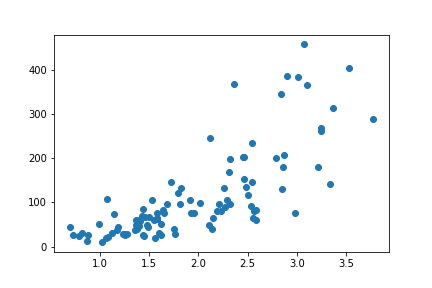
# Scatterplots

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. One basic way of creating a scatterplot is through Matplotlib's [**scatter**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html) function:

plt.scatter(data = df, x = 'num\_var1', y = 'num\_var2')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)**

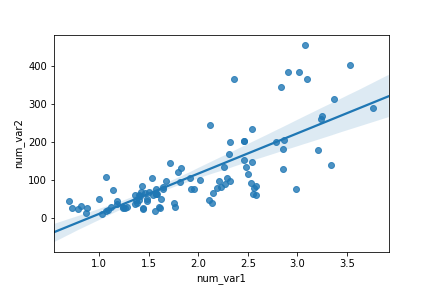
We can see a generally positive relationship between the two variables, as higher values of the x-axis variable are associated with greatly increasing values of the variable plotted on the y-axis.

**Alternative Approach**

Seaborn's **[regplot](https://seaborn.pydata.org/generated/seaborn.regplot.html" \t "_blank)** function combines scatterplot creation with regression function fitting:

sb.regplot(data = df, x = 'num\_var1', y = 'num\_var2')

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

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)**

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 \text{log}(y) \propto xlog(*y*)∝*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 reg\_fit = False in the regplot function call. Otherwise, if we want to plot the regression line on the observed relationship in the data, we need to transform the data, as seen in the previous lesson.

**def** **log\_trans**(x, inverse = False):

**if** **not** inverse:

**return** np.log10(x)

**else**:

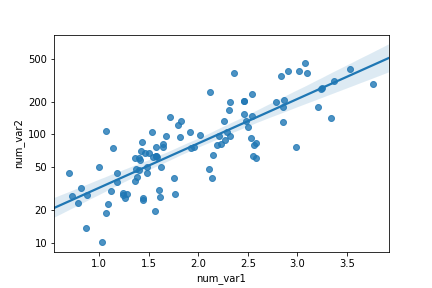
**return** np.power(10, x)

sb.regplot(df['num\_var1'], df['num\_var2'].apply(log\_trans))

tick\_locs = [10, 20, 50, 100, 200, 500]

plt.yticks(log\_trans(tick\_locs), tick\_locs)

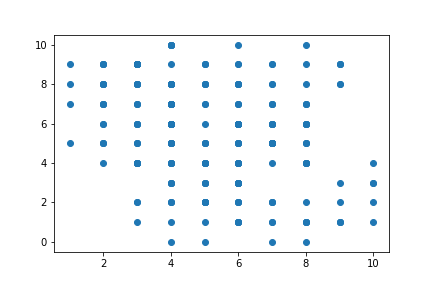
In this example, the x- and y- values sent to regplot are set directly as Series, extracted from the dataframe.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9d1316b3-f339-4d52-b63f-91994aefdd40)**

**Overplotting, Transparency, and Jitter**

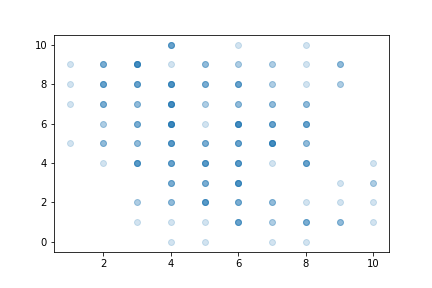
If we have a very large number of points to plot or our numeric variables are discrete-valued, then it is possible that using a scatterplot straightforwardly will not be informative. The visualization will suffer from *overplotting*, where the high amount of overlap in points makes it difficult to see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

plt.scatter(data = df, x = 'disc\_var1', y = 'disc\_var2')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)**

In the above plot, we can infer some kind of negative relationship between the two variables, but 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. Transparency can be added to a [**scatter**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html) call by adding the "alpha" parameter set to a value between 0 (fully transparent, not visible) and 1 (fully opaque).

plt.scatter(data = df, x = 'disc\_var1', y = 'disc\_var2', alpha = 1/5)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)**

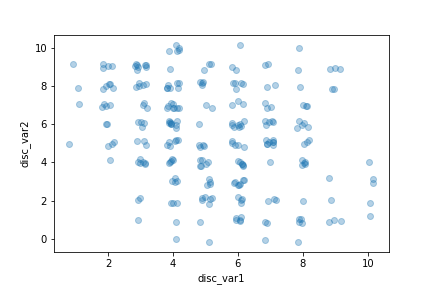
Where more points overlap, the darker the image will be. Here, we can now see that there is a moderate negative relationship between the two numeric variables. Values of 0 and 10 on the x-axis are much rarer than the central values.

As an alternative or companion to transparency, we can also add jitter to move the position of each point slightly from its true value. This is not a direct option in matplotlib's scatter function, but is a built-in option with seaborn's **[regplot](https://seaborn.pydata.org/generated/seaborn.regplot.html" \t "_blank)** function. x- and y- jitter can be added independently, and won't affect the fit of any regression function, if made:

sb.regplot(data = df, x = 'disc\_var1', y = 'disc\_var2', fit\_reg = **False**,

x\_jitter = 0.2, y\_jitter = 0.2, scatter\_kws = {'alpha' : 1/3})

The jitter settings will cause each point to be plotted in a uniform ±0.2 range of their true values. Note that transparency has been changed to be a dictionary assigned to the "scatter\_kws" parameter. This is necessary so that transparency is specifically associated with the scatter component of the regplot function.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/bd599deb-e989-4ce2-9dc2-d62316ecc0f1)**

# 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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist2d.html) function.

