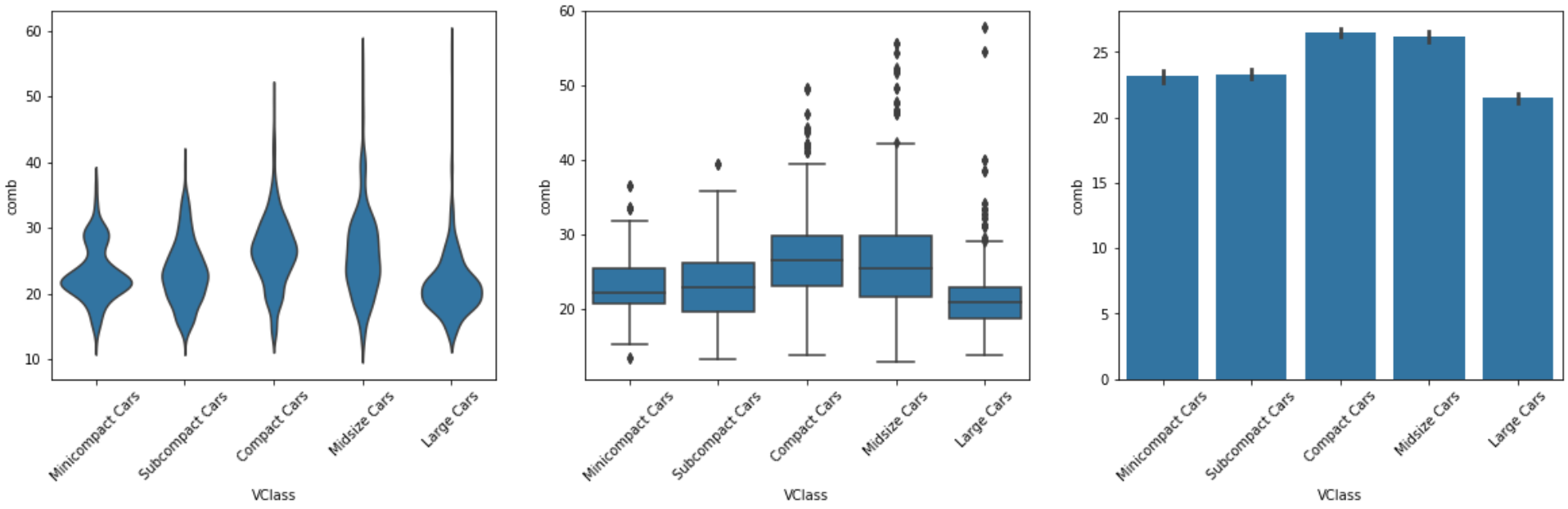
By default, `pointplot` will connect values by a line. This is fine if the categorical variable is ordinal in nature, but it can be a good idea to remove the line via `linestyles = ""` for nominal data.



The above plots can be useful alternatives to the box plot and violin plot if the data is not conducive to either of those plot types. For example, if the numeric variable is binary in nature, taking values only of 0 or 1, then a box plot or violin plot will not be informative, leaving the adapted bar chart as the best choice for displaying the data.

Example 3. Bringing a few charts together

```
plt.figure(figsize = [20, 5])
base_color = sb.color_palette()[0]

# left plot: violin plot
plt.subplot(1, 3, 1)
sb.violinplot(data=fuel_econ, x='VClass', y='comb', inner = None,
              color = base_color)
plt.xticks(rotation = 45); # include label rotation due to small subplot size

# center plot: box plot
plt.subplot(1, 3, 2)
sb.boxplot(data=fuel_econ, x='VClass', y='comb', color = base_color)
plt.xticks(rotation = 45);

# right plot: adapted bar chart
plt.subplot(1, 3, 3)
sb.barplot(data=fuel_econ, x='VClass', y='comb', color = base_color)
plt.xticks(rotation = 45);
```



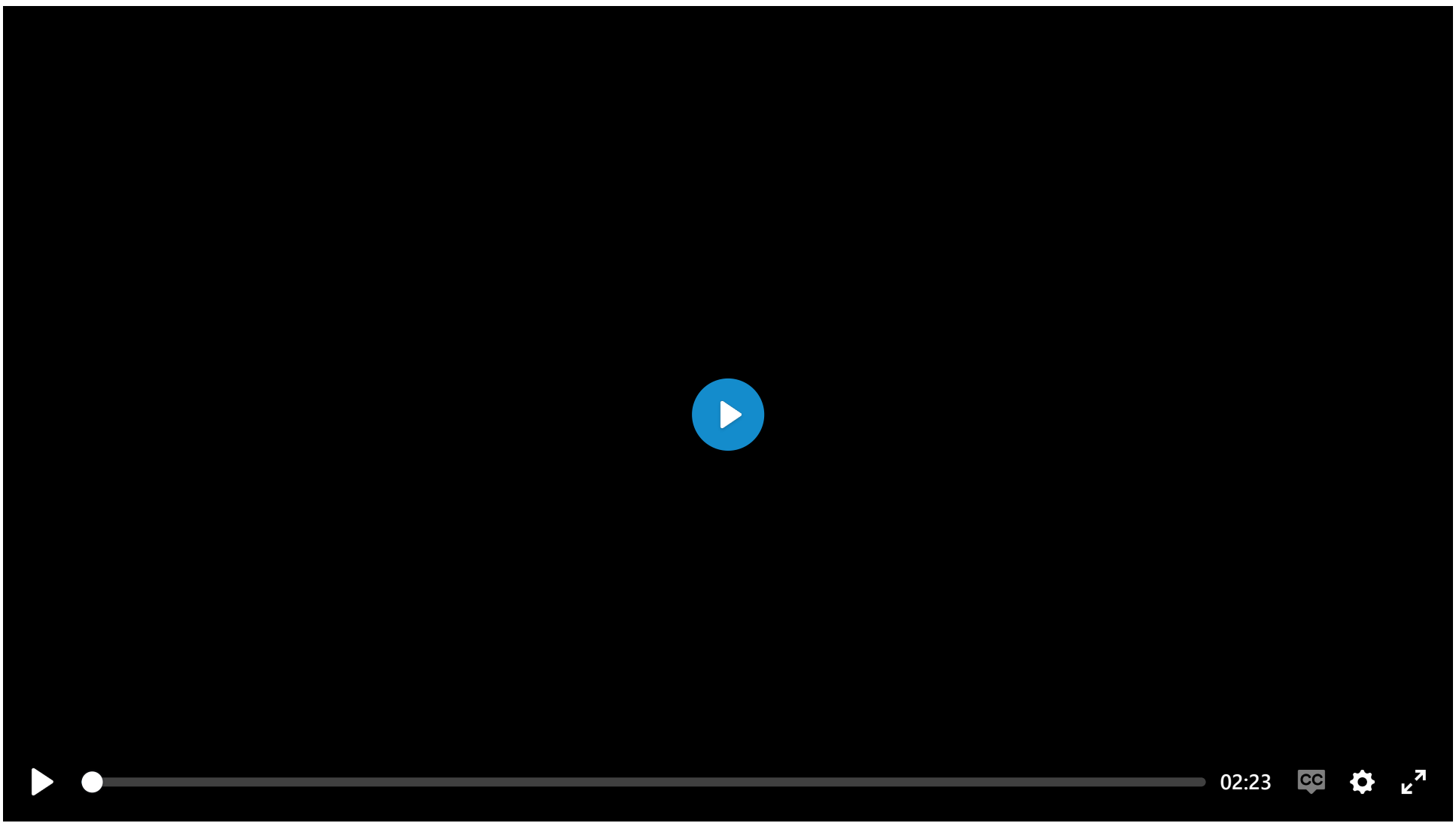
Do you know?
Matplotlib's hist() function can also be adapted so that bar heights indicate value other than a count of points through the use of the "weights" argument.

Next Concept

13. Line Plots

L4 131 Line Plots V1

Data Vis L4 C13 V1



Line Plots

The **line plot** is a fairly common plot type that is used to plot the trend of one numeric variable against values of a second variable. In contrast to a scatterplot, where all data points are plotted, in a line plot, only one point is plotted for every unique x-value or bin of x-values (like a histogram). If there are multiple observations in an x-bin, then the y-value of the point plotted in the line plot will be a summary statistic (like mean or median) of the data in the bin. The plotted points are connected with a line that emphasizes the sequential or connected nature of the x-values.

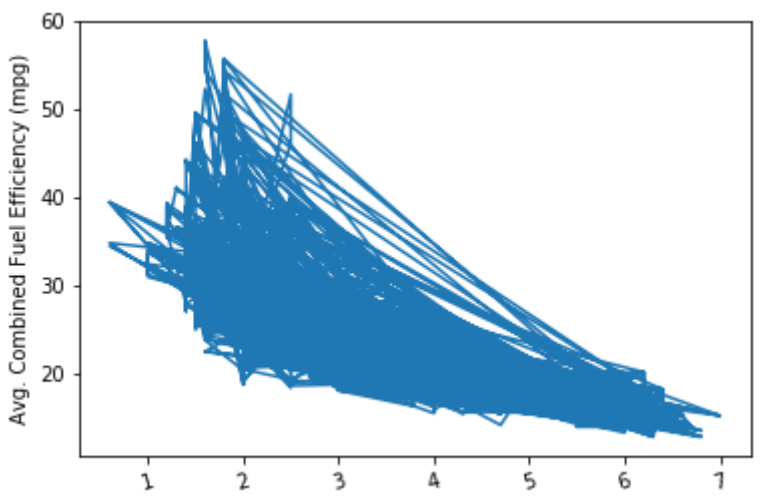
If the x-variable represents time, then a line plot of the data is frequently known as a **time series** plot. For example, we have only one observation per time period, like in stock or currency charts.

We will make use of Matplotlib's errorbar() function, performing some processing on the data in order to get it into its necessary form.

Let's see some examples below.

Example 1.

```
plt.errorbar(data=fuel_econ, x='displ', y='comb')
plt.xticks(rotation=15);
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```



If we just blindly stick a dataframe into the function without considering its structure, we might end up with a mess like the above. The function just plots all the data points as a line, connecting values from the first row of the dataframe to the last row. In order to create the line plot as intended, we need to do additional work to summarize the data.

Example 2.

```
# Set a number of bins into which the data will be grouped.
# Set bin edges, and compute center of each bin
bin_edges = np.arange(0.6, 7+0.2, 0.2)
bin_centers = xbin_edges[:-1] + 0.1

# Cut the bin values into discrete intervals. Returns a Series object.
displ_binned = pd.cut(fuel_econ['displ'], bin_edges, include_lowest = True)
displ_binned
```

```
0        (3.6, 3.8]
1        (1.8, 2.0]
2        (3.4, 3.6]
3        (3.4, 3.6]
4        (2.2, 2.4]
            ...
3924     (1.6, 1.8]
3925     (1.8, 2.0]
3926     (1.8, 2.0]
3927     (3.2, 3.4]
3928     (3.2, 3.4]
```

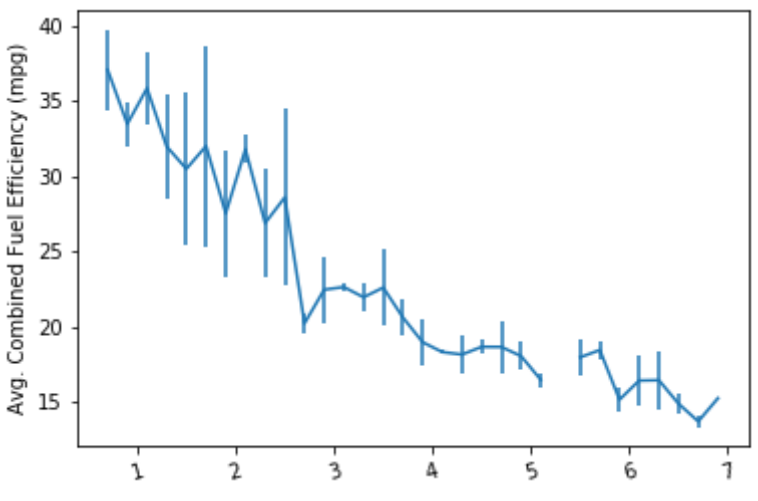
A series object returned when the `fuel_econ['displ']` column passed to the `pandas.cut()` function

---

```
# For the points in each bin, we compute the mean and standard error of the mean.
comb_mean = fuel_econ['comb'].groupby(displ_binned).mean()
comb_std = fuel_econ['comb'].groupby(displ_binned).std()

# Plot the summarized data
plt.errorbar(x=bin_centers, y=comb_mean, yerr=comb_std)
plt.xticks(rotation=15);
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```

Since the x-variable ('displ') is continuous, we first set a number of bins into which the data will be grouped. In addition to the usual edges, the center of each bin is also computed for later plotting. For the points in each bin, we compute the mean and standard error of the mean.

