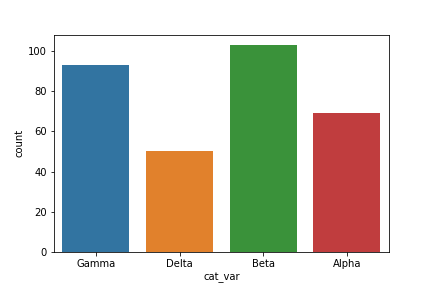
# Bar Charts

A **bar chart** is used to depict the distribution of a categorical variable. In a bar chart, each level of the categorical variable is depicted with a bar, whose height indicates the frequency of data points that take on that level. A basic bar chart of frequencies can be created through the use of seaborn's [countplot](https://seaborn.pydata.org/generated/seaborn.countplot.html" \t "_blank) function:

sb.countplot(data = df, x = 'cat\_var')

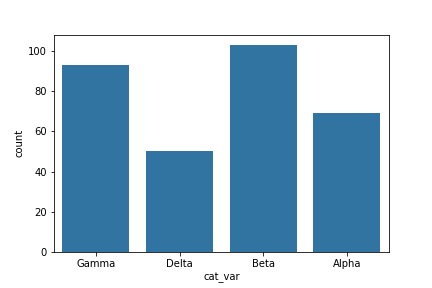
[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

For the example given, you can see that the Beta level has the highest frequency at over 100 counts, followed by Gamma and Alpha, with Delta the least frequent at around 50. By default, each category is given a different color. This might come in handy for building associations between these category labels and encodings in plots with more variables. Otherwise, it's a good idea to simplify the plot and reduce unnecessary distractions by plotting all bars in the same color. This can be set using the "color" parameter:

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

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

[color\_palette](https://seaborn.pydata.org/generated/seaborn.color_palette.html) returns a list of RGB tuples. Each tuple consists of three digits specifying the red, green, and blue channel values to specify a color. Calling this function without any parameters returns the current / default palette, and we take the first color to be the color for all bars.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

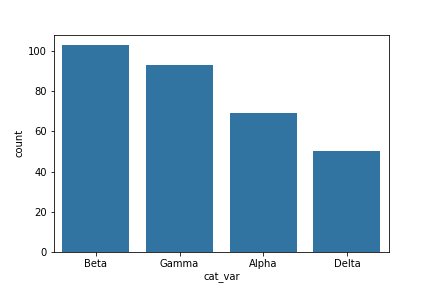
One thing that we might want to do with a bar chart is to sort the data in some way. For nominal-type data, one common operation is to sort the data in terms of frequency. With our data in a pandas DataFrame, we can use various DataFrame methods to compute and extract an ordering, then set that ordering on the "order" parameter:

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

cat\_order = df['cat\_var'].value\_counts().index

sb.countplot(data = df, x = 'cat\_var', color = base\_color, order = cat\_order)

(Documentation: [Series.value\_counts()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.value_counts.html" \t "_blank))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

For ordinal-type data, we probably want to sort the bars in order of the variables. While we could sort the levels by frequency like above, we usually care about whether the most frequent values are at high levels, low levels, etc. The best thing for us to do in this case is to convert the column into an ordered categorical data type. By default, pandas reads in string data as object types, and will plot the bars in the order in which the unique values were seen. By converting the data into an ordered type, the order of categories becomes innate to the feature, and we won't need to specify an "order" parameter each time it's required in a plot.

*# this method requires pandas v0.21 or later*

level\_order = ['Alpha', 'Beta', 'Gamma', 'Delta']

ordered\_cat = pd.api.types.CategoricalDtype(ordered = **True**, categories = level\_order)

df['cat\_var'] = df['cat\_var'].astype(ordered\_cat)

*# # use this method if you have pandas v0.20.3 or earlier*

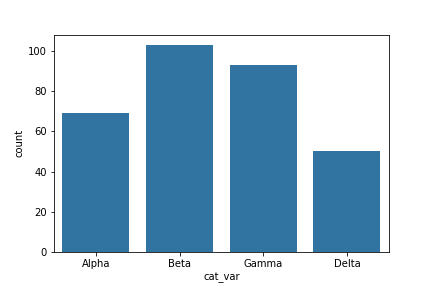
*# df['cat\_var'] = df['cat\_var'].astype('category', ordered = True,*

*# categories = level\_order)*

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

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

(Documentation: [CategoricalDtype](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.api.types.CategoricalDtype.html" \t "_blank))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

Should you find that you need to sort an ordered categorical type in a different order, you can always temporarily override the data type by setting the "order" parameter as above.

## Additional Variations

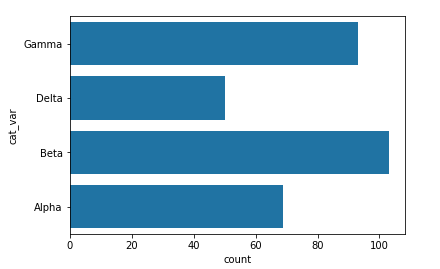
If your data is in a pandas Series, 1-d NumPy array, or list, you can also just set it as the first argument to the countplot function, as we do with the Series data\_var here:

sb.countplot(data\_var)

If you have a lot of category levels, or the category names are long, then you might end up with overcrowding of the tick labels. One way to address this is through creation of a horizontal bar chart. In a horizontal bar chart, it is the length of each bar that indicates frequency, rather than the height. In the code, instead of setting the data or variable on the "x" parameter, you can set the variable to be plotted on the parameter "y":

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

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

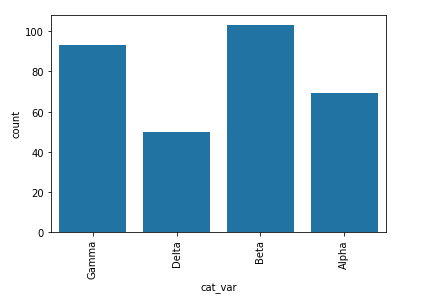
[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

Alternatively, you can use matplotlib's [xticks](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xticks.html" \t "_blank) function and its "rotation" parameter to change the orientation in which the labels will be depicted (as degrees counter-clockwise from horizontal):

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

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

plt.xticks(rotation = 90)

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/67ba40a3-42b6-4216-a100-19c5e565dfbe)

### Absolute vs. Relative Frequency

By default, seaborn's countplot function will summarize and plot the data in terms of **absolute frequency**, or pure counts. In certain cases, you might want to understand the distribution of data or want to compare levels in terms of proportions of the whole. In this case, you will want to plot the data in terms of **relative frequency**, where the height indicates the proportion of data taking each level, rather than the absolute count.

One method of plotting the data in terms of relative frequency on a bar chart is to just relabel the counts axis in terms of proportions. The underlying data will be the same, it will simply be the scale of the axis ticks that will be changed.

*# get proportion taken by most common group for derivation*

*# of tick marks*

n\_points = df.shape[0]

max\_count = df['cat\_var'].value\_counts().max()

max\_prop = max\_count / n\_points

*# generate tick mark locations and names*

tick\_props = np.arange(0, max\_prop, 0.05)

tick\_names = ['{:0.2f}'.format(v) **for** v **in** tick\_props]

*# create the plot*

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

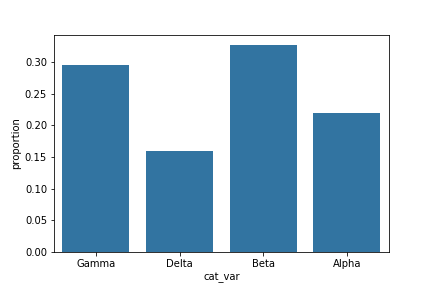
sb.countplot(data = df, x = 'cat\_var', color = base\_color)

plt.yticks(tick\_props \* n\_points, tick\_names)

plt.ylabel('proportion')

The [xticks](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xticks.html" \t "_blank) and [yticks](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.yticks.html" \t "_blank) functions aren't only about rotating the tick labels. You can also get and set their locations and labels as well. The first argument takes the tick locations: in this case, the tick proportions multiplied back to be on the scale of counts. The second argument takes the tick names: in this case, the tick proportions formatted as strings to two decimal places.