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

*# left plot: scatterplot of discrete data with jitter and transparency*

plt.subplot(1, 2, 1)

sb.regplot(data = df, x = 'disc\_var1', y = 'disc\_var2', fit\_reg = **False**,

x\_jitter = 0.2, y\_jitter = 0.2, scatter\_kws = {'alpha' : 1/3})

*# right plot: heat map with bin edges between values*

plt.subplot(1, 2, 2)

bins\_x = np.arange(0.5, 10.5+1, 1)

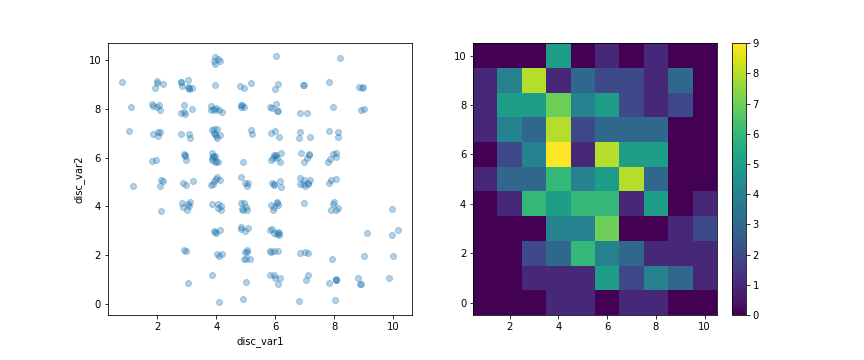
bins\_y = np.arange(-0.5, 10.5+1, 1)

plt.hist2d(data = df, x = 'disc\_var1', y = 'disc\_var2',

bins = [bins\_x, bins\_y])

plt.colorbar();

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. We add a **[colorbar](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.colorbar.html" \t "_blank)** function call to add a colorbar to the side of the plot, showing the mapping

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)**

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

Heatmaps can also be used as 2-d versions of bar charts, plotting counts on combinations of two categorical variables instead of numeric variables. There is a function in seaborn, heatmap, that is built for categorical heatmaps. This will be discussed in more detail later in the lesson, in the section "Clustered Bar Charts".

**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**](https://matplotlib.org/api/pyplot_summary.html#colors-in-matplotlib) 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.

bins\_x = np.arange(0.5, 10.5+1, 1)

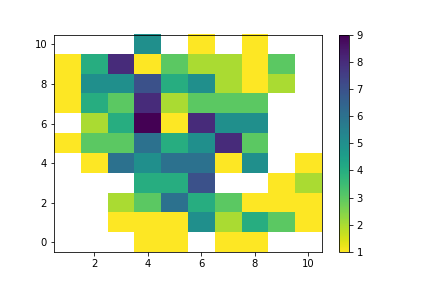
bins\_y = np.arange(-0.5, 10.5+1, 1)

plt.hist2d(data = df, x = 'disc\_var1', y = 'disc\_var2',

bins = [bins\_x, bins\_y], cmap = 'viridis\_r', cmin = 0.5)

plt.colorbar()

l00

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)**

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.

*# hist2d returns a number of different variables, including an array of counts*

bins\_x = np.arange(0.5, 10.5+1, 1)

bins\_y = np.arange(-0.5, 10.5+1, 1)

h2d = plt.hist2d(data = df, x = 'disc\_var1', y = 'disc\_var2',

bins = [bins\_x, bins\_y], cmap = 'viridis\_r', cmin = 0.5)

counts = h2d[0]

*# 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 >= 7: *# increase visibility on darkest cells*

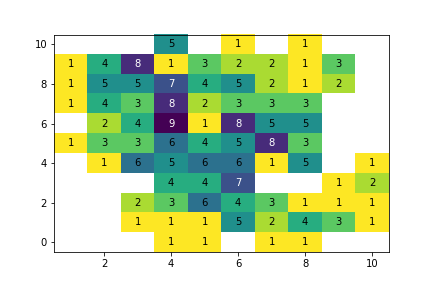
plt.text(bins\_x[i]+0.5, bins\_y[j]+0.5, int(c),

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')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/8f54c142-468e-4e1a-abef-4c6425403a3b)**

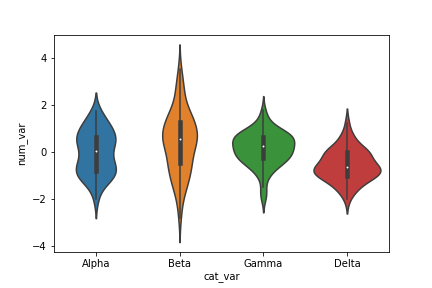
If you have too many cells in your heat map, then the annotations will end up being too overwhelming, too much to attend to. In cases like that, it's best to leave off the annotations and let the data and colorbar speak for themselves. You're more likely to see annotations in a categorical heat map, where there are going to be fewer cells plotted. Indeed, there is a parameter built into seaborn's heatmap function for just that, as will be seen later.

# 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](https://seaborn.pydata.org/generated/seaborn.violinplot.html" \t "_blank)** function can be used to create violin plots combined with box plots – we'll discuss box plots on the next page.

sb.violinplot(data = df, x = 'cat\_var', y = 'num\_var')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)**

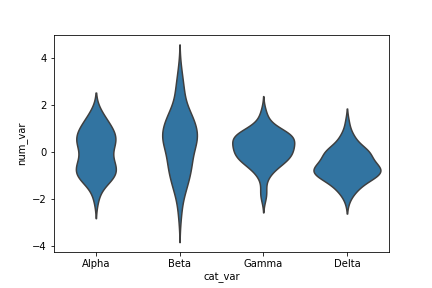
Here, you can see that the numeric data takes on a different shape in each categorical level: Some bimodality is suggested in group Alpha, a relatively high variance is observed in Beta, and Gamma and Delta are skewed negatively and positively, respectively. You can also see that each level has been 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. This is the miniature box plot mentioned above. 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.

base\_color = sb.color\_palette()[0]

sb.violinplot(data = df, x = 'cat\_var', y = 'num\_var', color = base\_color,

inner = **None**)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)**

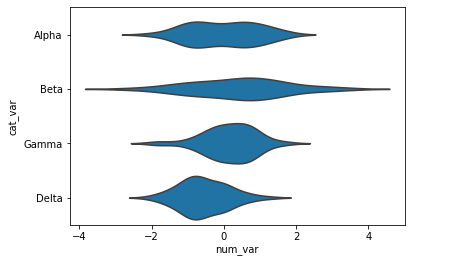
**Additional Variation**

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.

base\_color = sb.color\_palette()[0]

sb.violinplot(data = df, x = 'num\_var', y = 'cat\_var', color = base\_color,

inner = **None**)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/1dca3bcb-d2d4-44c2-b88e-ecea98402349)**