Documentation: Refer to the cut() function syntax.



Alternate Variations

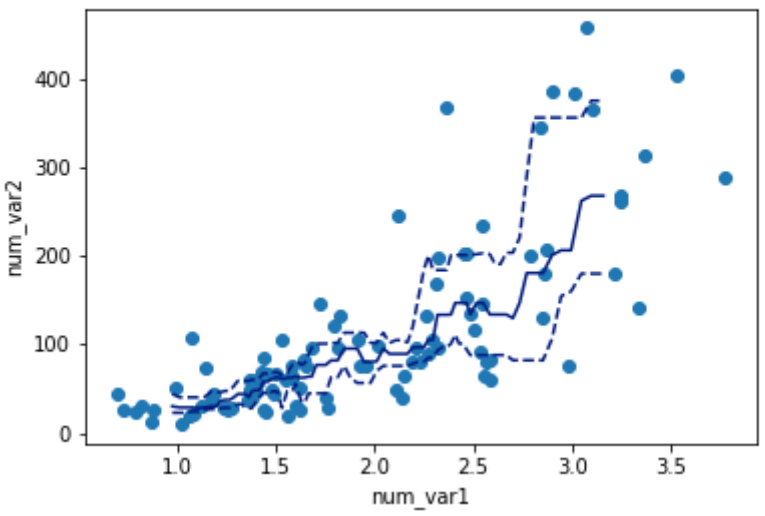
Note about the DataFrame object used in the examples below
The visualizations below are based on a synthetic dataframe object `df`, and show the plots based on its numeric (quantitative) variables, `num_var1`, `num_var2`, and a categorical (qualitative) variable, `cat_var`. **The new dataframe has been chosen to reflect the additional relationship between the selected variables.**

Instead of computing summary statistics on fixed bins, you can also make computations on a rolling window through use of pandas' `rolling` method. Since the rolling window will make computations on sequential rows of the dataframe, we should use `sort_values` to put the x-values in ascending order first.

Example 3.

```
# compute statistics in a rolling window
df_window = df.sort_values('num_var1').rolling(15)
x_winmean = df_window.mean()['num_var1']
y_median = df_window.median()['num_var2']
y_q1 = df_window.quantile(.25)['num_var2']
y_q3 = df_window.quantile(.75)['num_var2']

# plot the summarized data
base_color = sb.color_palette()[0]
line_color = sb.color_palette('dark')[0]
plt.scatter(data = df, x = 'num_var1', y = 'num_var2')
plt.errorbar(x = x_winmean, y = y_median, c = line_color)
plt.errorbar(x = x_winmean, y = y_q1, c = line_color, linestyle = '--')
plt.errorbar(x = x_winmean, y = y_q3, c = line_color, linestyle = '--')

plt.xlabel('num_var1')
plt.ylabel('num_var2')
```

Note that we're also not limited to just one line when plotting. When multiple Matplotlib functions are called one after the other, all of them will be plotted on the same axes. Instead of plotting the mean and error bars, we will plot the three central quartiles, laid on top of the scatterplot.

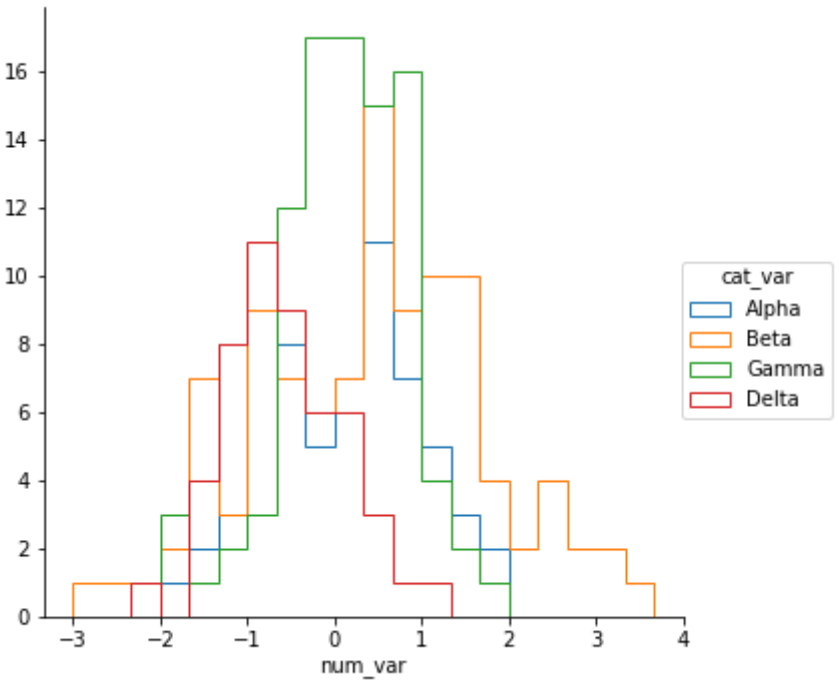


Another bivariate application of line plots is to plot the distribution of a numeric variable for different levels of a categorical variable. This is another alternative to using violin plots, box plots, and faceted histograms. With the line plot, one line is plotted for each category level, like overlapping the histograms on top of one another. This can be accomplished through multiple `errorbar` calls using the methods above, or by performing multiple `hist` calls, setting the "histtype = step" parameter so that the bars are depicted as unfilled lines.

Example 4.

```
bin_edges = np.arange(-3, df['num_var'].max()+1/3, 1/3)
g = sb.FacetGrid(data = df, hue = 'cat_var', size = 5)
g.map(plt.hist, "num_var", bins = bin_edges, histtype = 'step')
g.add_legend()
```

Note that I'm performing the multiple `hist` calls through the use of FacetGrid, setting the categorical variable on the "hue" parameter rather than the "col" parameter. You'll see more of this parameter of FacetGrid in the next lesson. I've also added an `add_legend` method call so that we can identify which level is associated with each curve.



Unfortunately, the "Alpha" curve seems to be pretty lost behind the other three curves since the relatively low number of counts is causing a lot of overlap. Perhaps connecting the centers of the bars with a line, like what was seen in the first `errorbar` example, would be better.

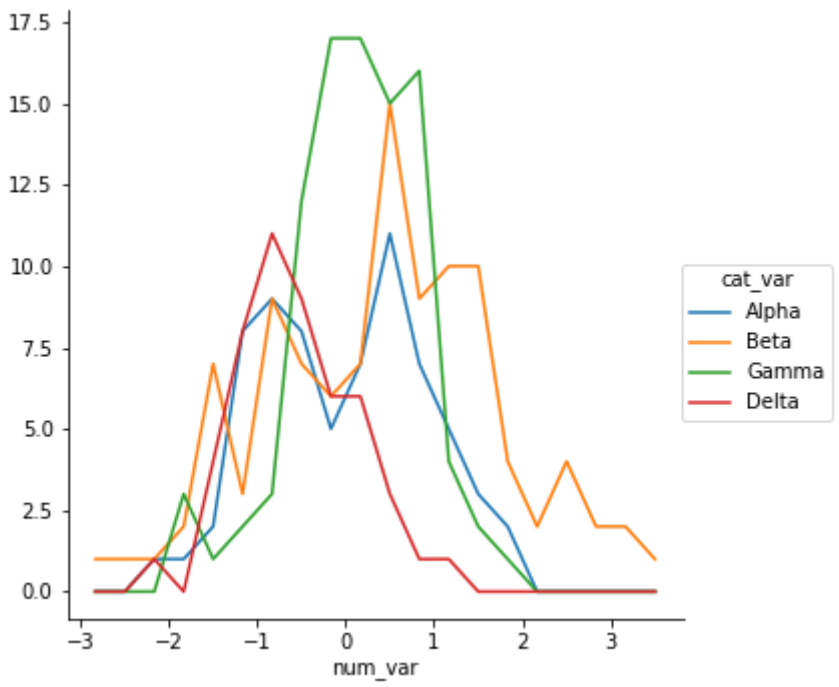
Functions you provide to the `map` method of FacetGrid objects do not need to be built-ins. Below, I've written a function to perform the summarization operations seen above to plot an `errorbar` line for each level of the categorical variable, then fed that function (`freq_poly`) to `map`.

Example 5.

```
def freq_poly(x, bins = 10, **kwargs):
    """ Custom frequency polygon / line plot code. """
    # set bin edges if none or int specified
    if type(bins) == int:
        bins = np.linspace(x.min(), x.max(), bins+1)
    bin_centers = (bin_edges[1:] + bin_edges[:-1]) / 2

    # compute counts
    data_bins = pd.cut(x, bins, right = False,
                       include_lowest = True)
    counts = x.groupby(data_bins).count()

    # create plot
    plt.errorbar(x = bin_centers, y = counts, **kwargs)

bin_edges = np.arange(-3, df['num_var'].max()+1/3, 1/3)
g = sb.FacetGrid(data = df, hue = 'cat_var', size = 5)
g.map(freq_poly, "num_var", bins = bin_edges)
g.add_legend()
```

`**kwargs` is used to allow additional keyword arguments to be set for the `errorbar` function.

