

Statistical data visualizations (static)

BIOF 440

Abhijit Dasgupta

Data visualization

Visualization for analysis

- Tool for understanding datasets
- You ask questions and quickly answer them
- Iterate to develop insights.

Context is important

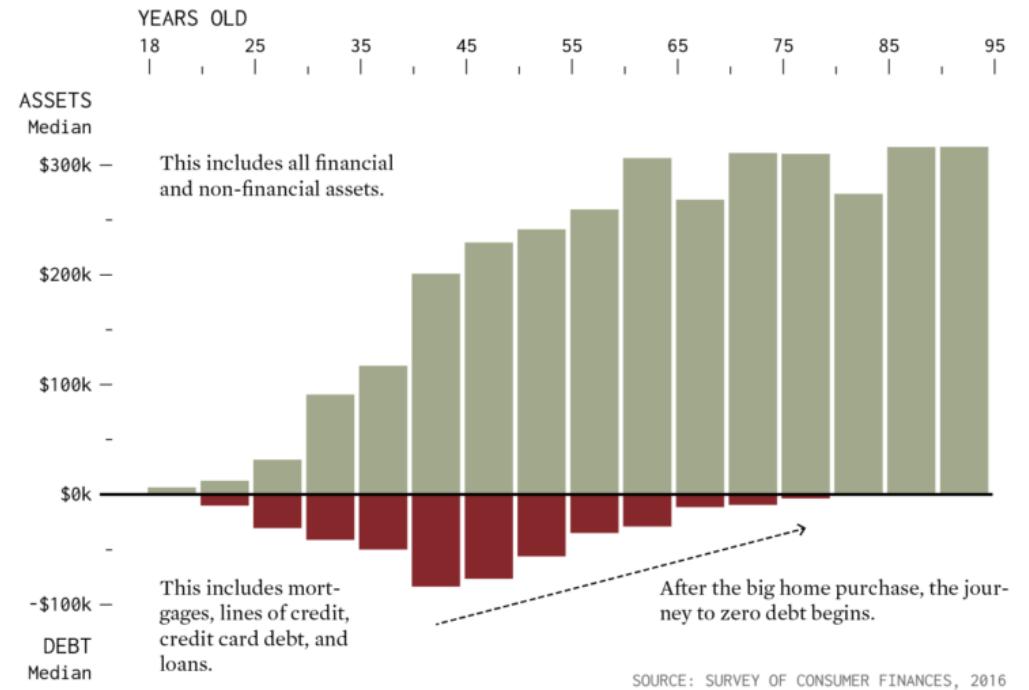
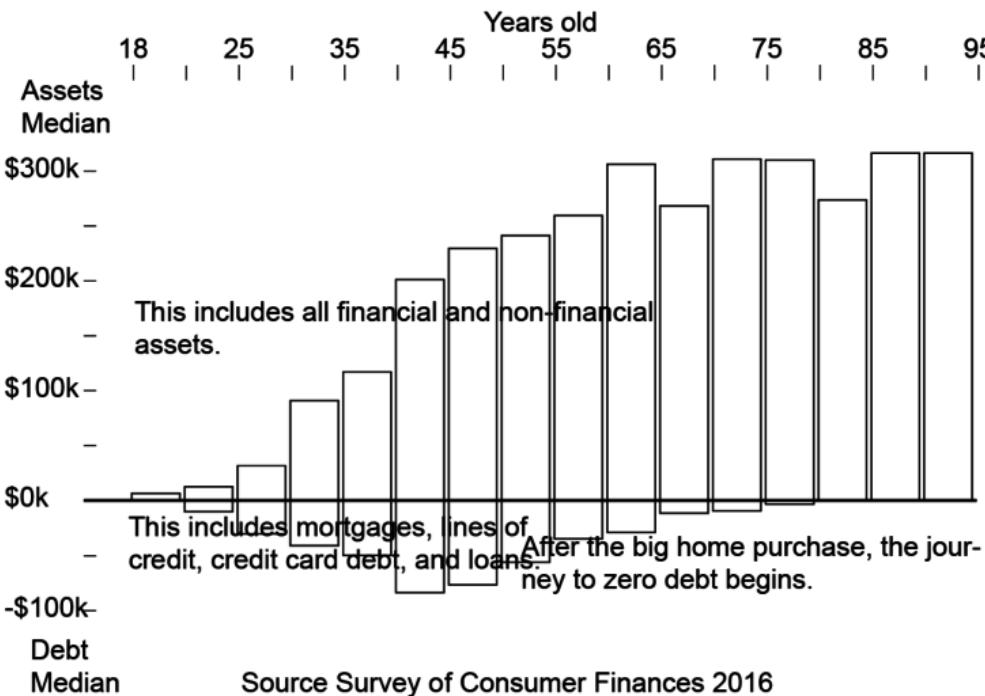
A data visualization should be self-contained and be able to express the **context** of the data.