# 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 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**](https://seaborn.pydata.org/generated/seaborn.boxplot.html) function.

plt.figure(figsize = [10, 5])

base\_color = sb.color\_palette()[0]

*# left plot: violin plot*

plt.subplot(1, 2, 1)

ax1 = sb.violinplot(data = df, x = 'cat\_var', y = 'num\_var', color = base\_color)

*# right plot: box plot*

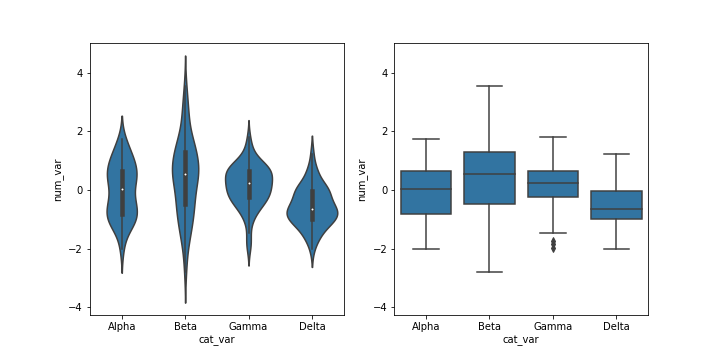
plt.subplot(1, 2, 2)

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*

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**](https://matplotlib.org/api/axes_api.html))

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)**

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.

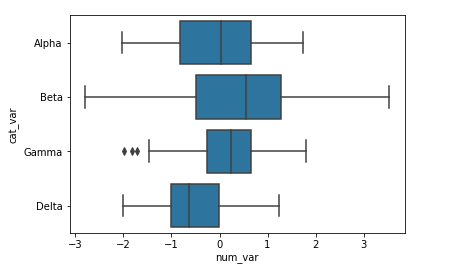
Comparing the two plots, the box plot is a cleaner summary of the data than the violin plot. It's easier to compare statistics between the groups with a box plot. This makes a box plot worth more consideration if you have a lot of groups to compare, or if you are building explanatory plots. You can clearly 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.

base\_color = sb.color\_palette()[0]

sb.boxplot(data = df, x = 'num\_var', y = 'cat\_var', color = base\_color)

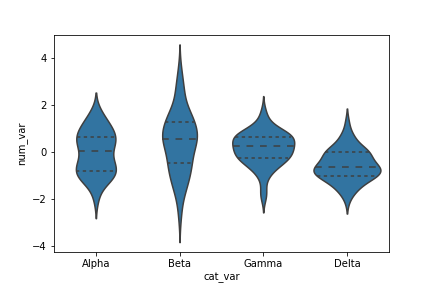
**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)**

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.

base\_color = sb.color\_palette()[0]

sb.violinplot(data = df, x = 'cat\_var', y = 'num\_var', color = base\_color,

inner = 'quartile')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/6ec37bb1-2d14-451c-9965-91c86be93be3)**

# Pandas Categorical D Types:

car\_sizes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize Cars', 'Large Cars']

v\_sizes = pd.api.types.CategoricalDtype(ordered = True, categories =car\_sizes )

fuel\_econ['VClass'] = fuel\_econ['VClass'].astype(v\_sizes);

# Rotating X labels:

base\_color = sb.color\_palette()[1]

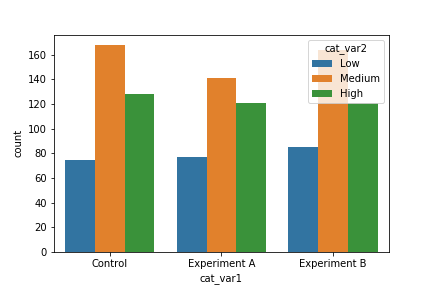
g = sb.violinplot(data=fuel\_econ, x = 'VClass',y ='displ', color = base\_color)

g.set\_xticklabels(labels = car\_sizes, rotation=15);

# 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](https://seaborn.pydata.org/generated/seaborn.countplot.html" \t "_blank)** function. To take the plot from univariate to bivariate, we add the second variable to be plotted under the "hue" argument:

sb.countplot(data = df, x = 'cat\_var1', hue = 'cat\_var2')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)**

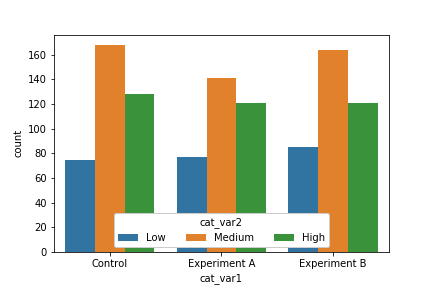
The first categorical variable is depicted by broad x-position (Control, Experiment A, Experiment B). Within each of these groups, three bars are plotted, one for each level of the second categorical variable (Low, Medium, High). Color differentiates each level, and is documented with the legend in the upper-right corner of the plot. The plot tells us that the three "cat\_var1" groups are fairly balanced in frequencies across the "cat\_var2" levels, though the "Experiment A" group appears to have slighly lower counts of "Medium" points (orange central bar) compared to the other two groups.

The legend position in this example is a bit distracting, however. We can use an [**Axes method**](https://matplotlib.org/api/axes_api.html) to set the legend properties on the Axes object returned from countplot.

ax = sb.countplot(data = df, x = 'cat\_var1', hue = 'cat\_var2')

ax.legend(loc = 8, ncol = 3, framealpha = 1, title = 'cat\_var2')

(Documentation: **[Axes.legend](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.legend.html" \t "_blank)**)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)**

**Alternative Approach (Heat Map)**

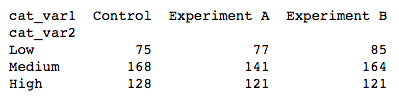
One alternative way of depicting the relationship between two categorical variables is through a heat map. Heat maps were introduced earlier as the 2-d version of a histogram; here, we're using them as the 2-d version of a bar chart. The seaborn function [**heatmap**](https://seaborn.pydata.org/generated/seaborn.heatmap.html) 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.

ct\_counts = df.groupby(['cat\_var1', 'cat\_var2']).size()