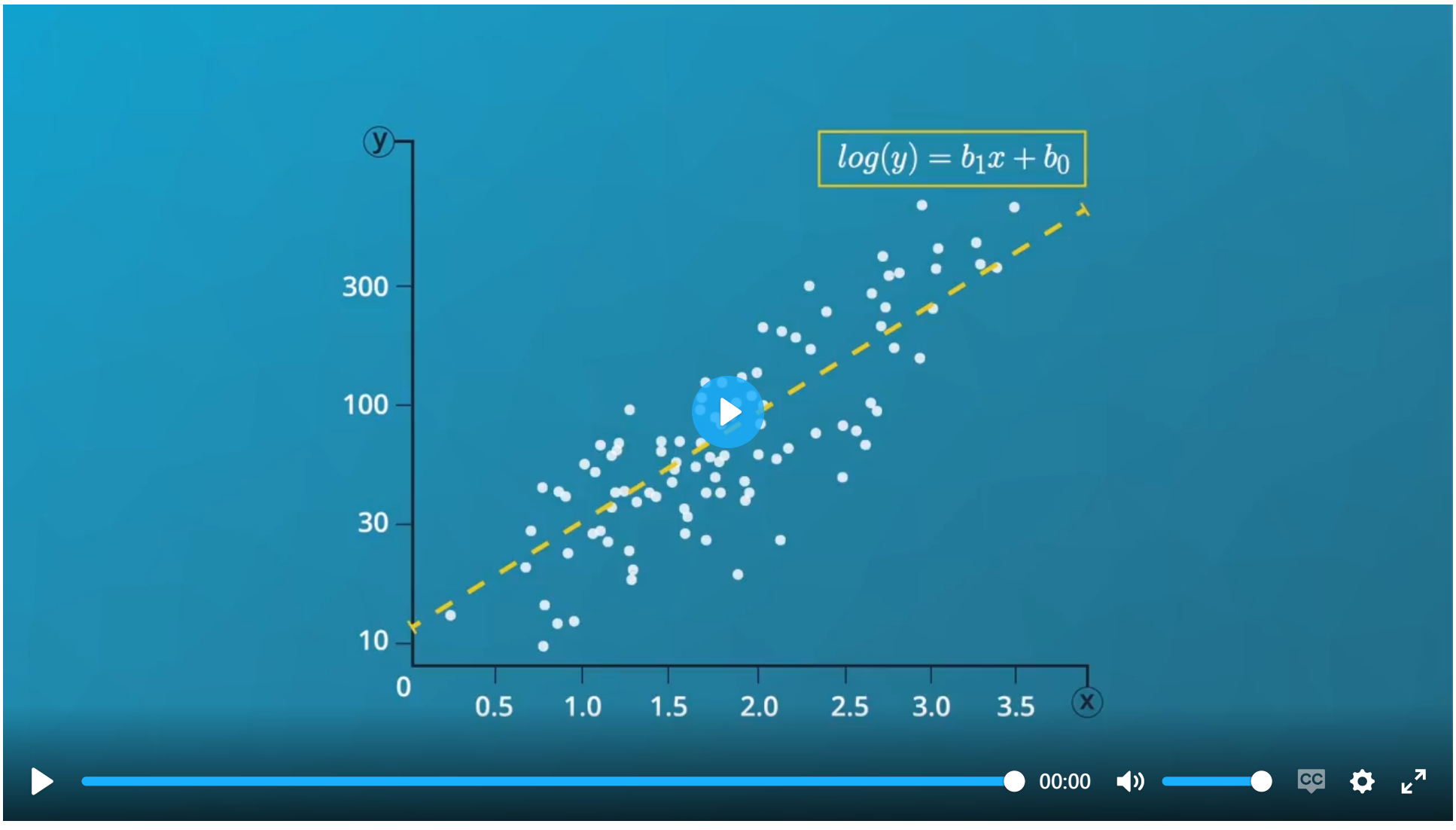
(Documentation: numpy `linspace`)



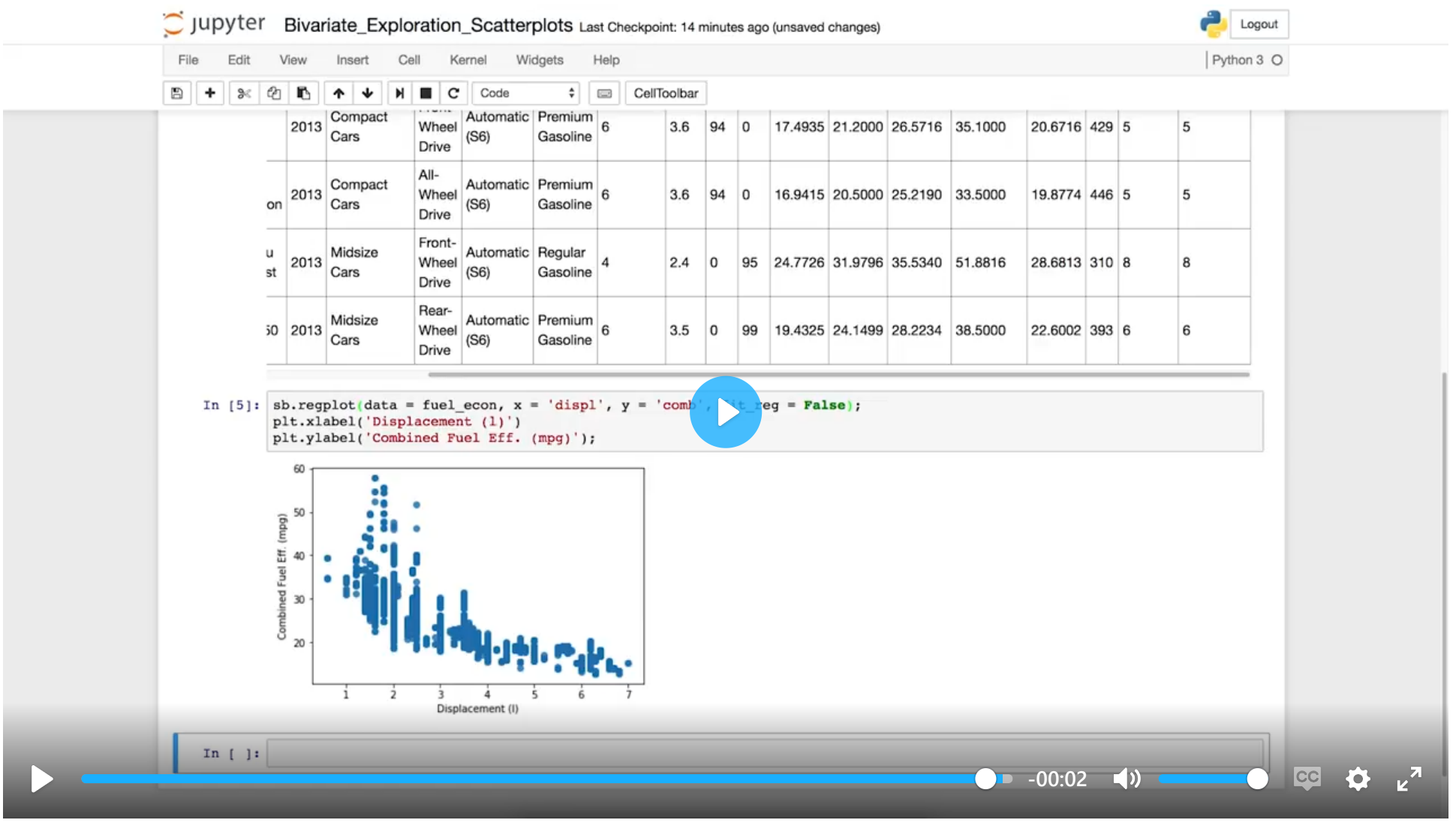
Next Concept

02. Scatterplots and Correlation

L4 021 Scatterplots And Correlation V2



Data Vis L4 C02 V1



Scatterplots

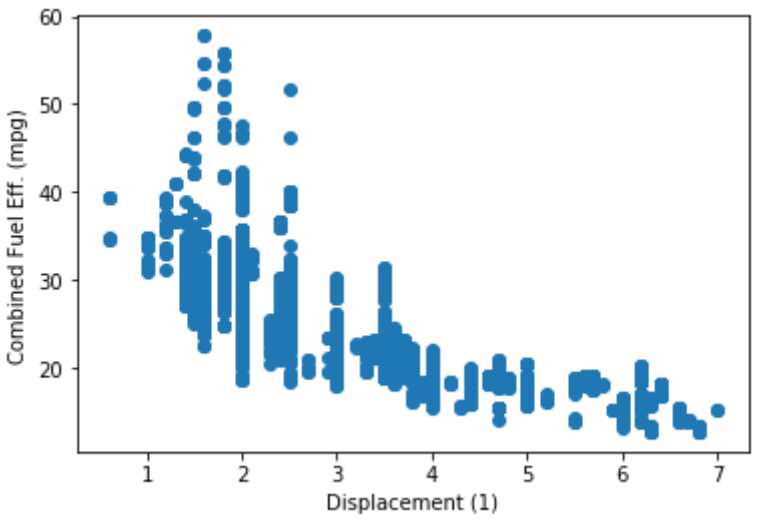
If we want to inspect the relationship between two numeric variables, the standard choice of plot is the **scatterplot**. In a scatterplot, each data point is plotted individually as a point, its x-position corresponding to one feature value and its y-position corresponding to the second.

matplotlib.pyplot.scatter()

One basic way of creating a scatterplot is through Matplotlib's `scatter` function:

Example 1 a. Scatter plot showing negative correlation between two variables

```
# TO DO: Necessary import

# Read the CSV file
fuel_econ = pd.read_csv('fuel_econ.csv')
fuel_econ.head(10)

# Scatter plot
plt.scatter(data = fuel_econ, x = 'displ', y = 'comb');
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)')
```



In the example above, the relationship between the two variables is negative because as higher values of the x-axis variable are increasing, the values of the variable plotted on the y-axis are decreasing.

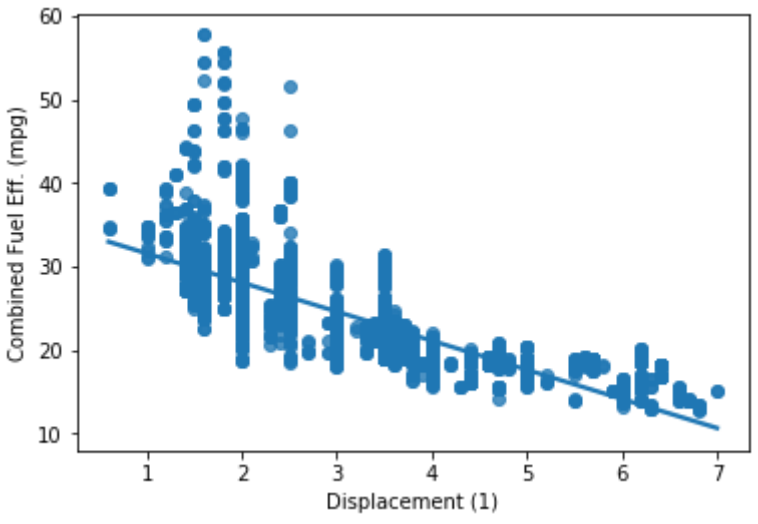
Alternative Approach - seaborn.regplot()

Seaborn's regplot() function combines scatterplot creation with regression function fitting:

Example 1 b. Scatter plot showing negative correlation between two variables

```
sb.regplot(data = fuel_econ, x = 'displ', y = 'comb');
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)')
```

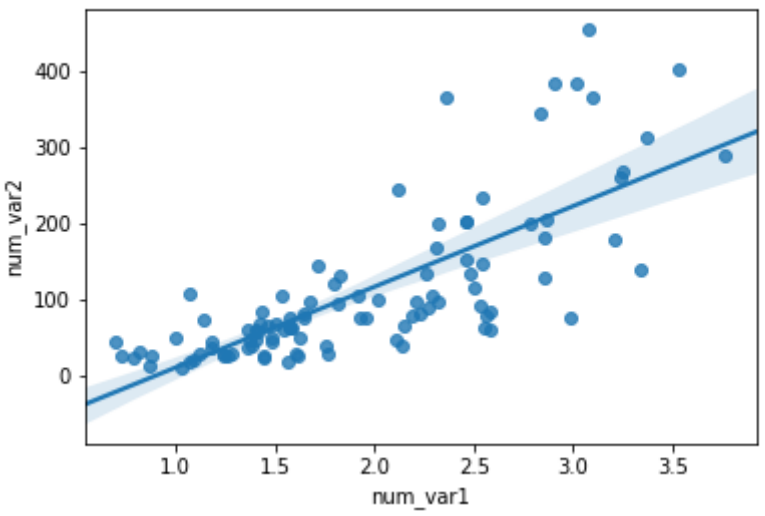
The basic function parameters, "data", "x", and "y" are the same for `regplot` as they are for matplotlib's `scatter`.



The regression line in a scatter plot showing a negative correlation between the two variables.

Example 2. Scatter plot showing a positive correlation between two variables

Let's consider another plot shown below that shows a positive correlation between two variables.



The regression line in a scatter plot showing a positive correlation between the two variables.