I've also added a [ylabel](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.ylabel.html" \t "_blank) call to make it clear that we're no longer working with straight counts.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/bd9948e9-eda5-48b7-8c6f-e97192a9ac48)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/bd9948e9-eda5-48b7-8c6f-e97192a9ac48)

## Additional Variation

Rather than plotting the data on a relative frequency scale, you might use text annotations to label the frequencies on bars instead. This requires writing a loop over the tick locations and labels and adding one text element for each bar.

*# create the plot*

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

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

*# add annotations*

n\_points = df.shape[0]

cat\_counts = df['cat\_var'].value\_counts()

locs, labels = plt.xticks() *# get the current tick locations and labels*

*# loop through each pair of locations and labels*

**for** loc, label **in** zip(locs, labels):

*# get the text property for the label to get the correct count*

count = cat\_counts[label.get\_text()]

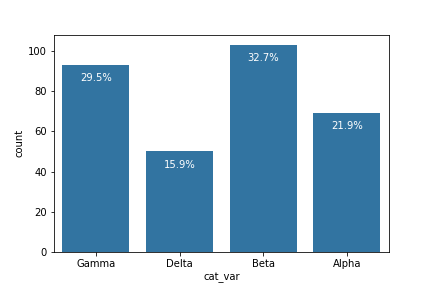
pct\_string = '{:0.1f}%'.format(100\*count/n\_points)

*# print the annotation just below the top of the bar*

plt.text(loc, count-8, pct\_string, ha = 'center', color = 'w')

I use the .get\_text() method to obtain the category name, so I can get the count of each category level. At the end, I use the [text](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.text.html#matplotlib.pyplot.text) function to print each percentage, with the x-position, y-position, and string as the three main parameters to the function.

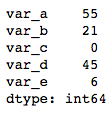
(Documentation: [Text objects](https://matplotlib.org/api/text_api.html?highlight=get_text#matplotlib.text.Text))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/bd9948e9-eda5-48b7-8c6f-e97192a9ac48)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/bd9948e9-eda5-48b7-8c6f-e97192a9ac48)

## Counting Missing Data

One interesting way we can apply bar charts is through the visualization of missing data. We can use pandas functions to create a table with the number of missing values in each column.

df.isna().sum()

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c33c9906-5be2-4c59-8012-eab668a50d3d)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c33c9906-5be2-4c59-8012-eab668a50d3d)

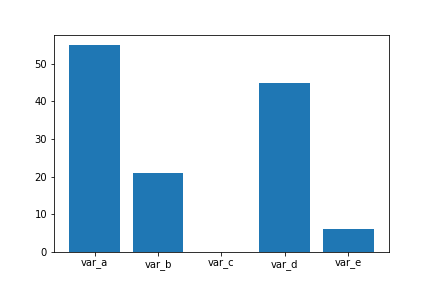
What if we want to visualize these missing value counts? We could treat the variable names as levels of a categorical variable, and create a resulting bar plot. However, since the data is not in its tidy, unsummarized form, we need to make use of a different plotting function. Seaborn's [barplot](https://seaborn.pydata.org/generated/seaborn.barplot.html" \t "_blank) function is built to depict a summary of one quantitative variable against levels of a second, qualitative variable, but can be used here.

na\_counts = df.isna().sum()

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

sb.barplot(na\_counts.index.values, na\_counts, color = base\_color)

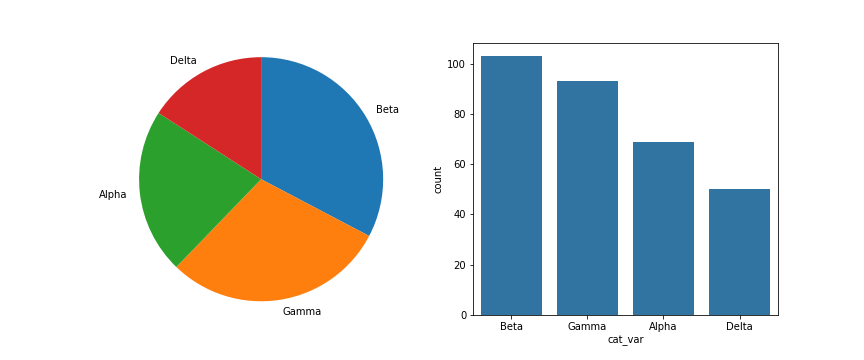
The first argument to the function contains the x-values (column names), the second argument the y-values (our counts).

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c33c9906-5be2-4c59-8012-eab668a50d3d)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c33c9906-5be2-4c59-8012-eab668a50d3d)

As a general note, this is a useful function to keep in mind if your data is summarized and you still want to build a bar chart. If your data is not yet summarized, however, just use the countplot function so that you don't need to do extra summarization work. In addition, you'll see what barplot's main purpose is in the next lesson, when we discuss adaptations of univariate plots for plotting bivariate data.

# Pie Charts

A **pie chart** is a common univariate plot type that is used to depict relative frequencies for levels of a categorical variable. Frequencies in a pie chart are depicted as wedges drawn on a circle: the larger the angle or area, the more common the categorical value taken.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/e7d15a6d-4d4b-418e-b85e-3cf0ed6f6740)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/e7d15a6d-4d4b-418e-b85e-3cf0ed6f6740)**

**[Pie chart (left) and bar chart (right) displaying the same categorical counts.](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/e7d15a6d-4d4b-418e-b85e-3cf0ed6f6740)**

Unfortunately, pie charts are a fairly limited plot type in the range of scenarios where they can be used, and it is easy for chart makers to try and spice up pie charts in a way that makes them more difficult to read. If you want to use a pie chart, try to follow certain guidelines:

* Make sure that your interest is in relative frequencies. Areas should represent parts of a whole, rather than measurements on a second variable (unless that second variable can logically be summed up into some whole).
* Limit the number of slices plotted. A pie chart works best with two or three slices, though it's also possible to plot with four or five slices as long as the wedge sizes can be distinguished. If you have a lot of categories, or categories that have small proportional representation, consider grouping them together so that fewer wedges are plotted, or use an 'Other' category to handle them.
* Plot the data systematically. One typical method of plotting a pie chart is to start from the top of the circle, then plot each categorical level clockwise from most frequent to least frequent. If you have three categories and are interested in the comparison of two of them, a common plotting method is to place the two categories of interest on either side of the 12 o'clock direction, with the third category filling in the remaining space at the bottom.

If these guidelines cannot be met, then you should probably make use of a bar chart instead. A bar chart is a safer choice in general. The bar heights are more precisely interpreted than areas or angles, and a bar chart can be displayed more compactly than a pie chart. There's also more flexibility with a bar chart for plotting variables with a lot of levels, like plotting the bars horizontally.

You can create a pie chart with matplotlib's [**pie**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.pie.html) function. This function requires that the data be in a summarized form: the primary argument to the function will be the wedge sizes.

*# code for the pie chart seen above*

sorted\_counts = df['cat\_var'].value\_counts()

plt.pie(sorted\_counts, labels = sorted\_counts.index, startangle = 90,

counterclock = **False**);

plt.axis('square')

To follow the guidelines in the bullet points above, I include the "startangle = 90" and "counterclock = False" arguments to start the first slice at vertically upwards, and will plot the sorted counts in a clockwise fashion. The [**axis**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.axis.html) function call and 'square' argument makes it so that the scaling of the plot is equal on both the x- and y-axes. Without this call, the pie could end up looking oval-shaped, rather than a circle.

## Additional Variation

A sister plot to the pie chart is the **donut plot**. It's just like a pie chart, except that there's a hole in the center of the plot. Perceptually, there's not much difference between a donut plot and a pie chart, and donut plots should be used with the same guidelines as a pie chart. Aesthetics might be one of the reasons why you would choose one or the other. For instance, you might see statistics reported in the hole of a donut plot to better make use of available space.

To create a donut plot, you can add a "wedgeprops" argument to the pie function call. By default, the radius of the pie (circle) is 1; setting the wedges' width property to less than 1 removes coloring from the center of the circle.