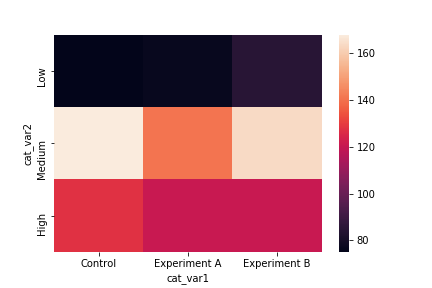
ct\_counts = ct\_counts.reset\_index(name = 'count')

ct\_counts = ct\_counts.pivot(index = 'cat\_var2', columns = 'cat\_var1', values = 'count')

(Documentation: [**Series reset\_index**](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reset_index.html), **[DataFrame pivot](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.pivot.html" \t "_blank)**)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)**

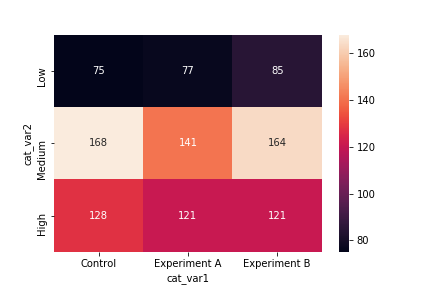
sb.heatmap(ct\_counts)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)**

The heat map tells the same story as the clustered bar chart: the similar colors across the rows suggests that the "cat\_var1" group sizes are similarly sized and distributed similarly across levels of "cat\_var2". The Experiment A has slightly fewer counts of Medium-level observations, while Control has higher counts of High data points and Experiment B has slightly higher counts of Low points. Compared to the clustered bar chart, however, there is less precision interpreting the magnitude of differences. For this reason, we might want to add annotations to the plot to report counts within each cell.

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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/7c2f7055-36bc-4370-b81e-68a897951353)**

# Faceting

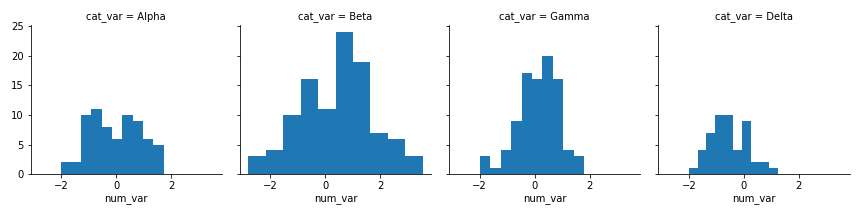
One general visualization technique that will be useful for you to know about to handle plots of two or more variables is **faceting**. In faceting, the data is divided into disjoint subsets, most often by different levels of a categorical variable. For each of these subsets of the data, the same plot 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](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html" \t "_blank)** 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 ("cat\_var" 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, histogram on "num\_var").

g = sb.FacetGrid(data = df, col = 'cat\_var')

g.map(plt.hist, "num\_var")

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 = "num\_var", or the mapping won't work properly.

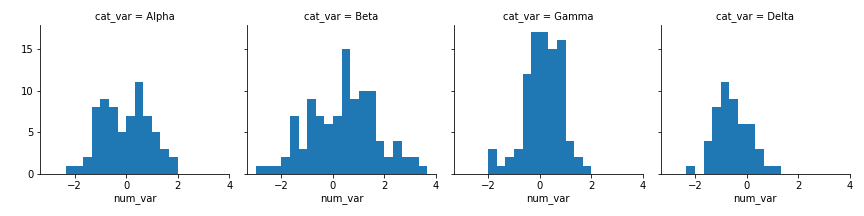
**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)**

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.

bin\_edges = np.arange(-3, df['num\_var'].max()+1/3, 1/3)

g = sb.FacetGrid(data = df, col = 'cat\_var')

g.map(plt.hist, "num\_var", bins = bin\_edges)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)**

**Additional Variation**

If you have many categorical levels to plot, then you might want to add more arguments to the FacetGrid object initialization to facilitate clarity in conveyance of information. The example below includes a categorical variable, "many\_cat\_var", that has fifteen different levels. Setting col\_wrap = 5 means that the plots will be organized into rows of five facets each, rather than a single long row of fifteen plots.

group\_means = df.groupby(['many\_cat\_var']).mean()

group\_order = group\_means.sort\_values(['num\_var'], ascending = **False**).index

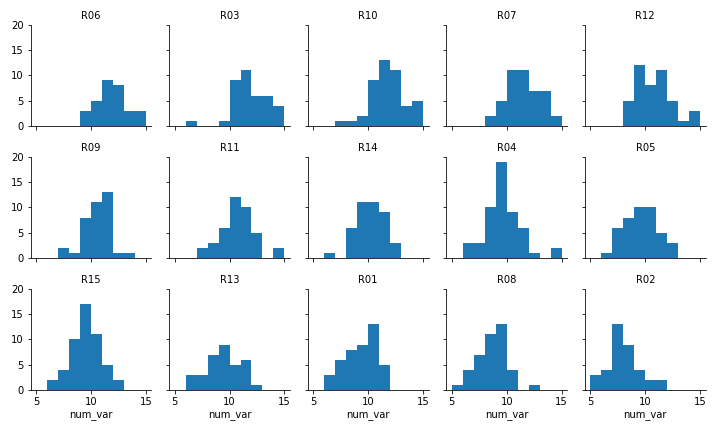
g = sb.FacetGrid(data = df, col = 'many\_cat\_var', col\_wrap = 5, size = 2,

col\_order = group\_order)

g.map(plt.hist, 'num\_var', bins = np.arange(5, 15+1, 1))

g.set\_titles('{col\_name}')

Other operations may be performed to increase the immediate readability of the plots: setting each facet height to 2 inches ("size"), sorting the facets by group mean ("col\_order"), limiting the number of bin edges, and changing the titles of each facet to just the categorical level name using the set\_titles method and {col\_name} template variable.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/2159eb35-129b-419e-a81e-258425d98e1c)**

# Adapted Bar Charts

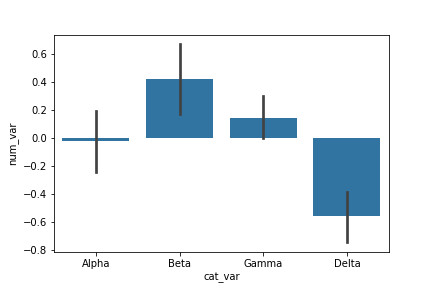
Histograms and bar charts were introduced in the previous lesson as depicting the distribution of numeric and categorical variables, respectively, with the height (or length) of bars indicating the number of data points that fell within each bar's range of values. These plots can be adapted for use as 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](https://seaborn.pydata.org/generated/seaborn.barplot.html" \t "_blank)** function:

base\_color = sb.color\_palette()[0]

sb.barplot(data = df, x = 'cat\_var', y = 'num\_var', color = base\_color)

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

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)**

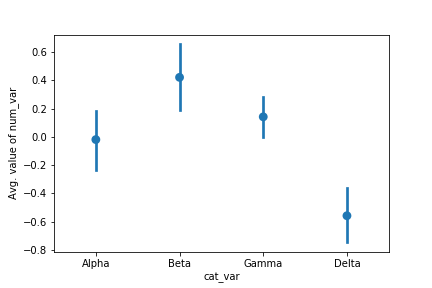
The bar heights indicate the mean value on the numeric variable, with error bars plotted to show the uncertainty in the mean based on variance and sample size. The Delta bar dips below the 0 axis due to the negative mean.