This makes the visualization **informative**.



Improve readability

Data visualizations should be readable. It should be obvious what the chart is about and how to interpret it.



Some ideas

- Your visualization depends on your audience
 - If the audience is your lab group (has a similar contextual background), then you may not have to provide as much context in your visualization
 - If your audience is a conference or a journal reader, you probably need to have more detail and context within the visualization
- Your visualization needs to reflect the character of the data
 - Continuous <--> categorical <--> binary
 - Dates and times
 - Spatial data (may be related to objects other than maps, like cell structure or organisms)
 - Networked or co-related data

Design is choice. The theory of the visual display of quantitative information consists of principles that generate design options and that guide choices among options. The principles should not be applied rigidly or in a peevish spirit; they are not logically or mathematically certain; and it is better to violate any principle than to place graceless or inelegant marks on paper. Most principles of design should be greeted with some skepticism, for word authority can dominate our vision, and we may come to see only through the lenses of word authority rather than with our own eyes.

-- Edward Tufte, *The Visual Display of Quantitative Data*

Tufte's Principles of Graphical Integrity

1. Show data variation, not design variation
2. Do not use graphics to quote data out of context
3. Use clear, detailed, thorough labelling.
4. Representation of numbers should be directly proportional to numerical quantities
5. Don't use more dimensions than the data require

Tufte's Principles of Graphical Integrity

1. Show data variation, not design variation
 - Don't get fancy, let the data speak
2. Do not use graphics to quote data out of context
 - Maintain accuracy
3. Use clear, detailed, thorough labelling.
 - Use annotations to make your point
4. Representation of numbers should be directly proportional to numerical quantities
 - This is essential for fair representation
5. Don't use more dimensions than the data require
 - Be appropriate in use of 3D graphics, for example

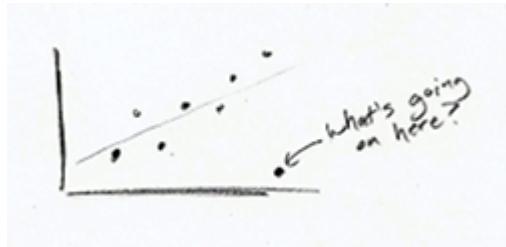
Tufte's Fundamental Principles of Design

1. Show comparisons
2. Show causality
3. Use multivariate data
4. Completely integrate modes (like text, images, numbers)
5. Establish credibility
6. Focus on content

Nathan Yau's Seven Basic Rules for Making Charts and Graphs

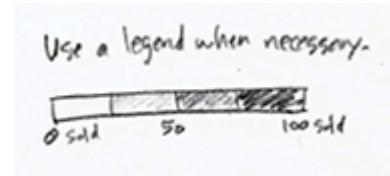
1. Check the data
2. Explain encodings
3. Label axes
4. Include units
5. Keep your geometry in check
6. Include your sources
7. Consider your audience

1) Check the data



- This should be obvious
- If your data is weak, your chart is weak
- Start with simple graphs to see if there are any outliers

2) Explain encodings



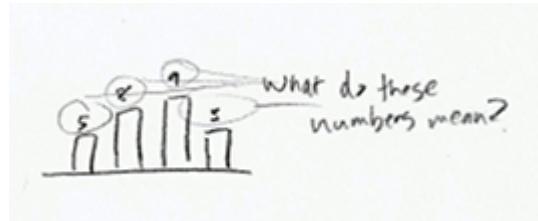
- Don't assume the reader knows what everything means
- Provide a legend
- Label shapes
- Explain color scales

3) Label axes



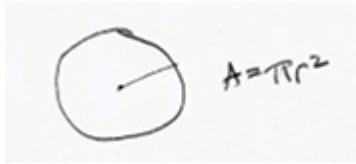
- Axes without labels or explanation are just decorations
- Describe the scale (incremental, exponential, logarithmic?)
- Have axes values start at zero