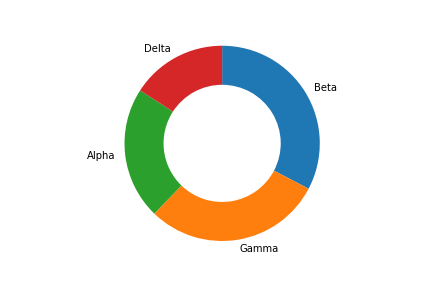
sorted\_counts = df['cat\_var'].value\_counts()

plt.pie(sorted\_counts, labels = sorted\_counts.index, startangle = 90,

counterclock = **False**, wedgeprops = {'width' : 0.4});

plt.axis('square')

(Documentation: [**Wedge patches**](https://matplotlib.org/api/_as_gen/matplotlib.patches.Wedge.html))

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/e7d15a6d-4d4b-418e-b85e-3cf0ed6f6740)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/e7d15a6d-4d4b-418e-b85e-3cf0ed6f6740)**

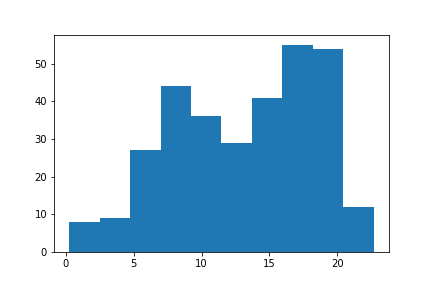
## Further Reading

* Eager Eyes: [**Understanding Pie Charts**](https://eagereyes.org/pie-charts)
* Eager Eyes: [**An Illustrated Tour of the Pie Chart Study Results**](https://eagereyes.org/blog/2016/an-illustrated-tour-of-the-pie-chart-study-results) - how accurately do people perceive different formulations of the pie chart?
* Datawrapper: [**What to Consider when Creating a Pie Chart**](https://academy.datawrapper.de/article/127-what-to-consider-when-creating-a-pie-chart)

# Histograms

A **histogram** is used to plot the distribution of a numeric variable. It's the quantitative version of the bar chart. However, rather than plot one bar for each unique numeric value, values are grouped into continuous bins, and one bar for each bin is plotted depicting the number. For instance, using the default settings for matplotlib's [**hist**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html) function:

plt.hist(data = df, x = 'num\_var')

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)**

You can see that there are eight data points that fall in the range between about 0 and 2.5 in the leftmost bin, and nine points in the range from about 2.5 to 5 in the adjacent bin. Overall, a generally bimodal distribution is observed (one with two peaks or humps). The direct adjacency of the bars in the histogram, in contrast to the separated bars in a bar chart, emphasize the fact that the data takes on a continuous range of values. When a data value is on a bin edge, it is counted in the bin to its right. The exception is the rightmost bin edge, which places data values equal to the uppermost limit into the right-most bin (to the upper limit's left).

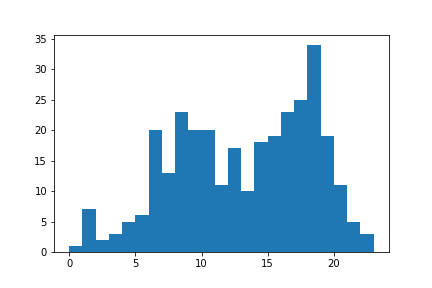
By default, the hist function divides the data into 10 bins, based on the range of values taken. In almost every case, we will want to change these settings. Usually, having only ten bins is too few to really understand the distribution of the data. And the default tick marks are often not on nice, 'round' values that make the ranges taken by each bin easy to interpret. Wouldn't it be better if I said "between 0 and 2.5" instead of "between about 0 and 2.5", and "from 2.5 to 5" instead of "from about 2.5 to 5" above?

You can use descriptive statistics (e.g. via [**df['num\_var'].describe()**](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.describe.html)) to gauge what minimum and maximum bin limits might be appropriate for the plot. These bin edges can be set using numpy's **[arange](https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html" \t "_blank)** function:

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

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

The first argument to arange is the leftmost bin edge, the second argument the upper limit, and the third argument the bin width. Note that even though I've specified the "max" value in the second argument, I've added a "+1" (the bin width). That is because arange will only return values that are strictly less than the upper limit. Adding in "+1" is a safety measure to ensure that the rightmost bin edge is at least the maximum data value, so that all of the data points are plotted. The leftmost bin is set as a hardcoded value to get a nice, interpretable value, though you could use functions like numpy's [**around**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.around.html) if you wanted to approach that end programmatically.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)**

When creating histograms, it's useful to play around with different bin widths to see what represents the data best. Too many bins, and you may see too much noise that interferes with identification of the underlying signal. Too few bins, and you may not be able to see the true signal in the first place.

plt.figure(figsize = [10, 5]) *# larger figure size for subplots*

*# histogram on left, example of too-large bin size*

plt.subplot(1, 2, 1) *# 1 row, 2 cols, subplot 1*

bin\_edges = np.arange(0, df['num\_var'].max()+4, 4)

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

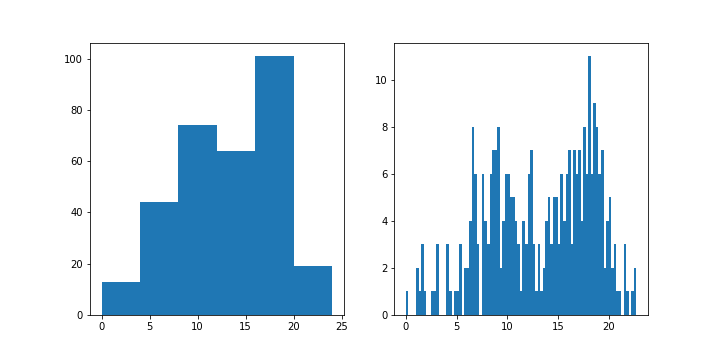
*# histogram on right, example of too-small bin size*

plt.subplot(1, 2, 2) *# 1 row, 2 cols, subplot 2*

bin\_edges = np.arange(0, df['num\_var'].max()+1/4, 1/4)

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

This example puts two plots side by side through use of the [**subplot**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html) function, whose arguments specify the number of rows, columns, and index of the active subplot (in that order). The [**figure()**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.figure.html) function is called with the "figsize" parameter so that we can have a larger figure to support having multiple subplots. (More details on figures and subplots are coming up next in the lesson.)

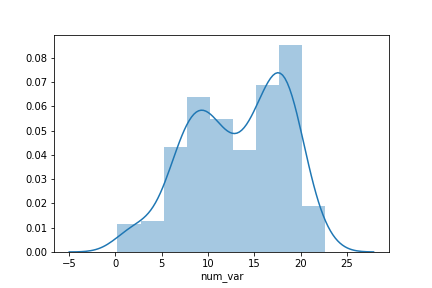
**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)**

## Alternative Approach

The seaborn function **[distplot](https://seaborn.pydata.org/generated/seaborn.distplot.html" \t "_blank)** can also be used to plot a histogram, and is integrated with other univariate plotting functions.

sb.distplot(df['num\_var'])

When we specify the data to be plotted, note that the first argument must be the Series or array with the points to be plotted. This is in contrast to our ability to specify a data source and column as separate arguments, like we've seen with and countplot and hist.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)**

The distplot function has built-in rules for specifying histogram bins, and by default plots a curve depicting the kernel density estimate (KDE) on top of the data. The vertical axis is based on the KDE, rather than the histogram: you shouldn't expect the total heights of the bars to equal 1, but the area under the curve should equal 1. If you want to learn more about KDEs, check out the extra page at the end of the lesson.