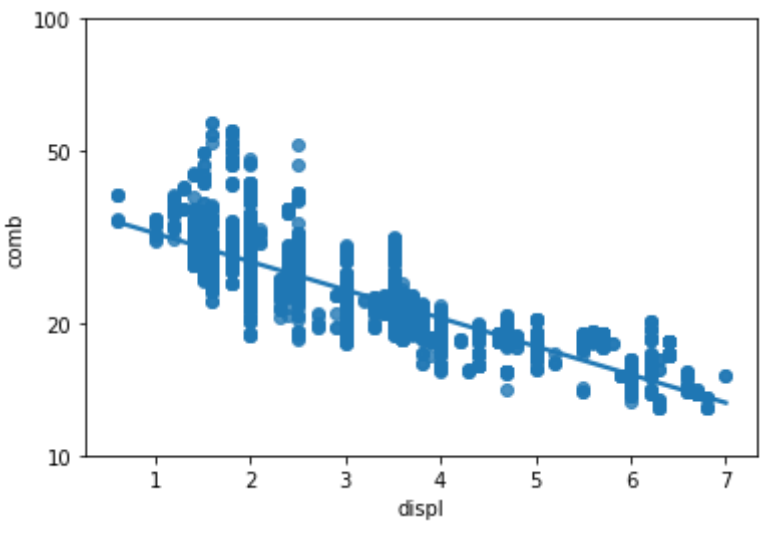
In the scatter plot above, by default, the regression function is linear and includes a shaded confidence region for the regression estimate. In this case, since the trend looks like a $\log(y) \propto x$ relationship (that is, linear increases in the value of x are associated with linear increases in the log of y), plotting the regression line on the raw units is not appropriate. If we don't care about the regression line, then we could set `fit_reg = False` in the `regplot` function call.

You can even plot the regression line on the transformed data as shown in the example below. For transformation, use a similar approach as you've learned in the last lesson.

Example 3. Plot the regression line on the transformed data

```
def log_trans(x, inverse = False):
    if not inverse:
        return np.log10(x)
    else:
        return np.power(10, x)

sb.regplot(fuel_econ['displ'], fuel_econ['comb'].apply(log_trans))
tick_locs = [10, 20, 50, 100]
plt.yticks(log_trans(tick_locs), tick_locs);
```

Note - In this example, the x- and y- values sent to `regplot` are set directly as Series, extracted from the dataframe.



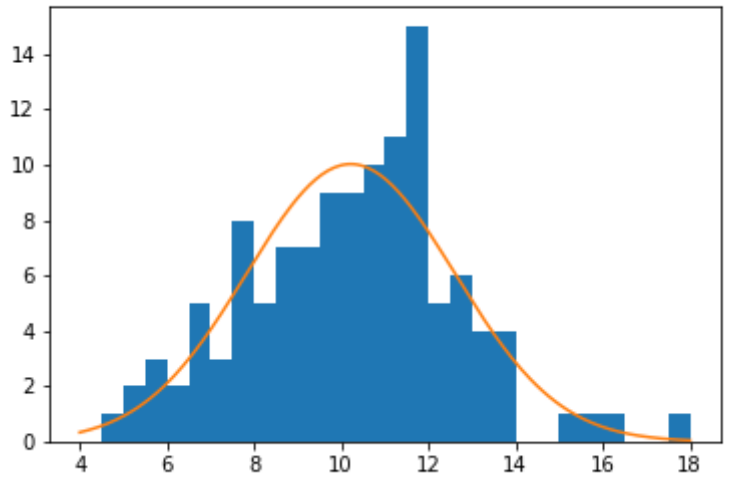
Regression line on a scattered plot based on the log-transformed data

16. Extra: Q-Q Plots

There might be cases where you are interested to see how closely your numeric data follows some hypothetical distribution. This might be important for certain parametric statistical tests, like checking for assumptions of normality. In cases like this, you can use a quantile-quantile plot, or **Q-Q plot**, to make a visual comparison between your data and your reference distribution. Take for example the following comparison of the following data and a hypothetical normal distribution using the sample statistics:

```
# create a histogram of the data
bin_size = 0.5
bin_edges = np.arange(4, 18 + bin_size, bin_size)
plt.hist(data = df, x = 'num_var', bins = bin_edges);

# overlay a theoretical normal distribution on top
samp_mean = df['num_var'].mean()
samp_sd = df['num_var'].std()

from scipy.stats import norm
x = np.linspace(4, 18, 200)
y = norm.pdf(x, samp_mean, samp_sd) # normal distribution heights
y *= df.shape[0] * bin_size # scale the distribution height

plt.plot(x, y)
```

The matplotlib `plot` function is a generic function for plotting y-values against x-values, by default a line connecting each x-y pair in sequence. In this case, I first use numpy's `linspace` function to generate x-values across the range of the plot. Note that the first two arguments match the `bin_edges` limits, while the third argument specifies the number of values to generate between the two endpoints. Then, I use the scipy package's `norm` class to get the height of the normal distribution curve at those x-values, using the sample mean and standard deviation as distribution parameters. `pdf` stands for probability density function, which returns the normal distribution height (density) at each value of x. These values are such that the total area under the curve will add up to 1. Since we've got a histogram with absolute counts on the y-axis, we need to scale the curve so it's on the same scale as the main plot: we do this by multiplying the curve heights by the number of data points and bin size. The code above gives us the following plot:



From a visual inspection of this overlaid plot, it looks like the data is a bit sparse on the right side compared to the expected normal distribution. There's also a bit of a spike of values between 11 and 12. On the other hand, the left side of the curve isn't too far off from the expected distribution, though it might be said that we might be missing some expected points in the left tail of the distribution. The question that we'd like to address is if there's enough evidence from what we've observed to say that the data is significantly different from the expected normal distribution.

One way we could approach this is through a statistical test, such as using scipy's `shapiro` function to perform the Shapiro-Wilk test. But since this is a course on data visualization, we'll inspect this question visually, using the Q-Q plot type teased at the top of the page. The main idea of the plot is this: if the data was normally distributed, then we'd expect a certain pattern in terms of how far each data point is from the mean of the distribution. If we order the points from smallest to largest, then we could compare how large the *k*-th ranked data point is against the *k*-th ranked point from the expected distribution.

To get these expected values, we'll make use of the `norm` class's `ppf` function, which stands for percent point function. The *ppf* takes as input a proportion (valued between 0 and 1) and returns the value in the distribution that would leave that proportion of the curve to the left. For a standard normal distribution (mean = 0, standard deviation = 1), the $ppf(0.25) = -0.674$, $ppf(0.5) = 0$, and $ppf(0.75) = 0.674$. The main question, then, is what values to stick into the *ppf*.

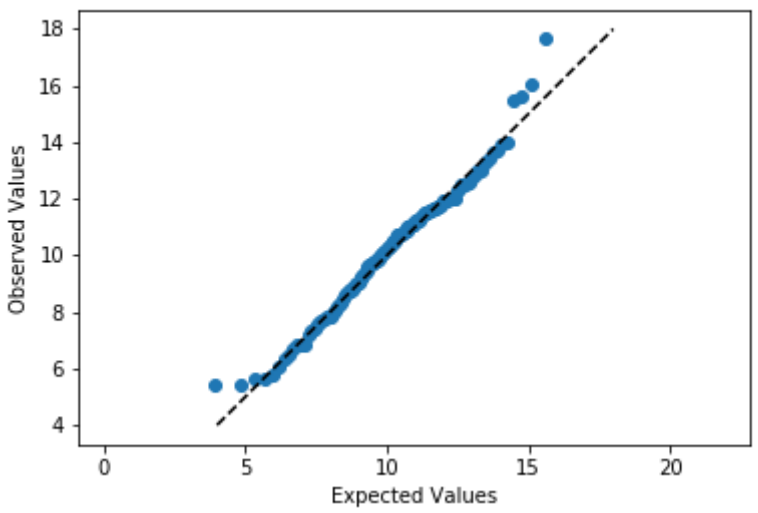
There's a few different conventions around this, but they generally take the form of the following equation:

> Given *n* data points, the *k*-th value should be at probability point $\frac{k-a}{n+1-2a}$, for some *a* between 0 and 1 (inclusive).

This equation distributes the probability points symmetrically about 0.5, and adjusting *a* changes how much probability is left in the tails of the [0,1] range. Commonly, *a* is set to a balanced value of 0.5, which gives the equation $\frac{k-0.5}{n}$. Let's put this all together using code:

```
n_points = df.shape[0]
qs = (np.arange(n_points) - .5) / n_points
expected_vals = norm.ppf(qs, samp_mean, samp_sd)

plt.scatter(expected_vals, df['num_var'].sort_values())
plt.plot([4,18],[4,18],'--', color = 'black')
plt.axis('equal')
plt.xlabel('Expected Values')
plt.ylabel('Observed Values')
```

It's a good idea to label the axes in this case. Since the actual and expected data are both on the same scale, the labels are a big help to keep things clear. In addition, rather than just plotting the expected and actual data alone, I've also added another `plot` call to add a diagonal x = y line. If the data matches the actual values perfectly on the expected value, they will fall directly on that diagonal line. The `plt.axis('equal')` line supports the visualization, as it will set the axis scaling to be equal, and the diagonal line will be at a 45 degree angle.

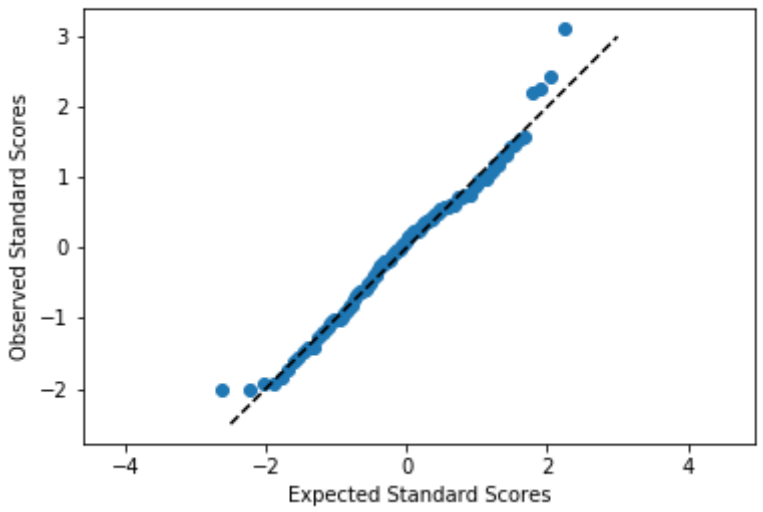


Excepting the smallest and largest few points, the distribution of observed values is actually fairly in line with the distribution of expected values – that is, it falls along the diagonal line. The smallest and largest observed points are larger than the values that would be expected from the normal distribution, but it's not by much. Given how much farther values can get spread out in the tails of the normal distribution, this shouldn't be a major concern. We're probably fine in treating the data as normally distributed.