4) Include units



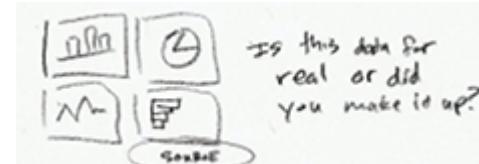
- Numbers without units are meaningless
- Remove the guesswork

5) Keep your geometry in check



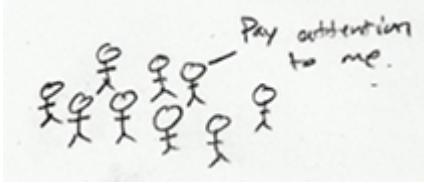
- This is something that is immediately noticeable
- Don't use area to compare two units unless they are an area. An increase in a unit squares the area.
- Tip: size circles and other 2D shapes by area, unless it's a bar chart

6) Include your sources



- This is another obvious one
- Always include the source of your data
- Makes your graphic more reputable
- Allows for others to dig deeper

7) Consider your audience



- What purpose do your charts have and who are they for?
- Avoid quirky fonts
- Make good design choices

Data to graphics

Visual encoding

The basic question in data visualization is how we transform data values into blobs of ink on paper, or more recently, pixels on a screen.

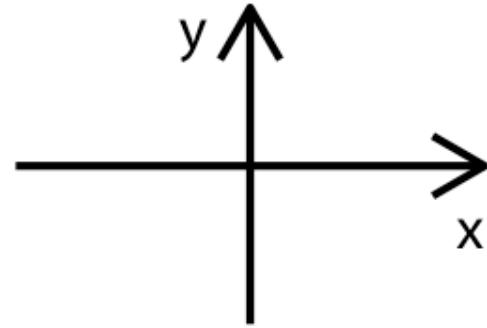
All data visualizations map data values into quantifiable features of the resulting graphic. We refer to these as *aesthetics*

-- *Fundamentals of Data Visualization* by Claus O. Wilke

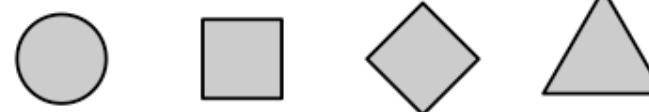
We can also refer to these as **visual encoding**, i.e., how we encode aspects of data visually

Common aesthetics/encodings

position



shape



size



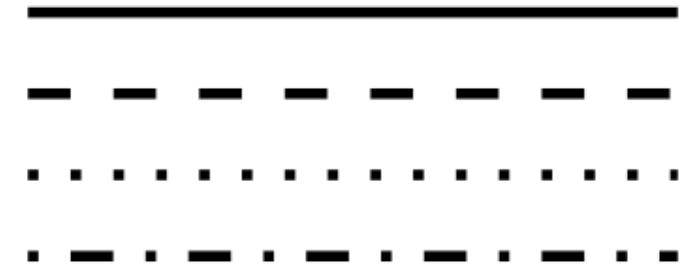
color



line width



line type



Common aesthetics/encodings

The choice of aesthetics often will be guided by the kind of data you're trying to visualize, as we said earlier

- Quantitative / continuous
- Categorical ordered
- Categorical unordered
- Time (dates, times, years)

Common aesthetics/encodings

Type	Encodings	Notes
Continuous	x, y, size, color, line width	sequential and divergent color scales
Ordered categorical	x, y, size, shape, color, line type, line width	sequential and divergent color scales
Unordered categorical	x, y, shape, color, line type	qualitative color scales
Time	x, y	

Can you think of examples where we can encode data types with different kinds of visualizations?