As an alternative, the **[pointplot](https://seaborn.pydata.org/generated/seaborn.pointplot.html" \t "_blank)** 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.

sb.pointplot(data = df, x = 'cat\_var', y = 'num\_var', linestyles = "")

plt.ylabel('Avg. value of num\_var')

By default, pointplotwill 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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)**

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.

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

base\_color = sb.color\_palette()[0]

*# left plot: violin plot*

plt.subplot(1, 3, 1)

sb.violinplot(data = df, x = 'condition', y = 'binary\_out', inner = **None**,

color = base\_color)

plt.xticks(rotation = 10) *# include label rotation due to small subplot size*

*# center plot: box plot*

plt.subplot(1, 3, 2)

sb.boxplot(data = df, x = 'condition', y = 'binary\_out', color = base\_color)

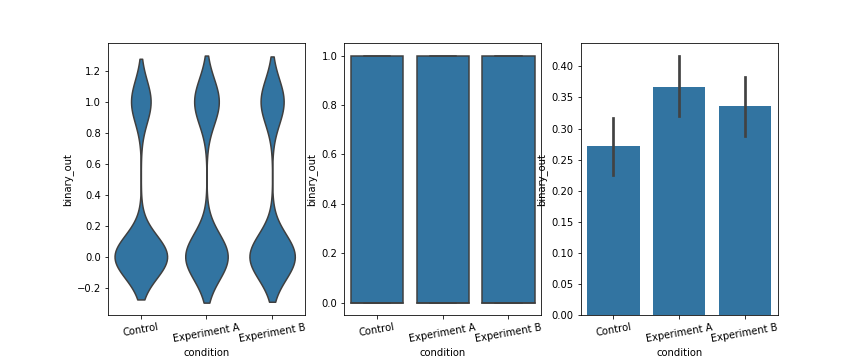
plt.xticks(rotation = 10)

*# right plot: adapted bar chart*

plt.subplot(1, 3, 3)

sb.barplot(data = df, x = 'condition', y = 'binary\_out', color = base\_color)

plt.xticks(rotation = 10)

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)**

**Adapted Histograms**

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" parameter. By default, each data point is given a weight of 1, so that the sum of point weights in each bin is equal to the number of points. If we change the weights to be a representative function of each point's value on a second variable, then the sum will end up representing something other than a count.

bin\_edges = np.arange(0, df['num\_var'].max()+1/3, 1/3)

*# count number of points in each bin*

bin\_idxs = pd.cut(df['num\_var'], bin\_edges, right = **False**, include\_lowest = **True**,

labels = **False**).astype(int)

pts\_per\_bin = df.groupby(bin\_idxs).size()

num\_var\_wts = df['binary\_out'] / pts\_per\_bin[bin\_idxs].values

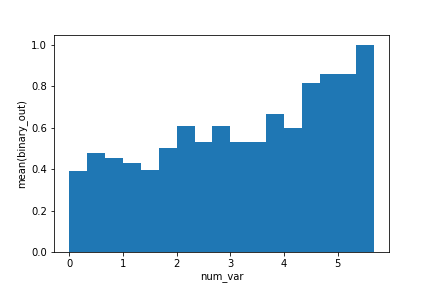
*# plot the data using the calculated weights*

plt.hist(data = df, x = 'num\_var', bins = bin\_edges, weights = num\_var\_wts)

plt.xlabel('num\_var')

plt.ylabel('mean(binary\_out)')

To get the mean of the y-variable ("binary\_out") in each bin, the weight of each point should be equal to the y-variable value, divided by the number of points in its x-bin (num\_var\_wts). As part of this computation, we make use of pandas' [**cut**](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.cut.html) function in order to associate each data point to a particular bin (bin\_idxs). The labels = False parameter means that each point's bin membership is associated by a numeric index, rather than a string. We use these numeric indices into the pts\_per\_bin, with the .values at the end necessary in order for the Series' indices to not be confused between the indices of df['binary\_out'].

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/c2627efd-b028-430f-b3f8-de1d17e9db89)**

This plot shows that the average outcome of the y-variable "binary\_out" generally increases across values of the x-variable "num\_var".

More commonly, you'll see plots summarizing one numeric variable against values of a second numeric variable not as adapted histograms, but instead as line plots. These are covered on the next page.

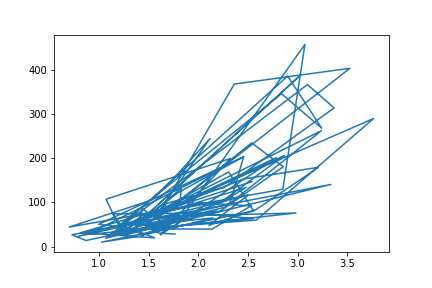
# Line Plots

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. Often, we have only one observation per time period, like in stock or currency charts. While there is a seaborn function **[tsplot](https://seaborn.pydata.org/generated/seaborn.tsplot.html" \t "_blank)** that is intended to be used with time series data, it is fairly specialized and (as of this writing's seaborn 0.8) is slated for major changes.

Instead, we will make use of Matplotlib's **[errorbar](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html" \t "_blank)** function, performing some processing on the data in order to get it into its necessary form.

plt.errorbar(data = df, x = 'num\_var1', y = 'num\_var2')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)**

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.