Usually, the Q-Q plot is computed and rendered in terms of standardized units, rather than the scale of the original data. A standardized dataset has a mean of 0 and standard deviation of 1, so to convert a set of values into standard scores, we just need to subtract the sample mean from each value to center it around 0, then divide by the sample standard deviation to scale it. Calling methods of the

`norm` class without arguments for the mean or standard deviation assume the standard normal distribution. The code changes as follows:

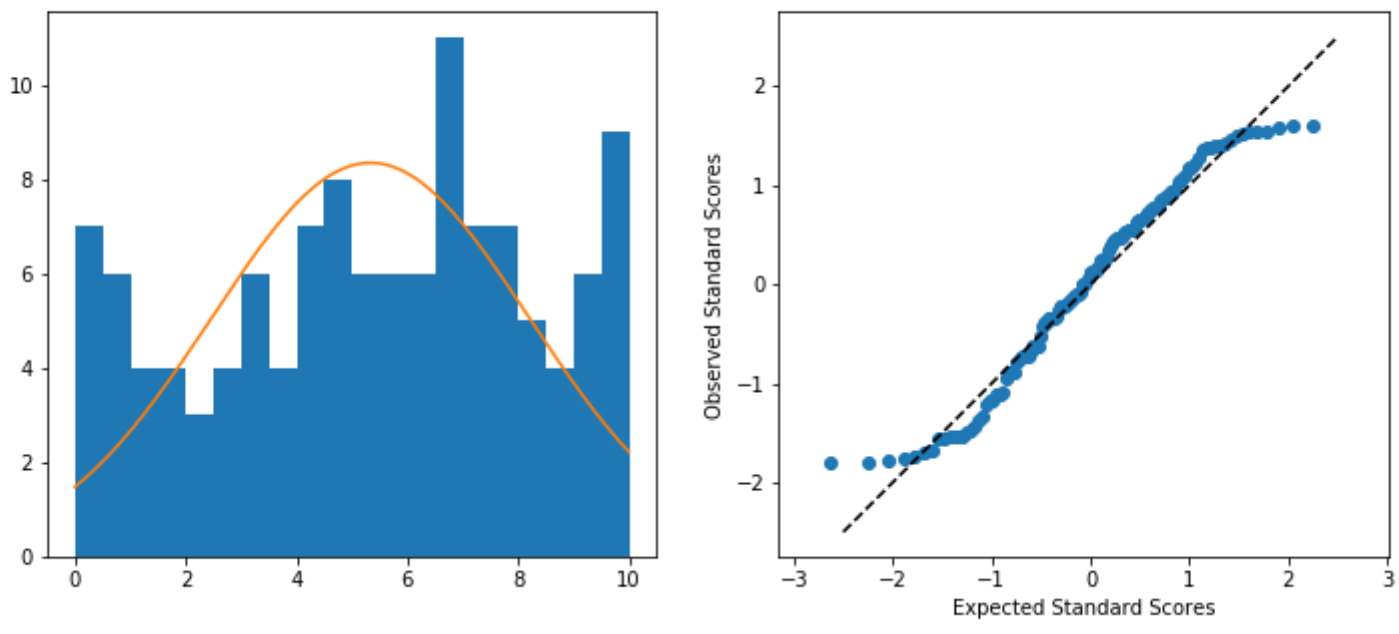
```
expected_scores = norm.ppf(qs)
data_scores = (df['num_var'].sort_values() - samp_mean) / samp_sd

plt.scatter(expected_scores, data_scores)
plt.plot([-2.5,3],[-2.5,3],'--', color = 'black')
plt.axis('equal')
plt.xlabel('Expected Standard Scores')
plt.ylabel('Observed Standard Scores')
```



Notice that the shape of the data has not changed since both datasets have been scaled in the exact same way. One of the reasons for performing this scaling is that it makes it easier to talk about the data values against the expected, theoretical distribution. In the first plot, there's no clear indication of where the center of the data lies, and how spread out the data is from that center. In the latter plot, we can use our expectations for how much of the data should be one or two standard deviations from the mean to better understand how the data is distributed. It also separates the values of the theoretical distribution from any properties of the observed data.

Before closing this page out, let's take a quick look at the Q-Q plot when the data distribution does *not* fit the normal distribution assumptions. Instead of generating data from a normal distribution, I'll now generate data from a uniform distribution:

```
# generate the data
np.random.seed(8322489)

n_points = 120
unif_data = np.random.uniform(0, 10, n_points)

# set up the figure
plt.figure(figsize = [12, 5])

# left subplot: plot the data
plt.subplot(1, 2, 1)
bin_size = 0.5
bin_edges = np.arange(0, 10 + bin_size, bin_size)
plt.hist(x = unif_data, bins = bin_edges);

# overlay a theoretical normal distribution on top
samp_mean = unif_data.mean()
samp_sd = unif_data.std()

from scipy.stats import norm
x = np.linspace(0, 10, 200)
y = norm.pdf(x, samp_mean, samp_sd) # normal distribution heights
y *= n_points * bin_size # scale the distribution height
plt.plot(x, y)

# right subplot: create a Q-Q plot
plt.subplot(1, 2, 2)

qs = (np.arange(n_points) - .5) / n_points
expected_scores = norm.ppf(qs)
data_scores = (np.sort(unif_data) - samp_mean) / samp_sd

plt.scatter(expected_scores, data_scores)
plt.plot([-2.5,2.5],[-2.5,2.5],'--', color = 'black')
plt.axis('equal')
plt.xlabel('Expected Standard Scores')
plt.ylabel('Observed Standard Scores')
```



Left: Original data; Right: Q-Q plot

When we compare the random standardized scores drawn from the uniform distribution to the expected scores from the theoretical normal distribution in the Q-Q plot, we see an S-shaped curve. The comparison of values in the middle of the curve are approximately linear in trend, but the slope is steeper than the desired y = x. Meanwhile on the edges, the slope is extremely shallow, as the uniform distribution is fixed to a finite range, but the normal distribution values in the tails are expected to be much further away. You can somewhat see this in the superimposed distribution line in the left-side plot, where even at the edges of the data, there is still quite a bit of height to the theoretical normal curve. All of this contributes to the result that the randomly-generated uniform data can't be well-approximated by the normal distribution.

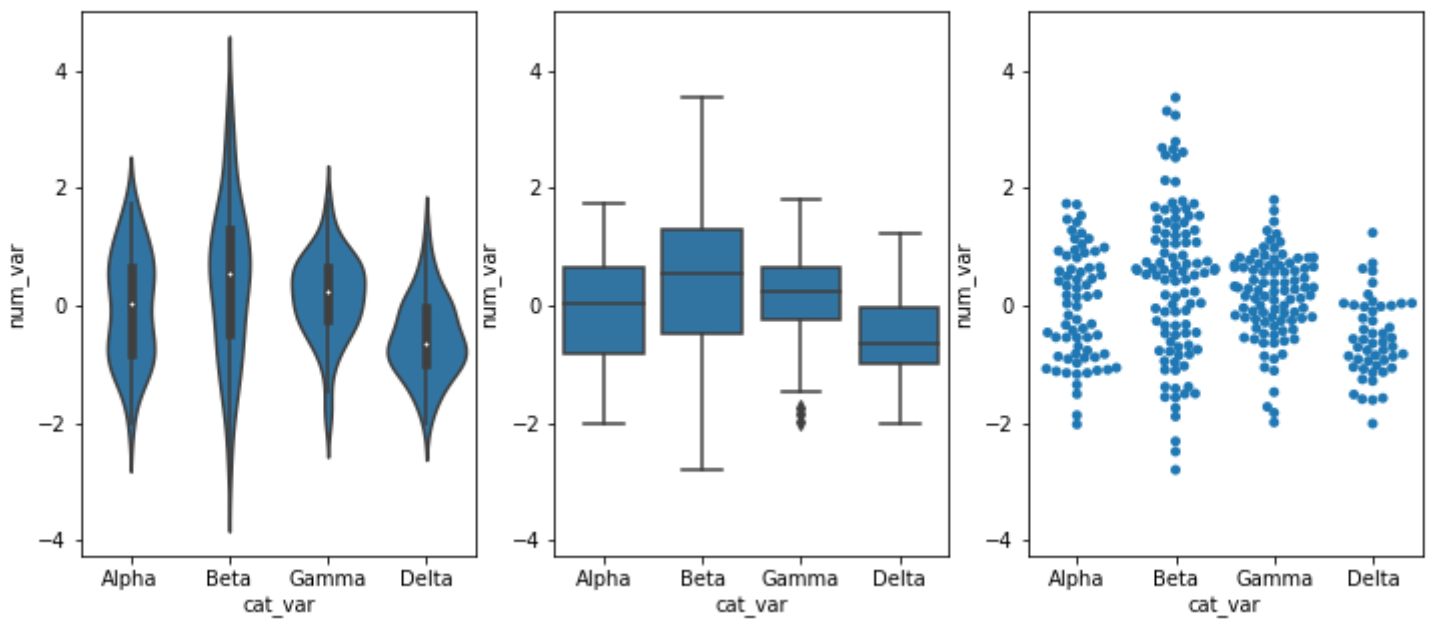
Next Concept

17. Extra: Swarm Plots

Swarm Plots

In this lesson, you saw many ways of depicting the relationship between a numeric variable and a categorical variable. Violin plots depicted distributions as density curves, while box plots took a more summary approach, plotting the quantiles as boxes with whiskers. Another alternative to these plots is the **swarm plot**. Similar to a scatterplot, each data point is plotted with position according to its value on the two variables being plotted. Instead of randomly jittering points as in a normal scatterplot, points are placed as close to their actual value as possible without allowing any overlap. A swarm plot can be created in seaborn using the `swarmplot` function, similar to how you would a call `violinplot` or `boxplot`.

```
plt.figure(figsize = [12, 5])
base_color = sb.color_palette()[0]

# left plot: violin plot
plt.subplot(1, 3, 1)
ax1 = sb.violinplot(data = df, x = 'cat_var', y = 'num_var', color = base_color)

# center plot: box plot
plt.subplot(1, 3, 2)
sb.boxplot(data = df, x = 'cat_var', y = 'num_var', color = base_color)
plt.ylim(ax1.get_ylim()) # set y-axis limits to be same as left plot

# right plot: swarm plot
plt.subplot(1, 3, 3)
sb.swarmplot(data = df, x = 'cat_var', y = 'num_var', color = base_color)
plt.ylim(ax1.get_ylim()) # set y-axis limits to be same as left plot
```



Looking at the plots side by side, you can see relative pros and cons of the swarm plot. Unlike the violin plot and box plot, every point is plotted, so we can now compare the frequency of each group in the same plot. While there is some distortion due to location jitter, we also have a more concrete picture of where the points actually lie, removing the long tails that can be present in violin plots.

However, it is only reasonable to use a swarm plot if we have a small or moderate amount of data. If we have too many points, then the restrictions against overlap will cause too much distortion or require a lot of space to plot the data comfortably. In addition, having too many points can actually be a distraction, making it harder to see the key signals in the visualization. Use your findings from univariate

visualizations to inform which bivariate visualizations will be best, or simply experiment with different plot types to see what is most informative.

Next Concept

18. Extra: Rug and Strip Plots

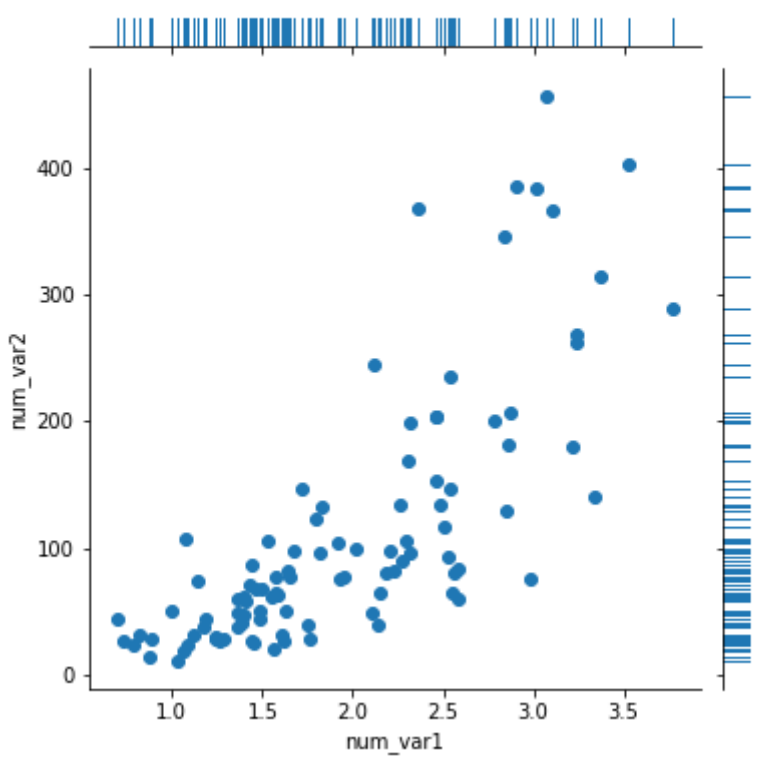
Rug and Strip Plots

You might encounter, or be interested in, marginal distributions that are plotted alongside bivariate plots such as scatterplots. A marginal distribution is simply the univariate distribution of a variable, ignoring the values of any other variable. For quantitative data, histograms or density curves are fine choices for marginal plot, but you might also see the **rug plot** employed. In a rug plot, all of the data points are plotted on a single axis, one tick mark or line for each one. Compared to a marginal histogram, the rug plot suffers somewhat in terms of readability of the distribution, but it is more compact in its representation of the data.