Statistical Data Visualization

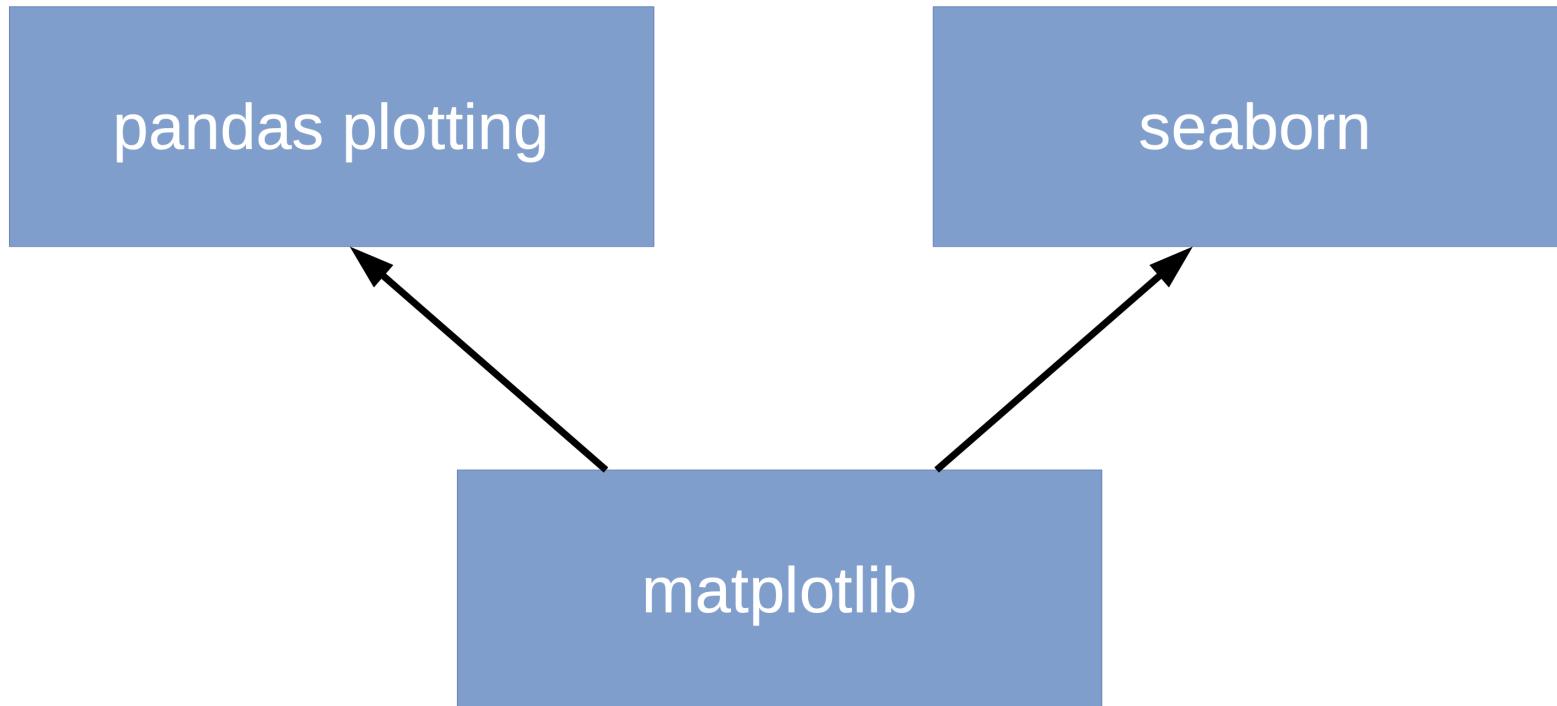
Statistical data visualization

In this class we will mainly be dealing with statistical data visualizations, rather than visualizing functions and fixed patterns.

The package [seaborn](#) will be our main high-level Python tool to enable us to do this.

seaborn, pandas and matplotlib

- pandas is the main tool to prepare data for visualization. It also has some plotting capabilities
- seaborn is the main tool for statistical visualizations as a high-level tool
- matplotlib is the primary tool for static data visualization in Python
 - Create customizable plots
 - Very granular; can control almost all aspects of a graph



seaborn, pandas and matplotlib

- Both `pandas` plotting and `seaborn` are built on top of `matplotlib`
- Both `pandas` plotting and `seaborn` allow the creation of data visualizations with simpler code
- You can use the capabilities of `matplotlib` to
 - set up visualizations
 - customize visualizations
 - save visualizations

Let's get started

Setting up

We'll use this set of packages almost always for creating static visualizations meant for a paper, poster, or website

```
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
plt.style.use("seaborn-whitegrid")
```

This will be the usual setup for the material this week

Starting with basic pandas plots

With `pandas`, we can do quite a bit of basic statistical plotting.

It also allows us to see the direct relationship between the data and visualizations

We can plot from both `Series` and `DataFrame` objects

`pandas` was originally built to work with time series, so a tacit assumption underlying `pandas` plotting is that the index of the `DataFrame` or `Series` is a series of dates or times, and each column is data collected at each of these time points for a particular variable. So a `DataFrame` was assumed to be a set of time series, and the plotting tools were designed accordingly. However, we won't follow that assumption here, since a `DataFrame` is used much more richly in data science.