*# set bin edges, compute centers*

bin\_size = 0.25

xbin\_edges = np.arange(0.5, df['num\_var1'].max()+bin\_size, bin\_size)

xbin\_centers = (xbin\_edges + bin\_size/2)[:-1]

*# compute statistics in each bin*

data\_xbins = pd.cut(df['num\_var1'], xbin\_edges, right = **False**, include\_lowest = **True**)

y\_means = df['num\_var2'].groupby(data\_xbins).mean()

y\_sems = df['num\_var2'].groupby(data\_xbins).sem()

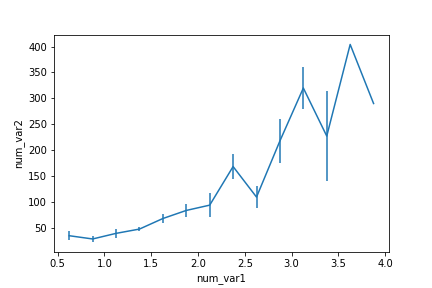
*# plot the summarized data*

plt.errorbar(x = xbin\_centers, y = y\_means, yerr = y\_sems)

plt.xlabel('num\_var1')

plt.ylabel('num\_var2')

Since the x-variable ('num\_var1') 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. Note that the [**cut**](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.cut.html) function call is simpler here than in the previous page, since we don't need to compute individual point weights.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)**

An interesting part of the above summarization of the data is that the uncertainty in the mean generally increases with increasing x-values. But for the largest two points, there are no error bars. Looking at the default errorbar plot (or the scatterplot below), we can see this is due to there only being one point in each of the last two bins.

**Alternate Variations**

Instead of computing summary statistics on fixed bins, you can also make computations on a rolling window through use of pandas' [**rolling**](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rolling.html) method. Since the rolling window will make computations on sequential rows of the dataframe, we should use **[sort\_values](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sort_values.html" \t "_blank)** to put the x-values in ascending order first.

*# 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)

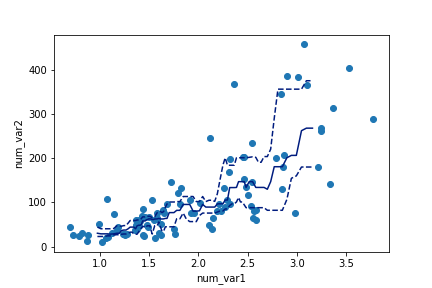
plt.errorbar(x = x\_winmean, y = y\_q1, c = line\_color, linestyle = '--')

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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)**

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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html) calls, setting the "histtype = step" parameter so that the bars are depicted as unfilled lines.

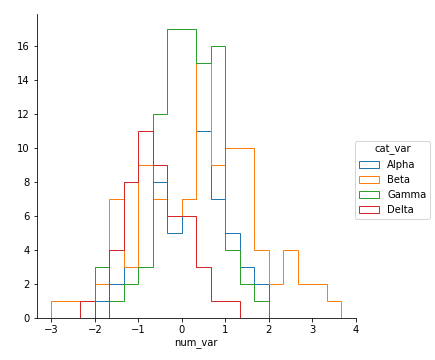
bin\_edges = np.arange(-3, df['num\_var'].max()+1/3, 1/3)

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](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html" \t "_blank)**, 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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)**

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.

**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)

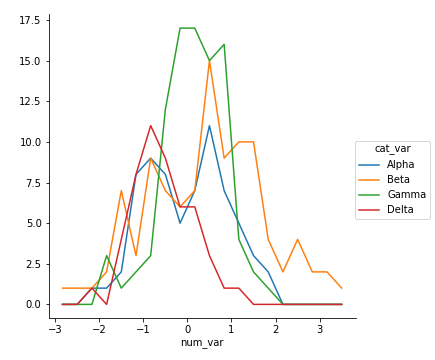
g = sb.FacetGrid(data = df, hue = 'cat\_var', size = 5)

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**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html))

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/95f3c6d7-2ad0-43c1-bb53-4b7fbc40262b)**

# 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

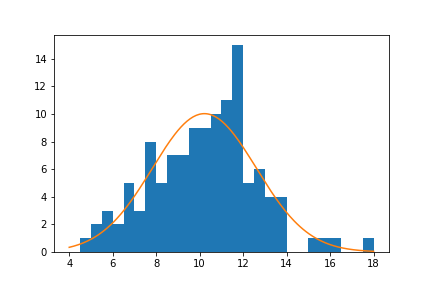
x = np.linspace(4, 18, 200)

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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html) 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](https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.linspace.html" \t "_blank)** 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**](https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.norm.html) 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:

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)**

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](https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.shapiro.html" \t "_blank)** 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*(0.25)=−0.674, ppf(0.5) = 0*ppf*(0.5)=0, and ppf(0.75) = 0.674*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}*n*+1−2*ak*−*a*​, 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}*nk*−0.5​. 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())

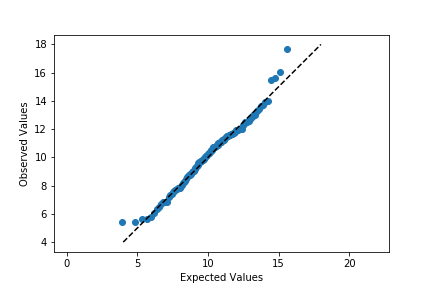
plt.plot([4,18],[4,18],'--', color = 'black')

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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)**

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

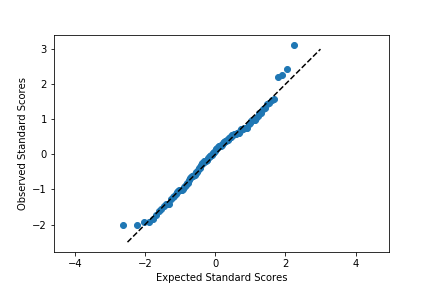
plt.scatter(expected\_scores, data\_scores)

plt.plot([-2.5,3],[-2.5,3],'--', color = 'black')

plt.axis('equal')

plt.xlabel('Expected Standard Scores')

plt.ylabel('Observed Standard Scores')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)**

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

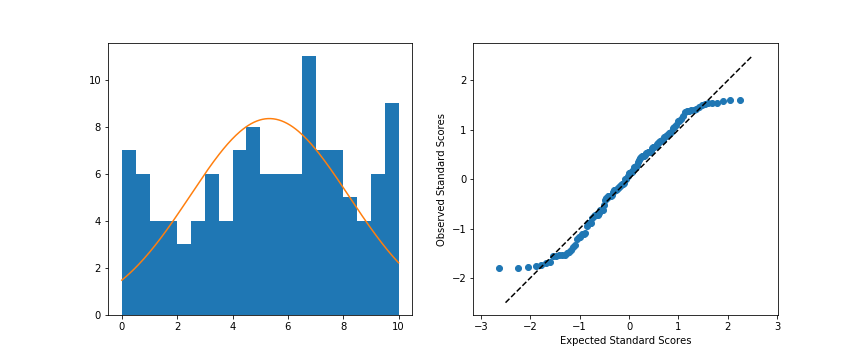
plt.scatter(expected\_scores, data\_scores)

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')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)**

**[Left: Original data; Right: Q-Q plot](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/cb344c1a-dc89-4866-9d2b-841fe590d5e4)**

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.