Seaborn's JointGrid class enables this plotting of bivariate relationship with marginal univariate plots for numeric data. The `plot_joint` method specifies a plotting function for the main, joint plot for the two variables, while the `plot_marginals` method specifies the plotting function for the two marginal plots. Here, we make use of seaborn's `rugplot` function.

```
g = sb.JointGrid(data = df, x = 'num_var1', y = 'num_var2')
g.plot_joint(plt.scatter)
g.plot_marginals(sb.rugplot, height = 0.25)
```

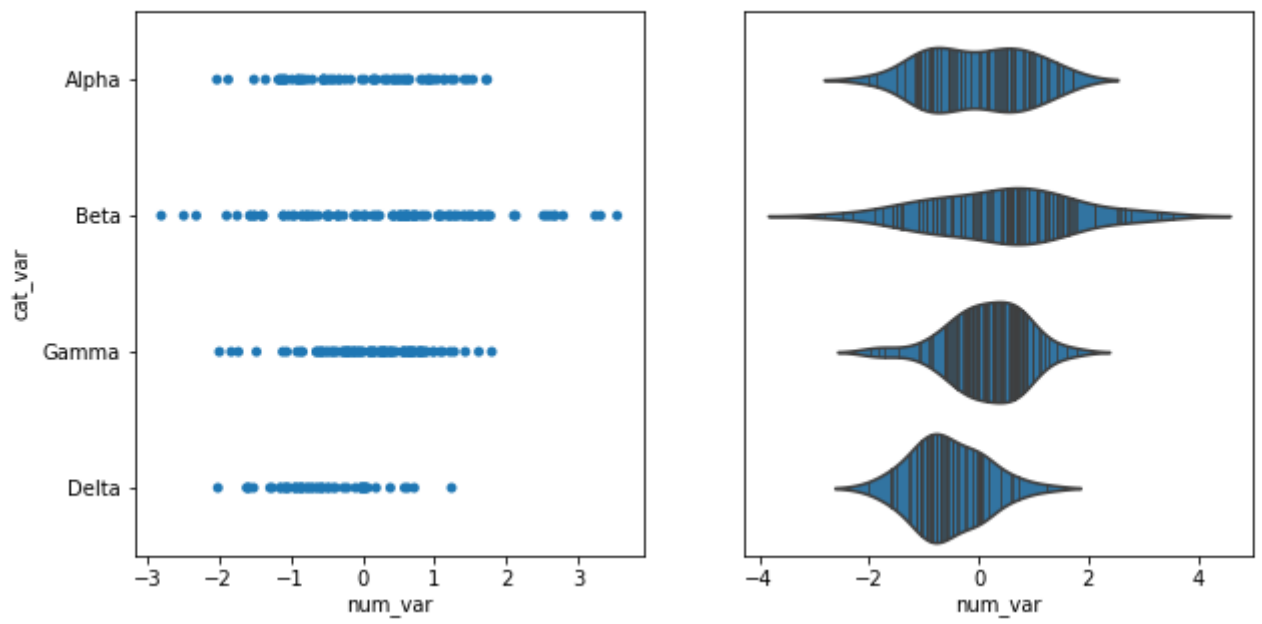
The "height" parameter specifies the rug ticks to be 0.25 the height of the marginal axis size.



The rug plot is fine here since the data isn't particularly numerous or overly dense. In other circumstances, a histogram or density curve will be more appropriate. You probably won't consider the rug plot as a primary plot choice, but it can be a good supporter plot in certain circumstances.

Another supporting plot type similar to the rug plot is the **strip plot**. It's like a swarm plot (see the previous page) but without any dodging or jittering to keep points separate or off the categorical line. You can also think of it as a rug plot faceted by categorical levels. You can use seaborn's `swarmplot` function to add a swarm plot to any other plot. The `inner = "stick"` and `inner = "point"` options can also be used with the `violinplot` function to include a swarm plot inside of the violin areas, instead of a box plot.

```
plt.figure(figsize = [10, 5])
base_color = sb.color_palette()[0]

# left plot: strip plot
plt.subplot(1, 2, 1)
ax1 = sb.stripplot(data = df, x = 'num_var', y = 'cat_var',
                   color = base_color)

# right plot: violin plot with inner strip plot as lines
plt.subplot(1, 2, 2)
sb.violinplot(data = df, x = 'num_var', y = 'cat_var', color = base_color,
              inner = 'stick')
```



Next Concept

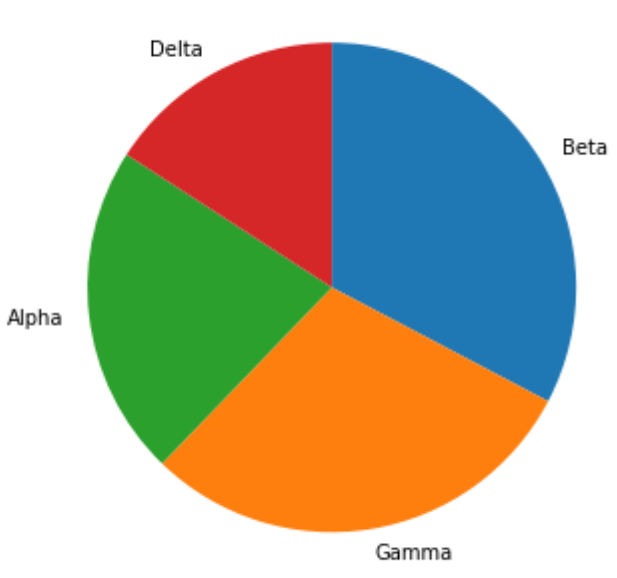
19. Extra: Stacked Plots

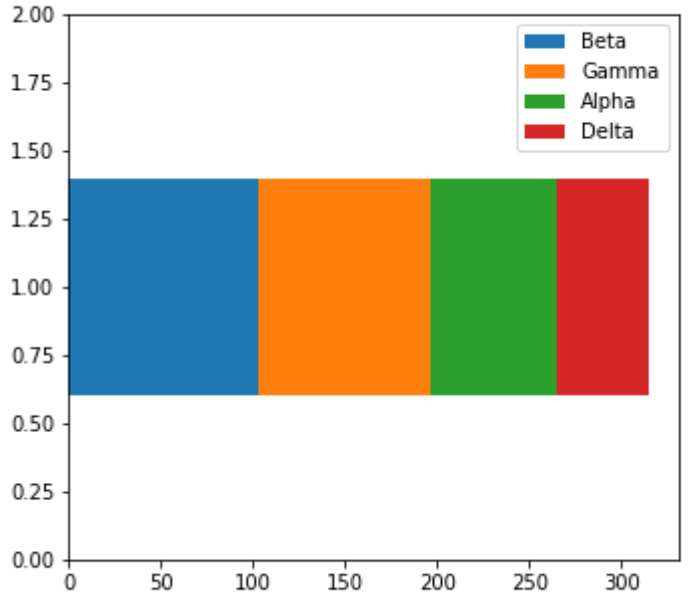
One common plotting technique has not been discussed thus far in the course, and that's **stacking**. Stacked bar charts and histograms are not uncommon, but there are often better plot choices available.

The most basic stacked chart takes a single bar representing the full count, and divides it into colored segments based on frequencies on a categorical variable. If this sounds familiar, that's because it almost perfectly coincides with the description of a pie chart, except that the shape being divided is different.

```
# pre-processing: count and sort by the number of instances of each category
sorted_counts = df['cat_var'].value_counts()

# establish the Figure
plt.figure(figsize = [12, 5])

# left plot: pie chart
plt.subplot(1, 2, 1)
plt.pie(sorted_counts, labels = sorted_counts.index, startangle = 90,
        counterclock = False);
plt.axis('square');

# right plot: horizontally stacked bar
plt.subplot(1, 2, 2)
baseline = 0
for i in range(sorted_counts.shape[0]):
    plt.barh(y = 1, width = sorted_counts[i], left = baseline)
    baseline += sorted_counts[i]

plt.legend(sorted_counts.index)  # add a legend for labeling
plt.ylim([0,2]) # give some vertical spacing around the bar
```

The stacked bar is built through successive calls of the matplotlib `barh` function; each time the function is called, the bar that is plotted is assigned a new color. The choice of "y" is arbitrary: it'll just center the bar around y = 1, but it doesn't have any inherent meaning. The "left" parameter specifies the left edge of each bar added to the stack, which starts at the `baseline` of 0 and is built up with each stacked bar. Note in this case that the bar is being plotted with absolute counts, rather than proportions. A discussion of absolute vs. relative frequencies will come later down the page!





Given this similarity, cautions regarding use of the stacked bar are fairly similar to that of the pie chart:

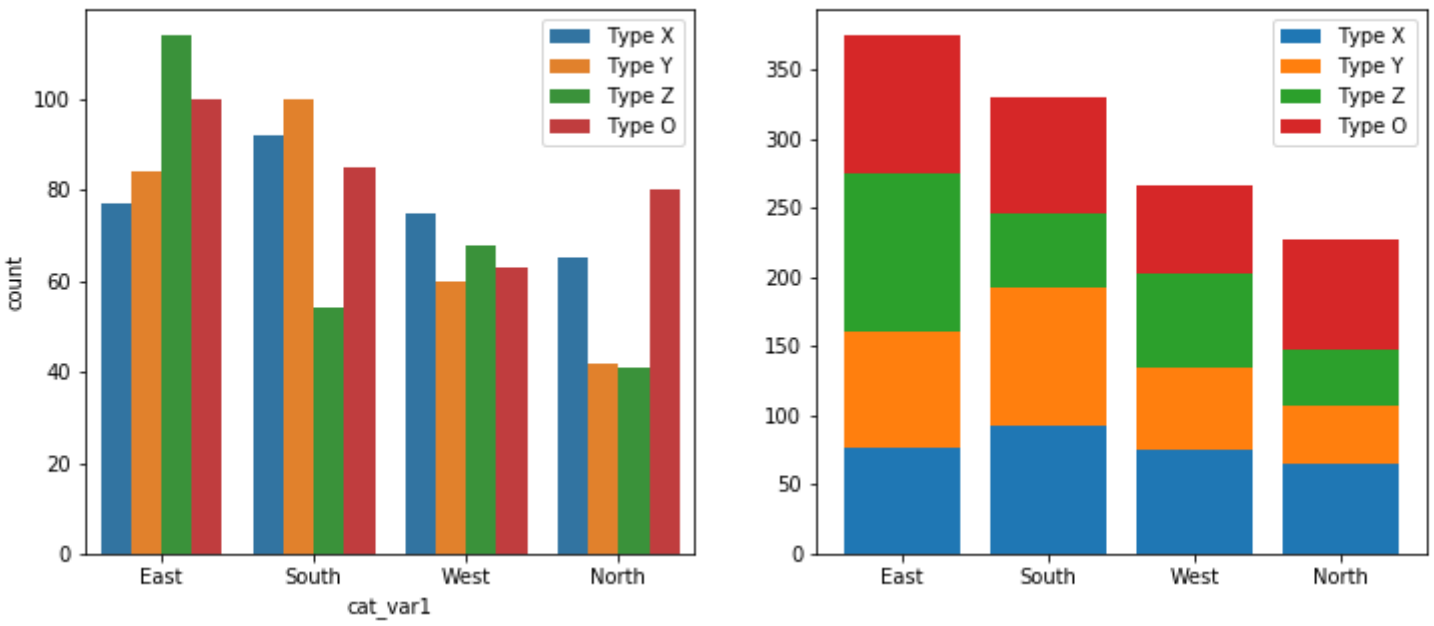
- Make sure that relative frequencies are a meaningful comparison.
- Try to limit yourself to a small number of categories, up to about five.
- Make sure that categories are arranged in a sensible order, e.g. by frequency for nominal data or by levels for ordinal data.