Starting with basic pandas plots

Let's start by importing the `cars2018.csv` into our session

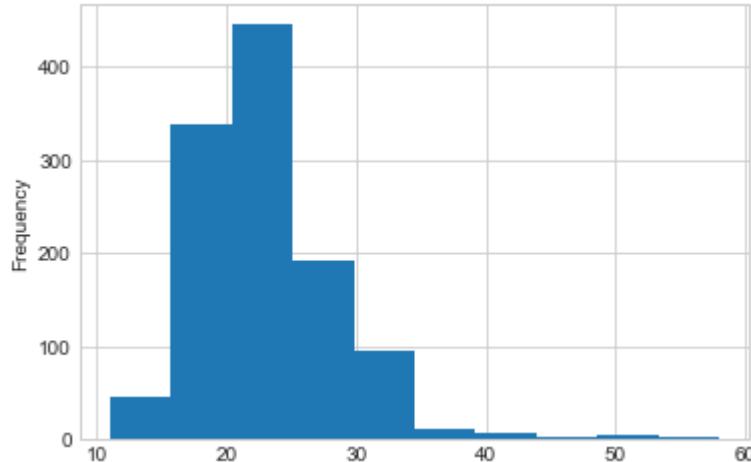
```
cars = pd.read_csv("data/cars2018.csv")
cars.head()
```

```
          Model  Model Index  ...  Exhaust Valves Per Cyl  Fuel injection
0      Acura NSX           57  ...                      2  Direct ignition
1  ALFA ROMEO 4C          410  ...                      2  Direct ignition
2    Audi R8 AWD            65  ...                      2  Direct ignition
3    Audi R8 RWD            71  ...                      2  Direct ignition
4  Audi R8 Spyder AWD        66  ...                      2  Direct ignition
[5 rows x 15 columns]
```

Visualizing one variable (Continuous)

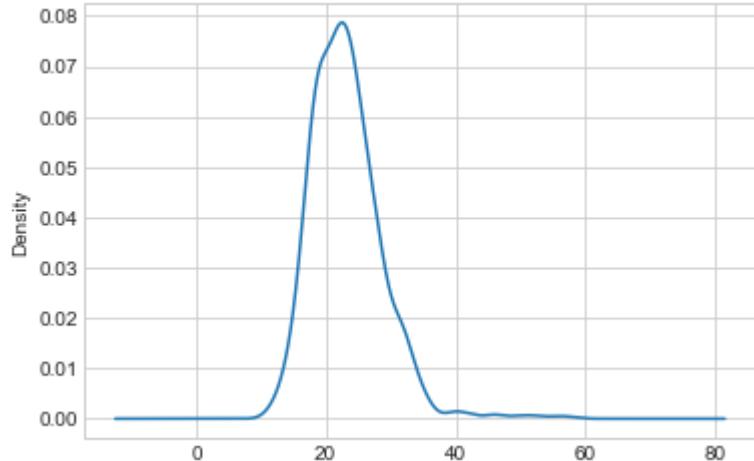
Histogram

```
cars["MPG"].plot(kind="hist")
```



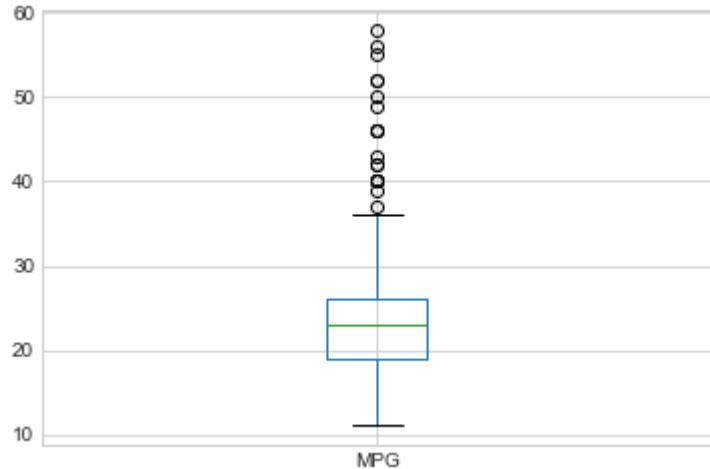
Density plot

```
cars[ "MPG" ].plot(kind="kde")
```