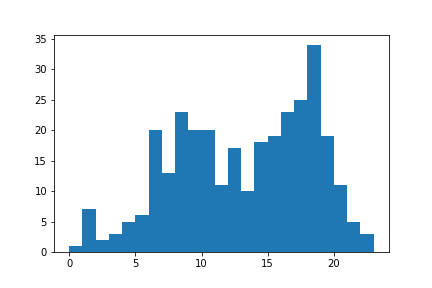
Despite the fact that the default bin-selection formula used by distplot might be better than the choice of ten bins that .hist uses, you'll still want to do some tweaking to align the bins to 'round' values. You can use other parameter settings to plot just the histogram and specify the bins like before:

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

sb.distplot(df['num\_var'], bins = bin\_edges, kde = **False**,

hist\_kws = {'alpha' : 1})

The alpha (transparency) setting must be associated as a dictionary to "hist\_kws" since there are other underlying plotting functions, like the KDE, that have their own optional keyword parameters.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/c0f67015-ef78-4bd8-98ea-78c25b5ac5b2)**

The result of the code above is exactly like the histogram above with bin width of 1. The units of the vertical axis are also back in terms of counts.

In summary, if your exploration is only interested in the histogram-depiction of the data, and not the additional functionality offered by distplot, then you might be better off with just using Matplotlib's hist function for simplicity. On the other hand, if you want a quick start on choosing a representative bin size for histogram plotting, you might take a quick look at the basic distplot first before getting into the customization.

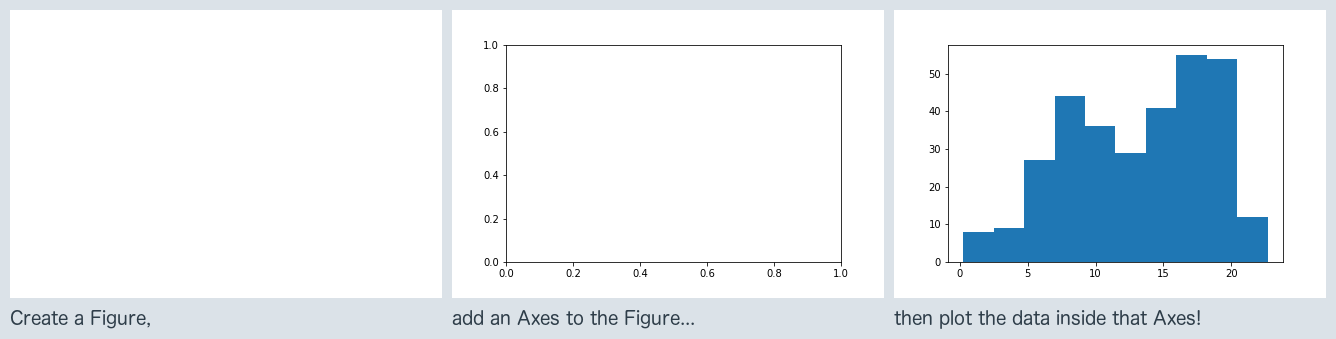
# Figures, Axes, and Subplots

At this point, you've seen and had some practice with some basic plotting functions using matplotlib and seaborn. The previous page introduced something a little bit new: creating two side-by-side plots through the use of matplotlib's [subplot()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html) function. If you have any questions about how that or the [figure()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.figure.html) function worked, then read on. This page will discuss the basic structure of visualizations using matplotlib and how subplots work in that structure.

The base of a visualization in matplotlib is a [Figure](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html) object. Contained within each Figure will be one or more [Axes](https://matplotlib.org/api/axes_api.html) objects, each Axes object containing a number of other elements that represent each plot. In the earliest examples, these objects have been created implicitly. Let's say that the following expression is run inside a Jupyter notebook to create a histogram:

plt.hist(data = df, x = 'num\_var')

Since we don't have a Figure area to plot inside, Python first creates a Figure object. And since the Figure doesn't start with any Axes to draw the histogram onto, an Axes object is created inside the Figure. Finally, the histogram is drawn within that Axes.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

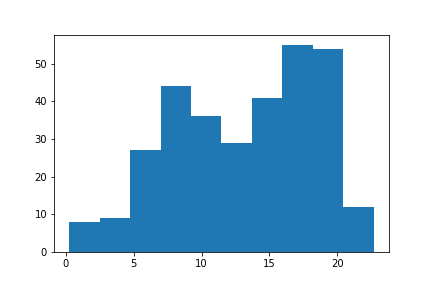
This hierarchy of objects is useful to know about so that we can take more control over the layout and aesthetics of our plots. One alternative way we could have created the histogram is to explicitly set up the Figure and Axes like this:

fig = plt.figure()

ax = fig.add\_axes([.125, .125, .775, .755])

ax.hist(data = df, x = 'num\_var')

[figure()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.figure.html) creates a new Figure object, a reference to which has been stored in the variable fig. One of the Figure methods is [.add\_axes()](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure.add_axes), which creates a new Axes object in the Figure. The method requires one list as argument specifying the dimensions of the Axes: the first two elements of the list the position of the lower-left hand corner of the Axes (in this case one quarter of the way from the lower-left corner of the Figure) and the last two elements specifying the Axes width and height, respectively. We refer to the Axes in the variable ax. Finally, we use the Axes method [.hist()](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.hist.html#matplotlib.axes.Axes.hist) just like we did before with plt.hist().

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

[This plot is just like the first histogram on the Histograms page.](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

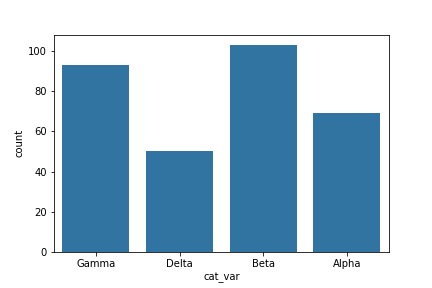
To use Axes objects with seaborn, seaborn functions usually have an "ax" parameter to specify upon which Axes a plot will be drawn.

fig = plt.figure()

ax = fig.add\_axes([.125, .125, .775, .755])

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

sb.countplot(data = df, x = 'cat\_var', color = base\_color, ax = ax)

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

[This is the same as the second plot on the Bar Charts page.](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

In the above two cases, there was no purpose to explicitly go through the Figure and Axes creation steps. And indeed, in most cases, you can just use the basic matplotlib and seaborn functions as is. Each function targets a Figure or Axes, and they'll automatically target the most recent Figure or Axes worked with. As an example of this, let's review in detail how [subplot()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html) was used on the Histograms page:

plt.figure(figsize = [10, 5]) *# larger figure size for subplots*

*# example of somewhat too-large bin size*

plt.subplot(1, 2, 1) *# 1 row, 2 cols, subplot 1*

bin\_edges = np.arange(0, df['num\_var'].max()+4, 4)

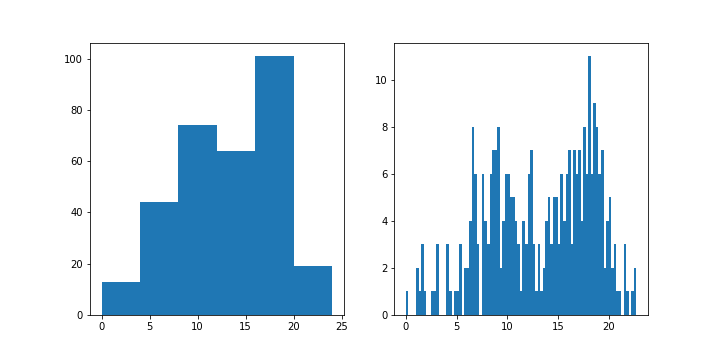
plt.hist(data = df, x = 'num\_var', bins = bin\_edges)

*# example of somewhat too-small bin size*

plt.subplot(1, 2, 2) *# 1 row, 2 cols, subplot 2*

bin\_edges = np.arange(0, df['num\_var'].max()+1/4, 1/4)

plt.hist(data = df, x = '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/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

First of all, plt.figure(figsize = [10, 5])creates a new Figure, with the "figsize" argument setting the width and height of the overall figure to 10 inches by 5 inches, respectively. Even if we don't assign any variable to return the function's output, Python will still implicitly know that further plotting calls that need a Figure will refer to that Figure as the active one.