# Extra 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](https://seaborn.pydata.org/generated/seaborn.swarmplot.html" \t "_blank)** 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)

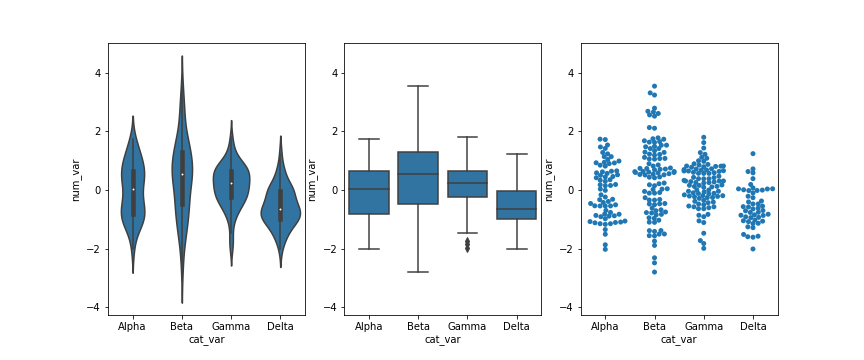
plt.ylim(ax1.get\_ylim()) *# set y-axis limits to be same as left plot*

*# 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*

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/37aa4a54-ea6b-441c-b2e1-b2d996bdff97)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/37aa4a54-ea6b-441c-b2e1-b2d996bdff97)**

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.

# Extra 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.

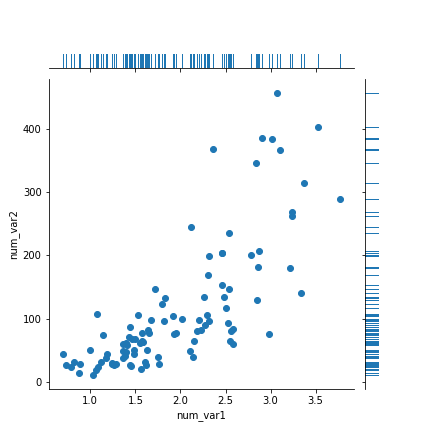
Seaborn's **[JointGrid](https://seaborn.pydata.org/generated/seaborn.JointGrid.html" \t "_blank)** class enables this plotting of bivariate relationship with marginal univariate plots for numeric data. The plot\_joint method specifies a plotting function for the main, joint plot for the two variables, while the plot\_marginals method specifies the plotting function for the two marginal plots. Here, we make use of seaborn's **[rugplot](https://seaborn.pydata.org/generated/seaborn.rugplot.html" \t "_blank)** 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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/a5e5b3af-fe44-40cf-aadf-3ee5c0630eaf)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/a5e5b3af-fe44-40cf-aadf-3ee5c0630eaf)**

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](https://seaborn.pydata.org/generated/seaborn.swarmplot.html" \t "_blank)** 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',

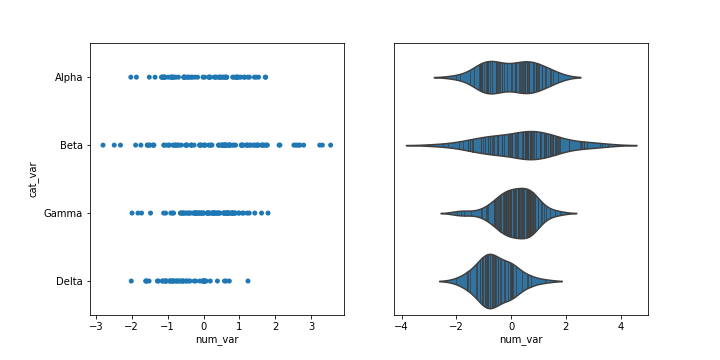
color = base\_color)

*# 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')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/a5e5b3af-fe44-40cf-aadf-3ee5c0630eaf)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/a5e5b3af-fe44-40cf-aadf-3ee5c0630eaf)**

# Extra 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()

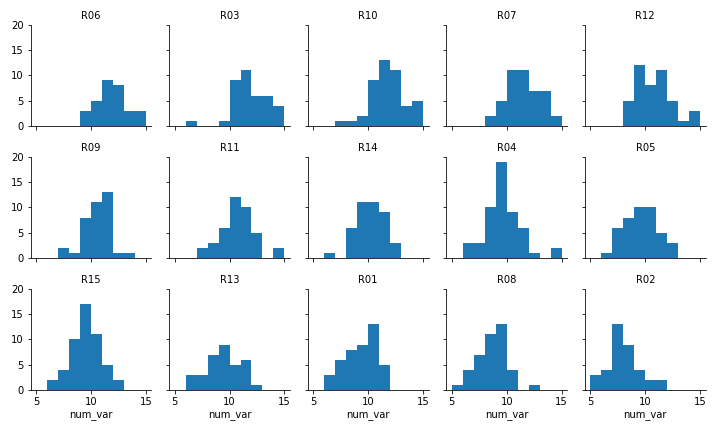
group\_order = group\_means.sort\_values(['num\_var'], ascending = **False**).index

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}')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)**

**[Faceted plot, whose data will be converted into a ridgeline form](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)**