Box plot

```
cars[ "MPG" ].plot(kind="box")
```



Visualizing one variable (categorical)

Frequency barplot

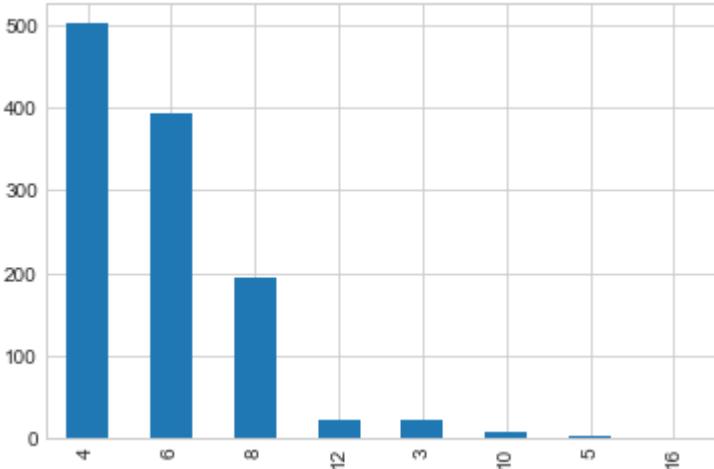
For a frequency barplot, you need to do a bit of data summarization using [pandas](#)

```
cars["Cylinders"].value_counts()
```

```
4      502
6      392
8      195
12     23
3      21
10     8
5      2
16     1
Name: Cylinders, dtype: int64
```

Frequency barplot

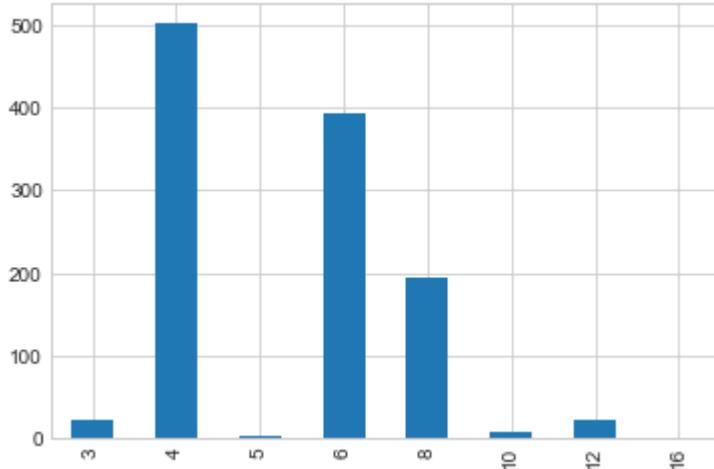
```
cars["Cylinders"].value_counts().plot(kind="bar")
```



Frequency barplot

To order the bars by their natural order, we can modify how `value_counts` is computed

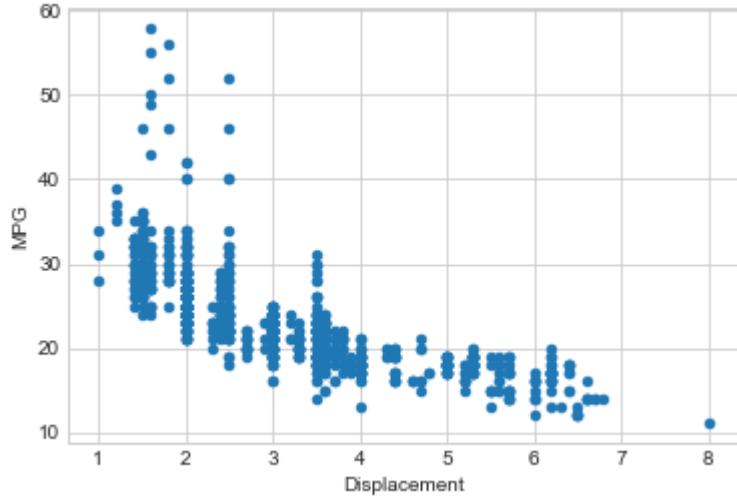
```
cars["Cylinders"].value_counts(sort=False).plot(kind="bar")
```



Visualizing bivariate relationships

Scatter plot (2 continuous variables)