Then, plt.subplot(1, 2, 1) creates a new Axes in our Figure, its size determined by the subplot() function arguments. The first two arguments says to divide the figure into one row and two columns, and the third argument says to create a new Axes in the first slot. Slots are numbered from left to right in rows from top to bottom. Note in particular that the index numbers start at 1 (rather than the usual Python indexing starting from 0). (You'll see the indexing a little better in the example at the end of the page.) Again, Python will implicitly set that Axes as the current Axes, so when the plt.hist() call comes, the histogram is plotted in the left-side subplot.

Finally, plt.subplot(1, 2, 2) creates a new Axes in the second subplot slot, and sets that one as the current Axes. Thus, when the next plt.hist() call comes, the histogram gets drawn in the right-side subplot.

## Additional Techniques

To close this page, we'll quickly run through a few other ways of dealing with Axes and subplots. The techniques above should suffice for basic plot creation, but you might want to keep the following in the back of your mind as additional tools to break out as needed.

If you don't assign Axes objects as they're created, you can retrieve the current Axes using [ax = plt.gca()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html), or you can get a list of all Axes in a Figure fig by using [axes = fig.get\_axes()](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure.get_axes). As for creating subplots, you can use [fig.add\_subplot()](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html" \l "matplotlib.figure.Figure.add_subplot" \t "_blank) in the same way as plt.subplot() above. If you already know that you're going to be creating a bunch of subplots, you can use the [plt.subplots()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html" \t "_blank) function:

fig, axes = plt.subplots(3, 4) *# grid of 3x4 subplots*

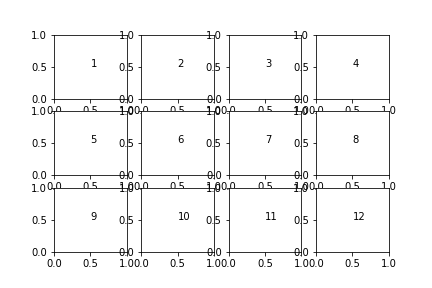
axes = axes.flatten() *# reshape from 3x4 array into 12-element vector*

**for** i **in** range(12):

plt.sca(axes[i]) *# set the current Axes*

plt.text(0.5, 0.5, i+1) *# print conventional subplot index number to middle of Axes*

As a special note for the text, the Axes limits are [0,1] on each Axes by default, and we increment the iterator counter i by 1 to get the subplot index, if we were creating the subplots through subplot(). (Reference: [plt.sca()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.sca.html" \t "_blank), [plt.text()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.text.html" \t "_blank))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/83335e78-7d71-423b-84a9-f47cdf64c0f9)

## Documentation

Documentation pages for Figure and Axes objects are linked below. Note that they're pretty dense, so I don't suggest reading them until you need to dig down deeper and override matplotlib or seaborn's default behavior. Even then, they are just reference pages, so they're better for skimming or searching in case other internet resources don't provide enough detail.

* [Figure](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html)
* [Axes](https://matplotlib.org/api/axes_api.html)

# Choosing a Plot for Discrete Data

If you want to plot a discrete quantitative variable, it is possible to select either a histogram or a bar chart to depict the data.

The histogram is the most immediate choice since the data is numeric, but there's one particular consideration to make regarding the bin edges. Since data points fall on set values, it can help to reduce ambiguity by putting bin edges between the actual values taken by the data. Your readers may not know that values on bin edges end up in the bin to their right, so this can help remove potential confusion when they interpret the plot. Compare the two visualizations of 100 random die rolls below (in die\_rolls), with bin edges \_on\_ the values in the left subplot, and bin edges in between values in the right subplot.

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

*# histogram on the left, bin edges on integers*

plt.subplot(1, 2, 1)

bin\_edges = np.arange(2, 12+1.1, 1) *# note `+1.1`, see below*

plt.hist(die\_rolls, bins = bin\_edges)

plt.xticks(np.arange(2, 12+1, 1))

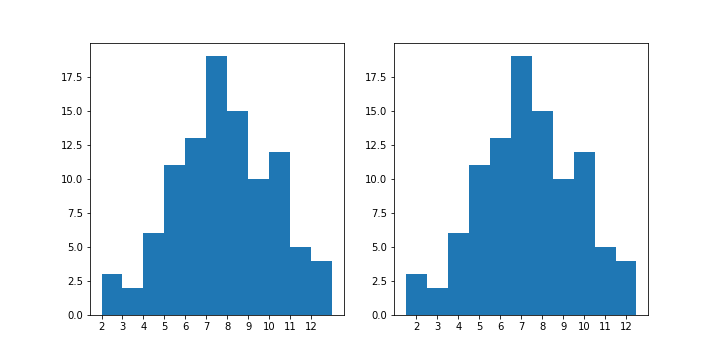
*# histogram on the right, bin edges between integers*

plt.subplot(1, 2, 2)

bin\_edges = np.arange(1.5, 12.5+1, 1)

plt.hist(die\_rolls, bins = bin\_edges)

plt.xticks(np.arange(2, 12+1, 1))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/74ab8c07-01e4-407c-8941-350a2cf49e78)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/74ab8c07-01e4-407c-8941-350a2cf49e78)

You'll notice for the left histogram, in a deviation from the examples that have come before, I've added 1.1 to the max value (12) for setting the bin edges, rather than just the desired bin width of 1. Recall that data that is equal to the rightmost bin edge gets lumped in to the last bin. This presents a potential problem in perception when a lot of data points take the maximum value, and so is especially relevant when the data takes on discrete values. The 1.1 adds an additional bin to the end to store the die rolls of value 12 alone, to avoid having the last bar catch both 11 and 12.

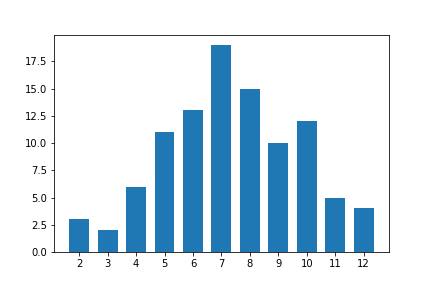
Alternatively to the histogram, consider if a bar chart with non-connected bins might serve your purposes better. The plot below takes the code from before, but adds the "rwidth" parameter to set the proportion of the bin widths that will be filled by each histogram bar.

bin\_edges = np.arange(1.5, 12.5+1, 1)

plt.hist(die\_rolls, bins = bin\_edges, rwidth = 0.7)

plt.xticks(np.arange(2, 12+1, 1))

With "rwidth" set to 0.7, the bars will take up 70% of the space allocated by each bin, with 30% of the space left empty. This changes the default display of the histogram (which you could think of as "rwidth = 1") into a bar chart.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/74ab8c07-01e4-407c-8941-350a2cf49e78)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/74ab8c07-01e4-407c-8941-350a2cf49e78)

By adding gaps between bars, you emphasize the fact that the data is discrete in value. On the other hand, plotting your quantitative data in this manner might cause it to be interpreted as ordinal-type data, which can have an effect on overall perception.

For continuous numeric data, you should not make use of the "rwidth" parameter, since the gaps imply discreteness of value. As another caution, it might be tempting to use seaborn's countplot function to plot the distribution of a discrete numeric variable as bars. Be careful about doing this, since each unique numeric value will get a bar, regardless of the spacing in values between bars. (For example, if the unique values were {1, 2, 4, 5}, missing 3, countplot would only plot four bars, with the bars for 2 and 4 right next to one another.) Also, even if your data is technically discrete numeric, you should probably not consider either of the variants depicted on this page unless the number of unique values is small enough to allow for the half-unit shift or discrete bars to be interpretable. If you have a large number of unique values over a large enough range, it's better to stick with the standard histogram than risk interpretability issues.

While you might justify plotting discrete numeric data using a bar chart, you’ll be less apt to justify the opposite: plotting ordinal data as a histogram. The space between bars in a bar chart helps to remind the reader that values are not contiguous in an ‘interval’-type fashion: only that there is an order in levels. With that space removed as in a histogram, it's harder to remember this important bit of interpretation.

# Descriptive Statistics, Outliers, and Axis Limits

As you create your plots and perform your exploration, make sure that you pay attention to what the plots tell you that go beyond just the basic descriptive statistics. Note any aspects of the data like number of modes and skew, and note the presence of outliers in the data for further investigation.