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

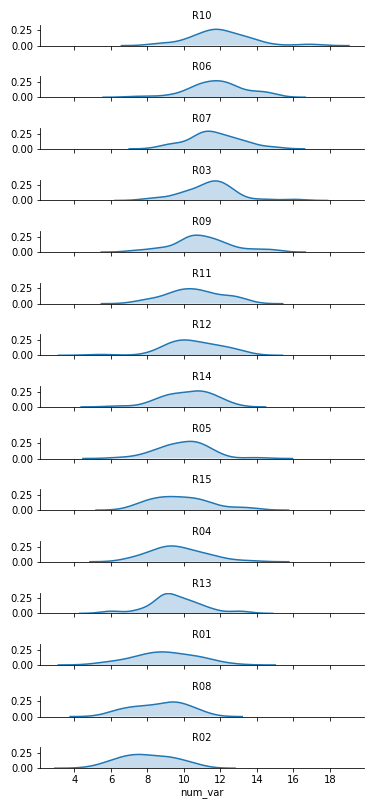
g = sb.FacetGrid(data = df, row = 'many\_cat\_var', size = 0.75, aspect = 7,

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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)**

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**)

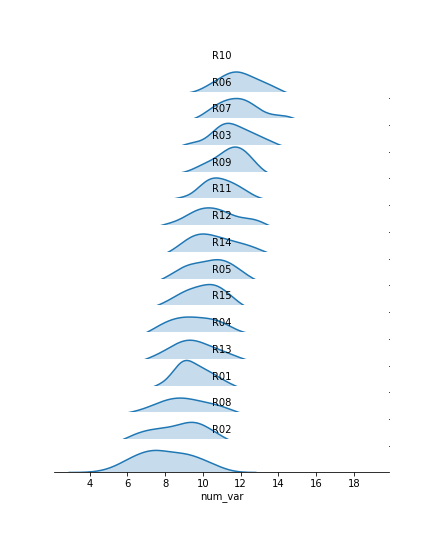
*# remove the y-axes*

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](https://matplotlib.org/api/_as_gen/matplotlib.gridspec.GridSpec.html" \t "_blank)** 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.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)**

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')

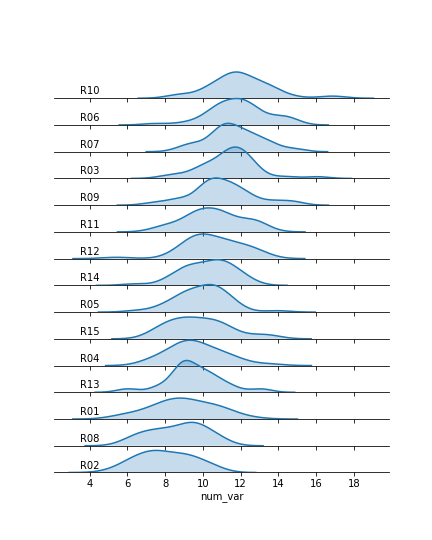
g.set\_xlabels('num\_var')

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](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html" \t "_blank)**), selects its background ([**patch**](https://matplotlib.org/api/_as_gen/matplotlib.patches.Patch.html)), 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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.text.html) 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:

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/9bc43497-5cbb-4ad3-8a68-37c145cfd2c0)**

## Further Reading

* Seaborn: [**ridge plot example**](https://seaborn.pydata.org/examples/kde_ridgeplot.html) - I actually used this example to clean up my initial attempts. It's got a little bit more aesthetic cleaning than the above demonstration.

# 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]):

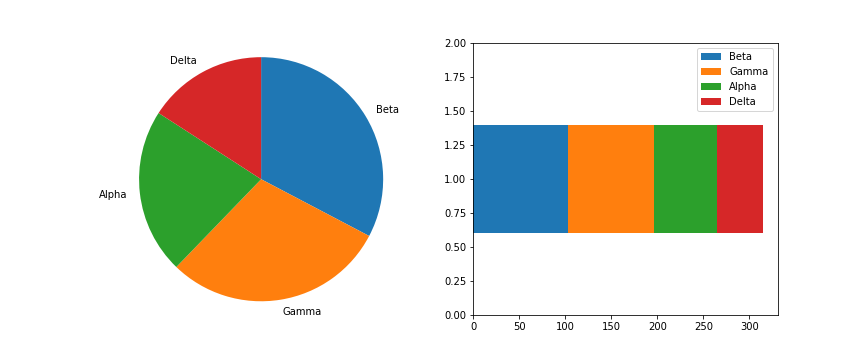
plt.barh(y = 1, width = sorted\_counts[i], left = baseline)

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](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html" \t "_blank)** 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!

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)**

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],

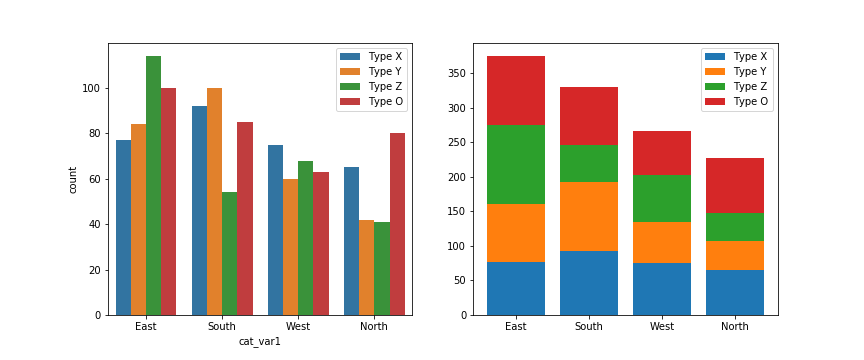
bottom = baselines)

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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html) 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!

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)**

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)

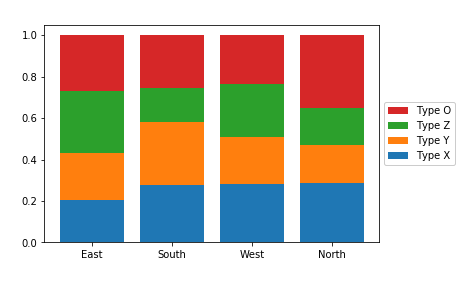
baselines += inner\_props[cat1\_order]

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**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html) function call, we make use of the built-in Python function [**reversed**](https://docs.python.org/3/library/functions.html#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".

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/8d08820f-00ab-4834-babd-cd33eae50104/concepts/28e18872-f2c7-4940-b9de-560c8c47a060)**

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**](https://eagereyes.org/techniques/stacked-bars-are-the-worst)
* Data Revelations: [**How to take the “screaming cats” out of stacked bar and area charts**](http://www.datarevelations.com/stackedbars.html)