Otherwise, the standard bar chart is a reliable option that should be used in most cases. Only use the pie chart or singly divided bar if there's a compelling reason to do so.

The debate becomes more interesting when multiple features get involved. When should we feel free to create a stacked bar chart versus using a clustered bar chart? There are two major categories of stacked bar chart that I want to focus on here: plotting by absolute frequency and plotting by relative frequency. We'll start with code for an absolute frequency stacked chart below.

```
cat1_order = ['East', 'South', 'West', 'North']
cat2_order = ['Type X', 'Type Y', 'Type Z', 'Type O']

plt.figure(figsize = [12, 5])

# left plot: clustered bar chart, absolute counts
plt.subplot(1, 2, 1)
sb.countplot(data = df, x = 'cat_var1', hue = 'cat_var2',
             order = cat1_order, hue_order = cat2_order)
plt.legend()

# right plot: stacked bar chart, absolute counts
plt.subplot(1, 2, 2)

baselines = np.zeros(len(cat1_order))
# for each second-variable category:
for i in range(len(cat2_order)):
    # isolate the counts of the first category,
    cat2 = cat2_order[i]
    inner_counts = df[df['cat_var2'] == cat2]['cat_var1'].value_counts()
    # then plot those counts on top of the accumulated baseline
    plt.bar(x = np.arange(len(cat1_order)), height = inner_counts[cat1_order],
            bottom = baselines)
    baselines += inner_counts[cat1_order]

plt.xticks(np.arange(len(cat1_order)), cat1_order)
plt.legend(cat2_order)
```

The strategy for this plot is very similar to the single stacked bar shown above, except that we're using the standard `bar` with "x" and "bottom" parameters, and that `baselines` is a list of base heights. We want to create all of the bars for a particular secondary category at the same time so that creation of the legend has a 1:1 mapping to `bar` calls. You might notice below that the order of labels in the legend is the reverse of the order in which the bars are stacked. You'll see code to handle this in the relative frequency plot below!

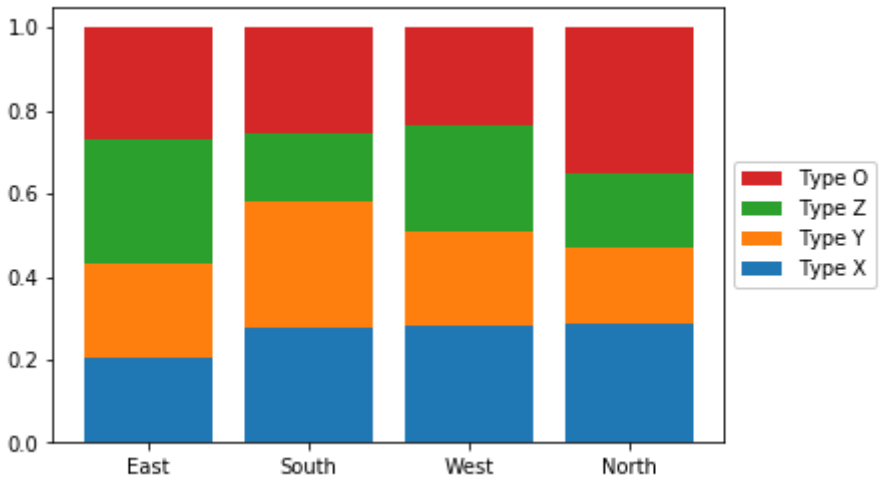


The stacked bar chart plotted by absolute frequency carries one big advantage over the clustered bar chart: for the variable plotted on the x-axis, it's clear which category level has the highest frequency, in this case "East". The values of this variable can be interpreted just like the univariate bar chart. The disadvantage of the stacked bar chart comes with interpretation of the second, stacked variable. If you want to compare the relative counts of this second variable across levels of the first, you can really only do that for the category plotted on the baseline, which in this case is the blue one, "Type X". For the remaining categories, it's much harder to make the comparison of values – you can't really tell that the counts of "Type O" are larger in the "South" than the "North" from the stacked chart, where it's directly comparable in the clustered bar chart.

Now, let's take a look at what happens when we create the stacked bar chart with relative frequencies instead, where each bar is scaled to a total height of 1.

```
cat1_order = ['East', 'South', 'West', 'North']
cat2_order = ['Type X', 'Type Y', 'Type Z', 'Type O']

artists = [] # for storing references to plot elements
baselines = np.zeros(len(cat1_order))
cat1_counts = df['cat_var1'].value_counts()

# for each second-variable category:
for i in range(len(cat2_order)):
    # isolate the counts of the first category,
    cat2 = cat2_order[i]
    inner_counts = df[df['cat_var2'] == cat2]['cat_var1'].value_counts()
    inner_props = inner_counts / cat1_counts
    # then plot those counts on top of the accumulated baseline
    bars = plt.bar(x = np.arange(len(cat1_order)),
                   height = inner_props[cat1_order],
                   bottom = baselines)
    artists.append(bars)
    baselines += inner_props[cat1_order]

plt.xticks(np.arange(len(cat1_order)), cat1_order)
plt.legend(reversed(artists), reversed(cat2_order), framealpha = 1,
           bbox_to_anchor = (1, 0.5), loc = 6);
```

There are two main changes to this code compared to the previous plot. First of all, the `cat1_counts` variable has been computed to change the absolute frequencies into relative frequencies within each x-axis category. Secondly, some code has been added to reverse the order of bars in the legend. The `artists` variable has been added to store references to each of the bar groups added from each `bar` call. Then in the `legend` function call, we make use of the built-in Python function `reversed` to reverse the order in which the artists and labels are included in the legend. The additional parameters affect the positioning of the legend: setting an anchor for the legend box on the right side of the plot via "bbox_to_anchor", and positioning the anchor to the legend's left with "loc = 6".



Since the bars are all the same height of 1 with a relative frequency stacked bar chart, we lose the ability to compare the absolute counts on the categorical variable plotted on the x-axis (i.e. we can't tell that "East" has the most counts and "North" the least amount). In exchange, we can now compare the relative prevalence of the stacked variable on both the first category on the bottom ("Type X") as well as the category on the top ("Type O"). We can now see that, in terms of relative frequency, "Type X" has a fairly consistent presence in "South", "West", and "North", and that "Type O" has its highest relative frequency in "North". Unfortunately, this still doesn't help us make easy comparisons about the "Type Y" and "Type Z" categories that are sandwiched in between. This major limitation is a big reason why other plot types like clustered bar or line charts are often preferable to stacking.

Further Reading

- Eager Eyes: Stacked Bars are the Worst

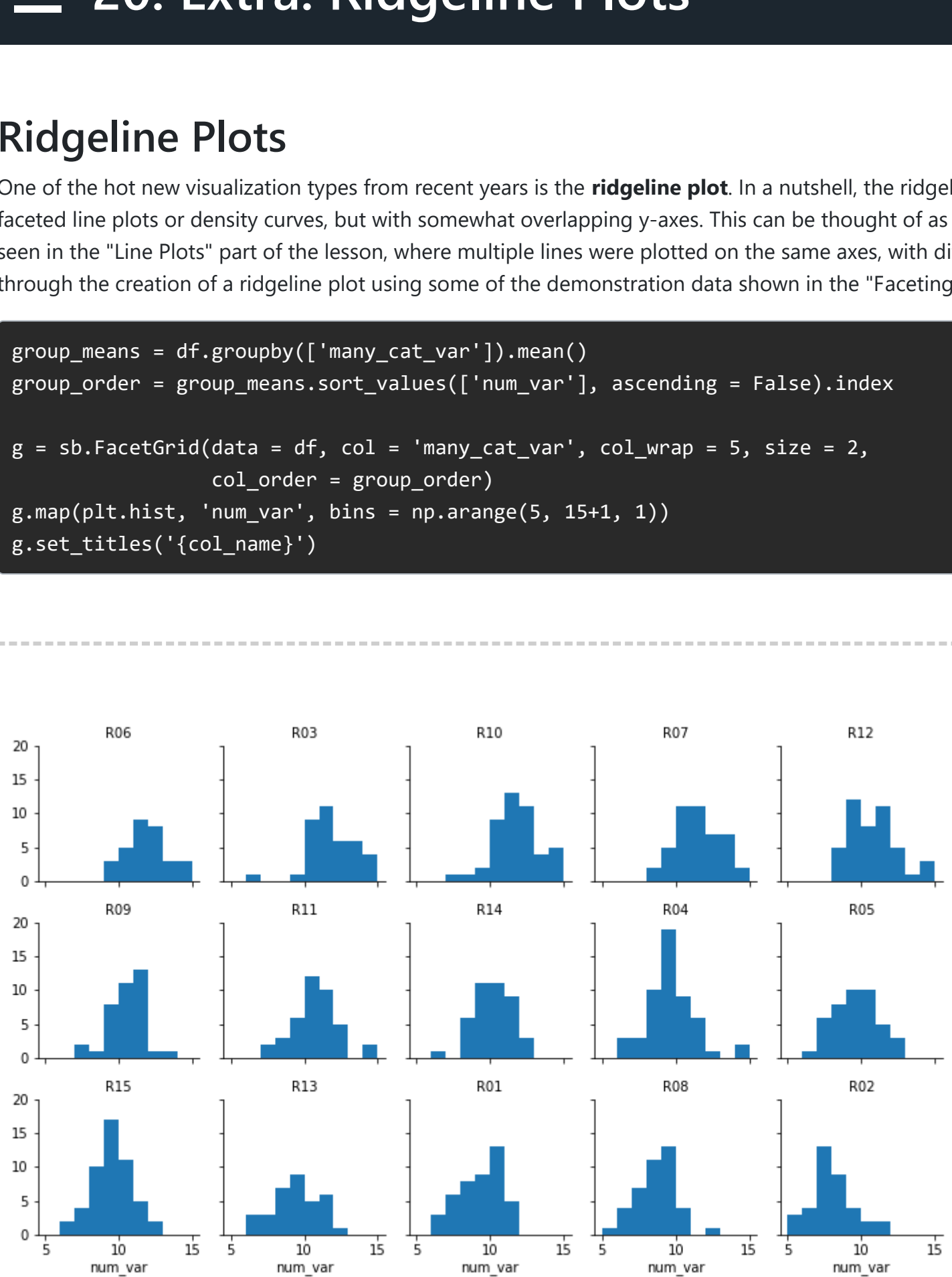
Next Concept

20. Extra: Ridgeline Plots

Ridgeline Plots

One of the hot new visualization types from recent years is the **ridgeline plot**. In a nutshell, the ridgeline plot is a series of vertically faceted line plots or density curves, but with somewhat overlapping y-axes. This can be thought of as a contrast to the line plot variation seen in the "Line Plots" part of the lesson, where multiple lines were plotted on the same axes, with different hues. On this page, I'll walk through the creation of a ridgeline plot using some of the demonstration data shown in the "Faceting" page:

```
group_means = df.groupby(['many_cat_var']).mean()
group_order = group_means.sort_values(['num_var'], ascending = False).index

g = sb.FacetGrid(data = df, col = 'many_cat_var', col_wrap = 5, size = 2,
                 col_order = group_order)
g.map(plt.hist, 'num_var', bins = np.arange(5, 15+1, 1))
g.set_titles('{col_name}')
```