Related to the latter point, you might need to change the limits or scale of what is plotted to take a closer look at the underlying patterns in the data. This page covers the topic of axis limits; the next the topic of scales and transformations. In order to change a histogram's axis limits, you can add a Matplotlib [xlim](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlim.html" \t "_blank) call to your code. The function takes a tuple of two numbers specifying the upper and lower bounds of the x-axis range. Alternatively, the xlim function can be called with two numeric arguments to the same result.

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

*# histogram on left: full data*

plt.subplot(1, 2, 1)

bin\_edges = np.arange(0, df['skew\_var'].max()+2.5, 2.5)

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

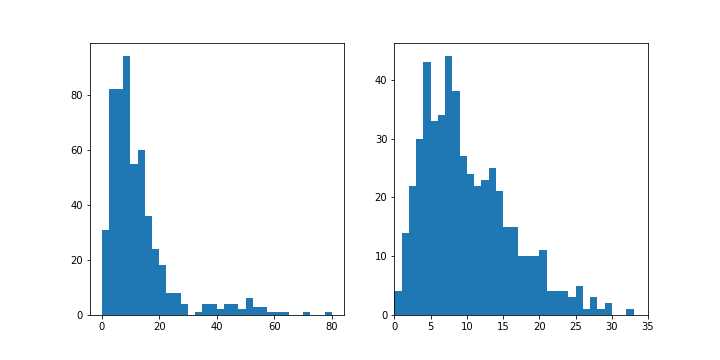
*# histogram on right: focus in on bulk of data < 35*

plt.subplot(1, 2, 2)

bin\_edges = np.arange(0, 35+1, 1)

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

plt.xlim(0, 35) *# could also be called as plt.xlim((0, 35))*

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/1523cdaa-1c53-47fd-ab3e-a31631dcacad)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/1523cdaa-1c53-47fd-ab3e-a31631dcacad)

In the generic example above, we might be interested in comparing patterns in other variables between data points that take values less than 35 to those that take values greater than 35. For anything that is concentrated on the bulk of the data in the former group (< 35), use of axis limits can allow focusing on data points in that range without needing to go through creation of a new DataFrame filtering out the data points in the latter group (> 35).

# Scales and Transformations

Certain data distributions will find themselves amenable to scale transformations. The most common example of this is data that follows an approximately [log-normal](https://en.wikipedia.org/wiki/Log-normal_distribution) distribution. This is data that, in their natural units, can look highly skewed: lots of points with low values, with a very long tail of data points with large values. However, after applying a logarithmic transform to the data, the data will follow a normal distribution. (If you need a refresher on the logarithm function, check out [this lesson on Khan Academy](https://www.khanacademy.org/math/algebra2/exponential-and-logarithmic-functions).)

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

*# left histogram: data plotted in natural units*

plt.subplot(1, 2, 1)

bin\_edges = np.arange(0, data.max()+100, 100)

plt.hist(data, bins = bin\_edges)

plt.xlabel('values')

*# right histogram: data plotted after direct log transformation*

plt.subplot(1, 2, 2)

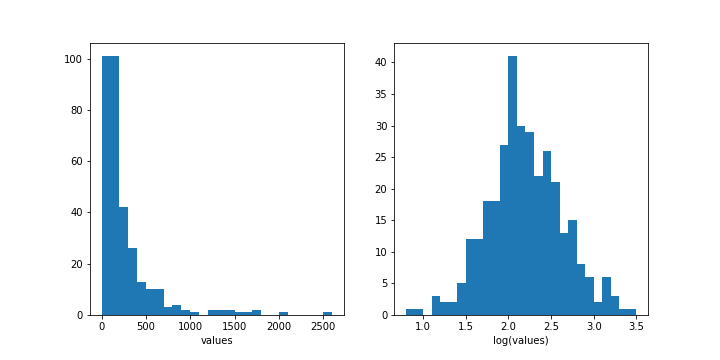
log\_data = np.log10(data) *# direct data transform*

log\_bin\_edges = np.arange(0.8, log\_data.max()+0.1, 0.1)

plt.hist(log\_data, bins = log\_bin\_edges)

plt.xlabel('log(values)')

(Documentation: [numpy log10](https://docs.scipy.org/doc/numpy/reference/generated/numpy.log10.html), [matplotlib xlabel](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html))

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)

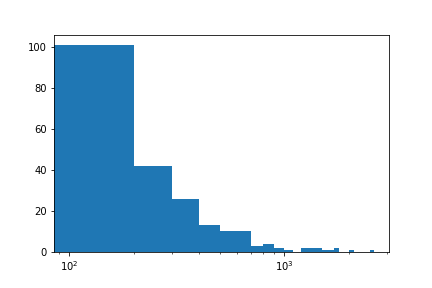
In the plot on the left, the few data points with value above 1000 mash the majority of the points into the bins on the far left. With the plot on the right, the logarithmic transform makes those large points look in line with the rest: a raw value of 1000 becomes a value of 3 under log transform, and a raw value of 100 becomes a log-transformed value of 2. The big problem with the right-side plot is that the units on the x-axis are difficult to interpret: for most people, it is only easy to convert from log values to natural values on the integers (and this assumes a nice base like 10 as used in the example).

This is where scale transformations are handy. In a scale transformation, the gaps between values are based on the transformed scale, but you can interpret data in the variable's natural units. It is also a convenient approach since you won't need to engineer new features. Matplotlib's [xscale](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xscale.html" \t "_blank) function includes a few built-in transformations: we'll use the 'log' scale here.

bin\_edges = np.arange(0, data.max()+100, 100)

plt.hist(data, bins = bin\_edges)

plt.xscale('log')

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)

Notice two things about the plot now. Even though the data is on a log scale, the bins are still linearly spaced. This means that they change size from wide on the left to thin on the right, as the values increase multiplicatively. Secondly, the default label settings are still somewhat tricky to interpret, and are sparse as well.

To address the bin size issue, we just need to change them so that they are evenly-spaced powers of 10. Depending on what you are plotting, a different base power like 2 might be useful instead. For the ticks, we can use [xticks](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xticks.html" \t "_blank) to specify locations and labels in their natural units. Remember: we aren't changing the values taken by the data, only how they're displayed. Between integer powers of 10, we don't have clean values for even markings, but we can still get close. Setting ticks in cycles of 1-3-10 or 1-2-5-10 are very useful for base-10 log transforms.

bin\_edges = 10 \*\* np.arange(0.8, np.log10(data.max())+0.1, 0.1)

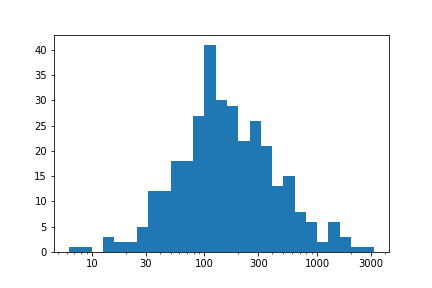
plt.hist(data, bins = bin\_edges)

plt.xscale('log')

tick\_locs = [10, 30, 100, 300, 1000, 3000]

plt.xticks(tick\_locs, tick\_locs)

It is important that the xticks are specified after xscale since that function has its own built-in tick settings.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)

We've ended up with the same plot as when we performed the direct log transform, but now with a much nicer set of tick marks and labels.

## Alternative Approach

Be aware that a logarithmic transformation is not the only one possible. When we perform a logarithmic transformation, our data values have to all be positive; it's impossible to take a log of zero or a negative number. In addition, the transformation implies that additive steps on the log scale will result in multiplicative changes in the natural scale, an important implication when it comes to data modeling. The type of transformation that you choose may be informed by the context for the data. For example, [this Wikipedia section](https://en.wikipedia.org/wiki/Log-normal_distribution#Occurrence_and_applications) provides a few examples of places where log-normal distributions have been observed.

If you want to use a different transformation that's not available in xscale, then you'll have to perform some feature engineering. In cases like this, we want to be systematic by writing a function that applies both the transformation and its inverse. The inverse will be useful in cases where we specify values in their transformed units and need to get the natural units back. For the purposes of demonstration, let's say that we want to try plotting the above data on a square-root transformation. (Perhaps the numbers represent areas, and we think it makes sense to model the data on a rough estimation of radius, length, or some other 1-d dimension.) We can create a visualization on this transformed scale like this:

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

""" transformation helper function """

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

**return** np.sqrt(x)

**else**:

**return** x \*\* 2

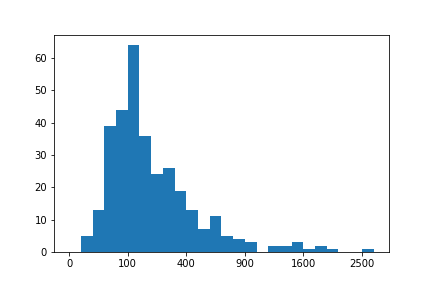
bin\_edges = np.arange(0, sqrt\_trans(data.max())+1, 1)

plt.hist(data.apply(sqrt\_trans), bins = bin\_edges)

tick\_locs = np.arange(0, sqrt\_trans(data.max())+10, 10)

plt.xticks(tick\_locs, sqrt\_trans(tick\_locs, inverse = **True**).astype(int))

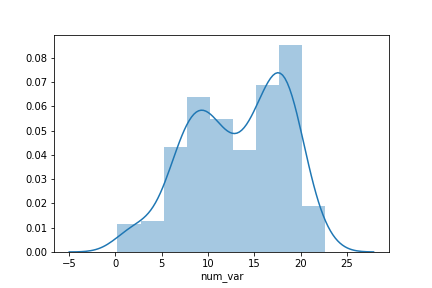
Note that data is a pandas Series, so we can use the [apply](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.apply.html) method for the function. If it were a NumPy Array, we would need to apply the function like in the other cases. The tick locations could have also been specified with the natural values, where we apply the standard transformation function on the first argument of xticks instead.

[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/7623c273-f807-4da3-9ce4-93124bc9ce42)

# Kernel Density Estimation

Earlier in this lesson, you saw an example of [**kernel density estimation**](https://en.wikipedia.org/wiki/Kernel_density_estimation) (KDE) through the use of seaborn's **[distplot](https://seaborn.pydata.org/generated/seaborn.distplot.html" \l "seaborn.distplot" \t "_blank)** function, which plots a KDE on top of a histogram.

sb.distplot(df['num\_var'])

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/cfbaee5f-1a27-4457-8a67-8421bf19cb05)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/cfbaee5f-1a27-4457-8a67-8421bf19cb05)**

Kernel density estimation is one way of estimating the probability density function of a variable. In a KDE plot, you can think of each observation as replaced by a small ‘lump’ of area. Stacking these lumps all together produces the final density curve. The default settings use a normal-distribution kernel, but most software that can produce KDE plots also include other kernel function options.

Seaborn's distplot function calls another function, **[kdeplot](https://seaborn.pydata.org/generated/seaborn.kdeplot.html" \t "_blank)**, to generate the KDE. The demonstration code below also uses a third function called by distplot for illustration, **[rugplot](https://seaborn.pydata.org/generated/seaborn.rugplot.html" \t "_blank)**. In a rugplot, data points are depicted as dashes on a number line.

data = [0.0, 3.0, 4.5, 8.0]

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

*# left plot: showing kde lumps with the default settings*

plt.subplot(1, 3, 1)

sb.distplot(data, hist = **False**, rug = **True**, rug\_kws = {'color' : 'r'})

*# central plot: kde with narrow bandwidth to show individual probability lumps*

plt.subplot(1, 3, 2)

sb.distplot(data, hist = **False**, rug = **True**, rug\_kws = {'color' : 'r'},

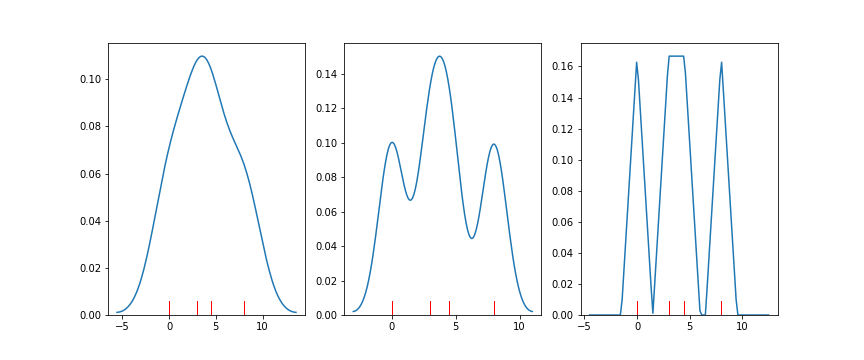
kde\_kws = {'bw' : 1})

*# right plot: choosing a different, triangular kernel function (lump shape)*

plt.subplot(1, 3, 3)

sb.distplot(data, hist = **False**, rug = **True**, rug\_kws = {'color' : 'r'},

kde\_kws = {'bw' : 1.5, 'kernel' : 'tri'})

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/cfbaee5f-1a27-4457-8a67-8421bf19cb05)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/cfbaee5f-1a27-4457-8a67-8421bf19cb05)**

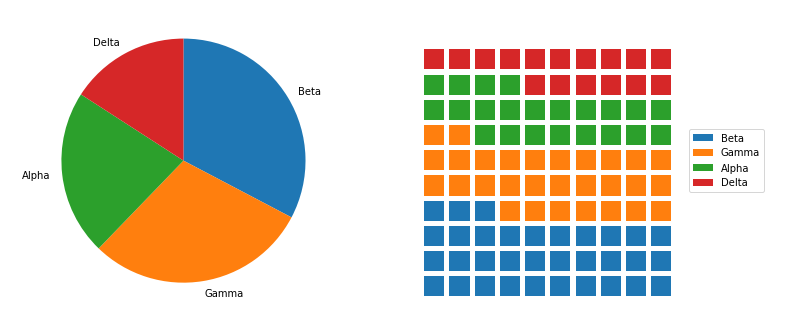
Interpreting proportions from this plot type is slightly trickier than a standard histogram: the vertical axis indicates a density of data rather than straightforward proportions. Under a KDE plot, the total area between the 0-line and the curve will be 1. The probability of an outcome falling between two values is found by computing the area under the curve that falls between those values. Making area judgments like this without computer assistance is difficult and likely to be inaccurate.

Despite the fact that making specific probability judgments are not as intuitive with KDE plots as histograms, there are still reasons to use kernel density estimation. If there are relatively few data points available, KDE provides a smooth estimate of the overall distribution of data. These ideas may not be so easily conveyed through histograms, in which the large discreteness of jumps may end up misleading.

It should also be noted that there is a bandwidth parameter in KDE that specifies how wide the density lumps are. Similar to bin width for histograms, we need to choose a bandwidth size that best shows the signal in the data. A too-small bandwidth can make the data look noisier than it really is, and a too-large bandwidth can smooth out useful features that we could use to make inferences about the data. It’s good to keep this in mind in case the default bandwidths chosen by your visualization software don’t look quite right or if you need to perform further investigations.

# Waffle Plots

One alternative univariate plot type that you might see for categorical data is the **waffle plot**, also known as the square pie chart. While the standard pie chart uses a circle to represent the whole, a waffle plot is plotted onto a square divided into a 10x10 grid. Each small square in the grid represents one percent of the data, and a number of squares are colored by category to indicate total proportions. Compared to a pie chart, it is much easier to make precise assessments of relative frequencies.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

**[You've seen code for the pie chart (left) previously in the lesson. Code for the waffle plot (right) will be walked through below.](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

There's no built-in function for waffle plots in Matplotlib or Seaborn, so we'll need to take some additional steps in order to build one with the tools available. First, we need to create a function to decide how many blocks to allocate to each category. The function below, percentage\_blocks, uses a rule where each category gets a number of blocks equal to the number of full percentage points it covers. The remaining blocks to get to the full one hundred are assigned to the categories with the largest fractional parts.

**def** **percentage\_blocks**(df, var):

"""

Take as input a dataframe and variable, and return a Pandas series with

approximate percentage values for filling out a waffle plot.