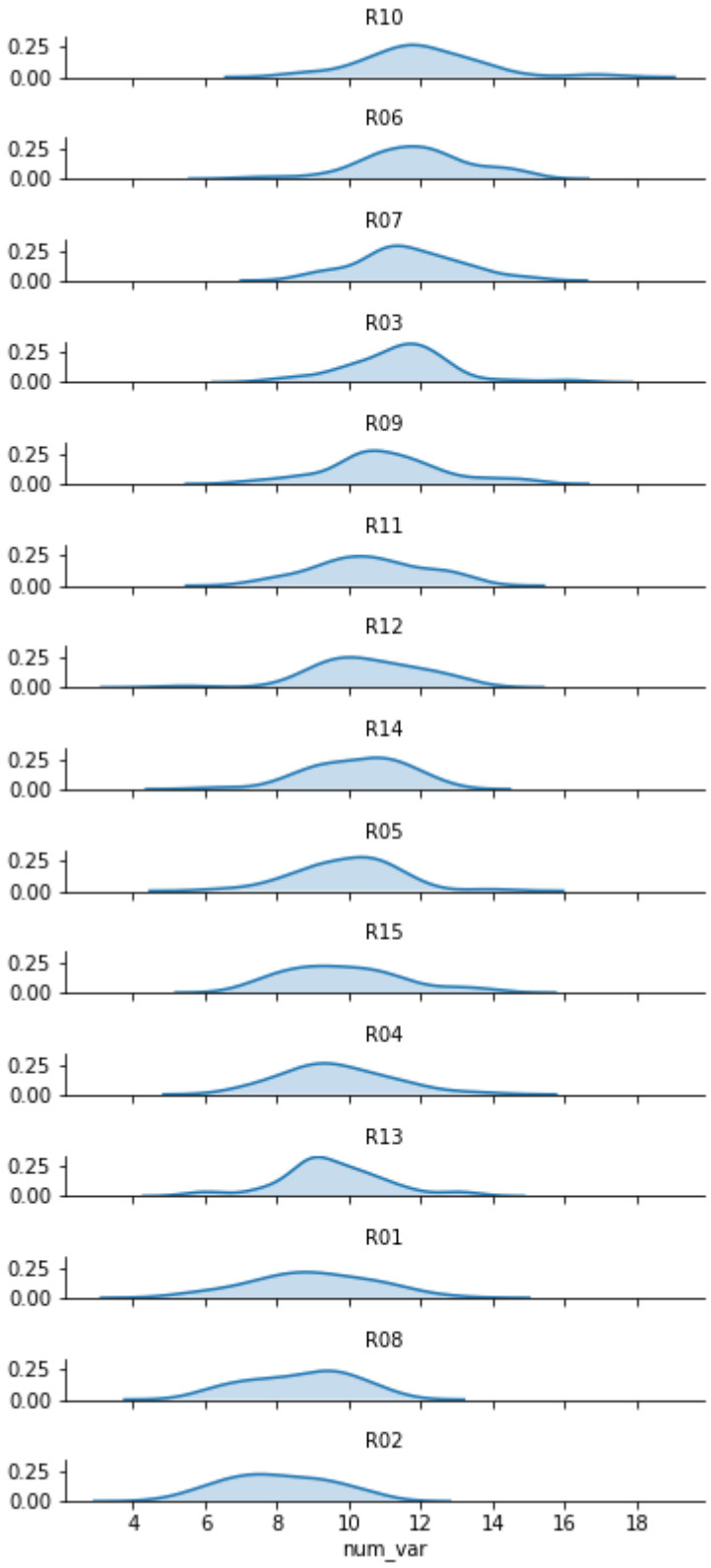


Faceted plot, whose data will be converted into a ridgeline form

Two things immediately come to mind for changing the faceted histograms into a ridgeline plot. First of all, changing the form of the distribution plots from histograms to kernel density estimates (as seen in the Extras of the previous lesson) will make the overlaps a bit smoother. Second, we need to facet the levels by rows so that they're all stacked up on top of one another.

```
group_means = df.groupby(['many_cat_var']).mean()
group_order = group_means.sort_values(['num_var'], ascending = False).index

g = sb.FacetGrid(data = df, row = 'many_cat_var', size = 0.75, aspect = 7,
                 row_order = group_order)
g.map(sb.kdeplot, 'num_var', shade = True)
g.set_titles('{row_name}')
```

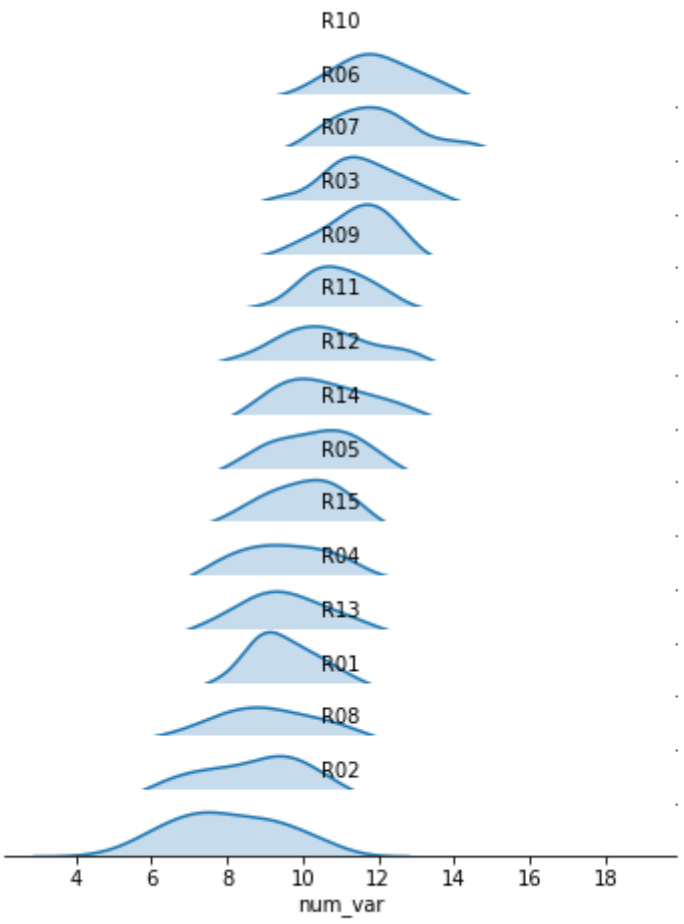
`FacetGrid` and `set_titles` change "col" to "row", also removing "col_wrap". The "size" and "aspect" dimensions have also been adjusted for the large vertical stacking of facets. The `map` function changes to `kdeplot` and removes "bins", adding the "shade" parameter in its place.



Now we've got all of the group distributions stacked on top of one another for a uni-dimensional comparison, but the plot's still pretty tall. Next, we'll create some overlap between the individual subplots.

```
group_means = df.groupby(['many_cat_var']).mean()
group_order = group_means.sort_values(['num_var'], ascending = False).index

# adjust the spacing of subplots with gridspec_kws
g = sb.FacetGrid(data = df, row = 'many_cat_var', size = 0.5, aspect = 12,
                 row_order = group_order, gridspec_kws = {'hspace' : -0.2})
g.map(sb.kdeplot, 'num_var', shade = True)

# remove the y-axes
g.set(yticks=[])
g.despine(left=True)

g.set_titles('{row_name}')
```

I've added the "gridspec_kws" parameter to the `FacetGrid` call to adjust the arrangement of subplots in the grid through Matplotlib's `GridSpec` class. By setting "hspec" to a negative value, the subplot axes bounds will overlap vertically. The "size" and "aspect" parameters have also been adjusted. While I'm at it, I'll add some code on the `FacetGrid` object to remove the y-axis through the `despine` method and remove the ticks through the `set` method. They're going to start overlapping, and we don't really need them – we're mostly interested in the relative positions of the distributions rather than specific heights.



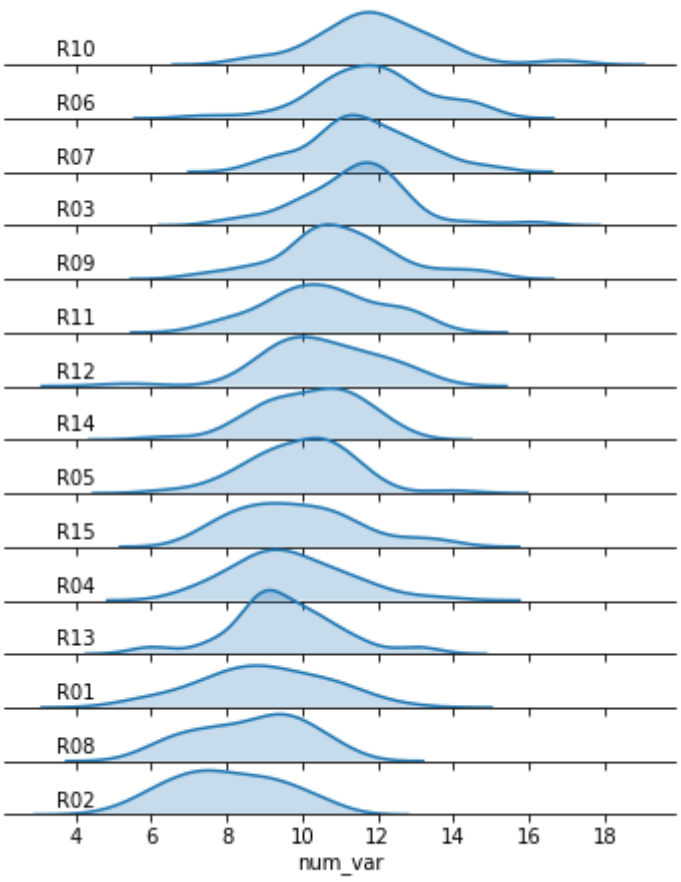
The individual subplots now overlap, but we've still got a problem: the backgrounds of the subplots are opaque, thus obscuring all but the tops of all of the individual group distributions, with the exception of the lowest. In addition, the individual subplot titles overlap the other distributions with some ambiguity: these should be moved elsewhere in the individual plots. The revised code and plot look like this:

```
group_means = df.groupby(['many_cat_var']).mean()
group_order = group_means.sort_values(['num_var'], ascending = False).index

g = sb.FacetGrid(data = df, row = 'many_cat_var', size = 0.5, aspect = 12,
                 row_order = group_order, gridspec_kws = {'hspace' : -0.2})
g.map(sb.kdeplot, 'num_var', shade = True)

g.set(yticks=[])
g.despine(left=True)

# set the transparency of each subplot to full
g.map(lambda **kwargs: plt.gca().patch.set_alpha(0))

# remove subplot titles and write in new labels
def label_text(x, **kwargs):
    plt.text(4, 0.02, x.iloc[0], ha = 'center', va = 'bottom')
g.map(label_text, 'many_cat_var')
g.set_xlabels('num_var')
g.set_titles('')
```

We make clever use of the FacetGrid object's `map` function to perform the plot modifications. Previously, you've seen `map` used where the first argument is a plotting function, the following arguments are positional variable strings, and any additional arguments are keyword arguments for the plotting function. In actuality, you can set any function as the first argument, which will be applied to each facet. To apply the transparency using `map`, I set up an anonymous lambda function that gets the current Axes (`gca`), selects its background (`patch`), and sets its transparency to full.

As for the second `map` argument, it sends a pandas Series to the function specified by the first argument. This Series is filtered to include only the column specified by the second `map` argument, with only the rows appropriate for each facet. In this case, I exploit the fact that the 'many_cat_var' column is filled with copies of the categorical feature string to specify the `text` string, with hardcoded positional values appropriate to the plot. (`map` also sends a few general keyword arguments like 'color' automatically to the specified function, hence the need for `**kwargs` to capture them despite not specifying any myself.) One downside to this approach is that the x-axis labels get replaced with 'many_cat_var' after the `map` call, thus requiring the addition of a `set_xlabels` function call to reset the string.

The final ridgeline plot looks like this, where we can see the distribution of our numeric variable on each category, sorted by mean:



Further Reading

- Seaborn: ridge plot example - I actually used this example to clean up my initial attempts. It's got a little bit more aesthetic cleaning than the above demonstration.