"""

*# compute base quotas*

percentages = 100 \* df[var].value\_counts() / df.shape[0]

counts = np.floor(percentages).astype(int) *# integer part = minimum quota*

decimal = (percentages - counts).sort\_values(ascending = **False**)

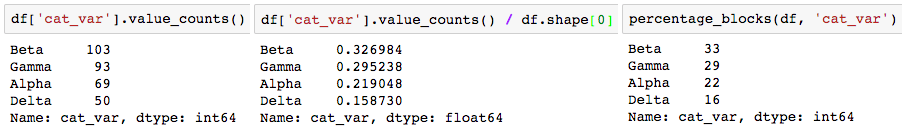
*# add in additional counts to reach 100*

rem = 100 - counts.sum()

**for** cat **in** decimal.index[:rem]:

counts[cat] += 1

**return** counts

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

**[Note that if we just rounded the proportions (center), we would round all of them up, ending up with a total of 101 blocks.](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

Now it's time to actually plot those counts as boxes in the waffle plot form. To do this, we'll make use of Matplotlib's [**bar**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html) function. We could have used this function earlier in the lesson to create our bar charts instead of Seaborn's countplot, but it would have required us to aggregate the data first to get the height of each bar. For the case of the waffle plot, we're going to specify the x- and y- coordinates of the boxes, and set their widths and heights to be equal, to create squares. The initial plotting code looks like this:

waffle\_counts = percentage\_blocks(df, 'cat\_var')

prev\_count = 0

*# for each category,*

**for** cat **in** range(waffle\_counts.shape[0]):

*# get the block indices*

blocks = np.arange(prev\_count, prev\_count + waffle\_counts[cat])

*# and put a block at each index's location*

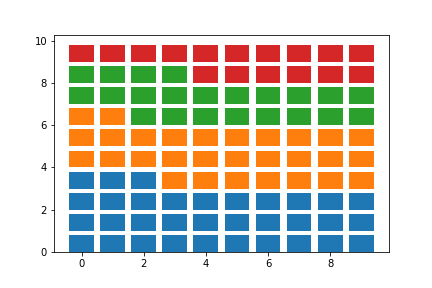
x = blocks % 10 *# use mod operation to get ones digit*

y = blocks // 10 *# use floor division to get tens digit*

plt.bar(x = x, height = 0.8, width = 0.8, bottom = y)

prev\_count += waffle\_counts[cat]

The blocks are drawn from left to right, bottom to top, using the ones and tens digits for numbers from 0 to 99 to specify the x- and y- positions, respectively. A loop is used to call the bar function once for each category; each time it is called, the plotted bars are assigned a different color.

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

The last steps that we need to do involve aesthetic cleaning to polish it up for interpretability. We can take away the plot border and ticks, since they're arbitrary, but we should change the limits so that the boxes are square. We should also add a legend so that the mapping from colors to category levels is clear.

waffle\_counts = percentage\_blocks(df, 'cat\_var')

prev\_count = 0

*# for each category,*

**for** cat **in** range(waffle\_counts.shape[0]):

*# get the block indices*

blocks = np.arange(prev\_count, prev\_count + waffle\_counts[cat])

*# and put a block at each index's location*

x = blocks % 10 *# use mod operation to get ones digit*

y = blocks // 10 *# use floor division to get tens digit*

plt.bar(x = x, height = 0.8, width = 0.8, bottom = y)

prev\_count += waffle\_counts[cat]

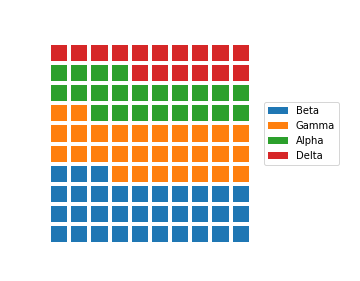
*# aesthetic wrangling*

plt.legend(waffle\_counts.index, bbox\_to\_anchor = (1, 0.5), loc = 6)

plt.axis('off')

plt.axis('square')

The two calls to Matplotlib's [**axis**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.axis.html) function make use of two convenience strings for arguments: 'off' removes the axis lines, ticks, and labels, while 'square' ensures that the scaling on each axis is equal within a square bounding box. As for the [**legend**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html) call, the first argument is a list of categories as obtained from the sorted waffle\_counts Series variable. This will match each category to each bar call, in order. The "bbox\_to\_anchor" argument sets an anchor for the legend to the right side of the plot, and "loc = 6" positions the anchor to the center left of the legend. The final plot is as it looks at the top of the page:

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

Other variants of the waffle plot exist to extend it beyond just displaying probabilities. By associating each square with an amount rather than a percentage, we can use waffle plots to show absolute frequencies instead. This might cause us to end up with something other than 100 squares.

*# each box represents five full counts*

waffle\_counts = (df['cat\_var'].value\_counts() / 5).astype(int)

prev\_count = 0

*# for each category,*

**for** cat **in** range(waffle\_counts.shape[0]):

*# get the block indices*

blocks = np.arange(prev\_count, prev\_count + waffle\_counts[cat])

*# and put a block at each index's location*

x = blocks % 10

y = blocks // 10

plt.bar(y, 0.8, 0.8, x)

prev\_count += waffle\_counts[cat]

*# box size legend*

plt.bar(7.5, 0.8, 0.8, 2, color = 'white', edgecolor = 'black', lw = 2)

plt.text(8.1, 2.4,'= 5 data points', va = 'center')

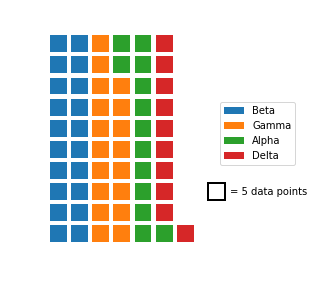
*# aesthetic wrangling*

plt.legend(waffle\_counts.index, bbox\_to\_anchor = (0.8, 0.5), loc = 6)

plt.axis('off')

plt.axis('square')

In the above code, waffle\_counts has been adjusted so that each box represents 5 data points. Most of the code is the same as before, though it should be noted that the x and y variables have been swapped in the bar function so that the boxes are plotted in columns from left to right. Additional bar and [**text**](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.text.html) calls have been added to the plot to act as an ad hoc legend. The positions of these elements, and the legend, have been adjusted manually through some trial and error to improve the aesthetic appeal. Note that this constitutes more of an explanatory polishing than it is a part of exploration!

**[[](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)](https://classroom.udacity.com/nanodegrees/nd002-ent/parts/9c2eb5bf-2bda-4e0c-85bf-d997082a8571/modules/22d4ed2d-b711-4e1d-8fed-604f524f0595/lessons/24200b64-07aa-4526-b558-df1ae31faa7c/concepts/48ccde12-1496-48e0-ba5e-1b3fa2aa0083)**

As a further extension, there's no restriction against us using icons for each tally, rather than just squares. Infographics often take this approach, by having each icon represent some number of units (e.g. one person icon representing one million people). But while it can be tempting to use icons to represent values as a bit of visual flair, an icon-based plot contains more chart junk than a bar chart that conveys the same information. There’s a larger cognitive challenge in having to count a number of icons to understand the scale of a value, compared to just referencing a box's endpoint on a labeled axis.

One other downside of the waffle plot is that it is not commonly supported out of the box for most visualization libraries, including Matplotlib and Seaborn. The length of the demonstration code presented above is a testament to that. The effort required to create a meaningful and useful waffle plot means that it is best employed carefully as a part of explanatory visualizations. During the exploratory phase, you're better off using more traditional plots like the bar chart to more rapidly build your understanding of the data.

## Additional Resources

You don't actually need to go through all of the code wrangling shown above to create waffle plots in Python. The **[PyWaffle](https://github.com/ligyxy/PyWaffle" \t "_blank)** package can be used with Matplotlib's figure function to create waffle plots, with a few options for the orientation and order of icons, but you'll need to install it separately since it's not a major package. One of the main reasons why I didn't use it above is that the syntax for using it is very different from what you've seen and will see in this course. If you want to make use of the library, check out the examples on the linked GitHub page.

NEXT