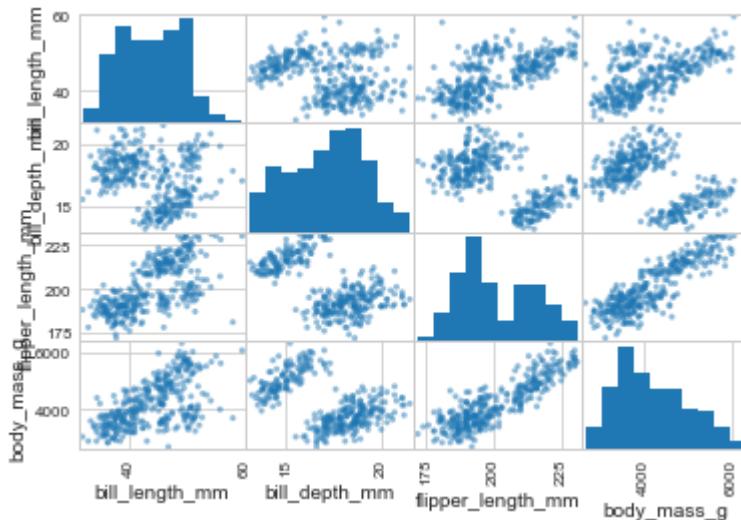
```
cars.plot(x="Displacement", y="MPG", kind="scatter")
```



Scatterplot matrix

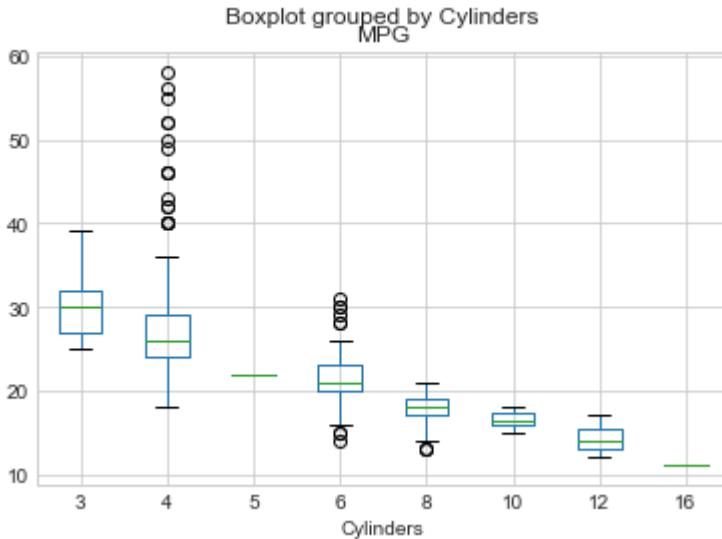
You can look at inter-relationships between all the continuous variables in a dataset using a scatterplot matrix. We'll use the *penguins* data, which we can load through the [seaborn](#) package.

```
penguins = sns.load_dataset("penguins")
pd.plotting.scatter_matrix(penguins)
```



Box plots (continuous x categorical)

```
cars.boxplot(column="MPG", by="Cylinders")
```



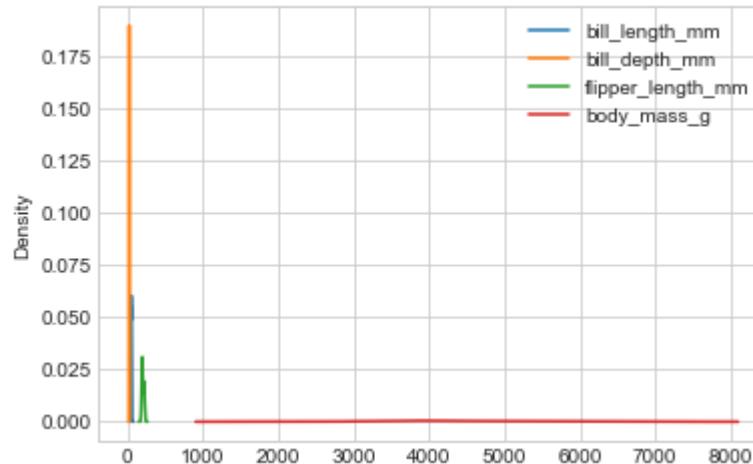
An aside about pandas plotting

Plotting several columns in a DataFrame

Because **pandas** was created to deal with sets of time series, the plotting rules were set up to encode each column into a separate visualization

For example, if you want to look at univariate characteristics of all continuous variables in a DataFrame:

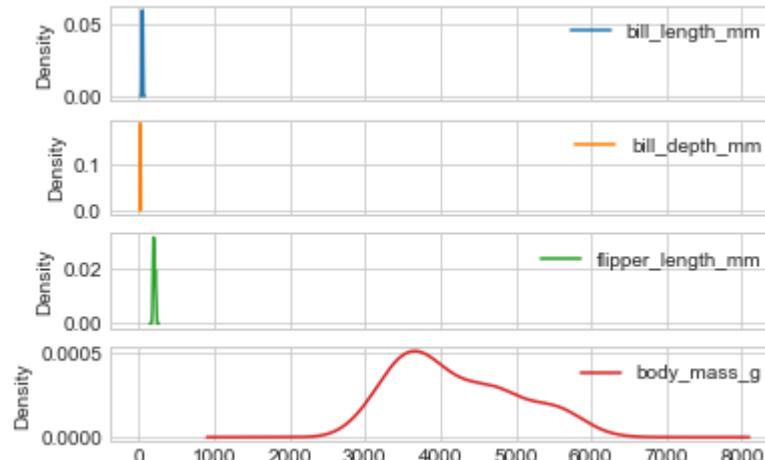
```
penguins.plot(kind="kde")
```



Plotting several columns in a DataFrame

We'd do better by putting each variable into a separate plot

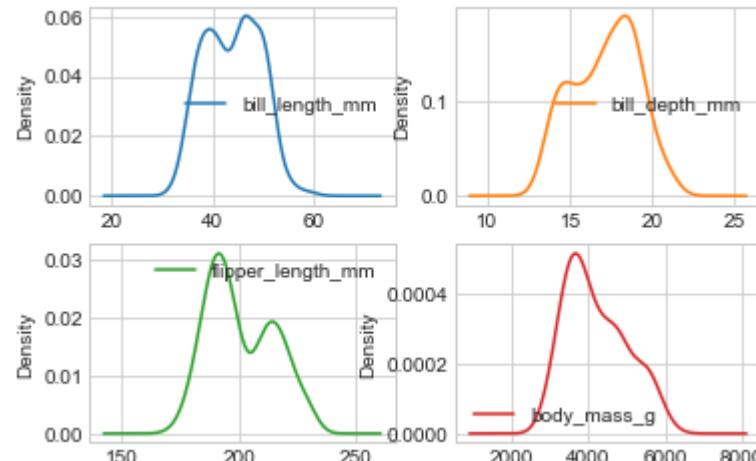
```
penguins.plot(kind="kde", subplots=True)
```



Plotting several columns in a DataFrame

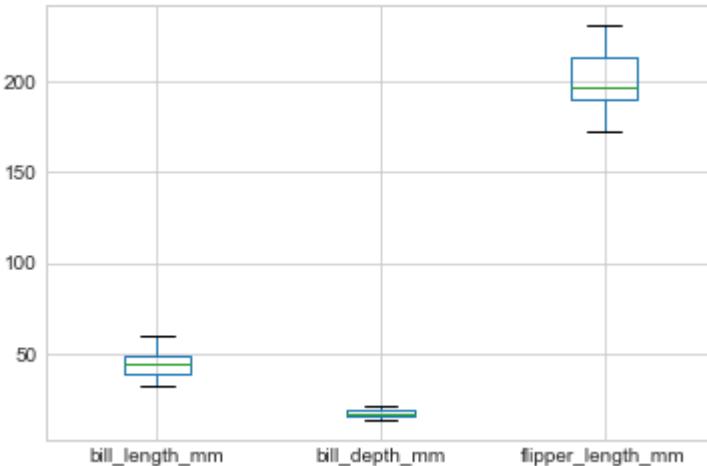
Let's put each subplot on its own scale

```
penguins.plot(kind="kde", subplots=True, sharex=False, layout=(2, 2))
```



Boxplots of several columns

```
penguins[["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]].plot(kind="box")  
plt.show()
```



Plotting against the index of a Series or DataFrame

With time series, `pandas` stores the time aspect in the index of the `Series` or `DataFrame`. So typically, `pandas` plots each variable against the index for encodings like line plots or bar plots

We saw this in the frequency bar plot, where `value_counts` creates a `Series` with unique values as the index and the values as the frequencies

```
freqs = penguins["species"].value_counts()  
freqs.index
```

```
Index(['Adelie', 'Gentoo', 'Chinstrap'], dtype='object')
```

```
freqs.values
```

```
array([152, 124, 68])
```

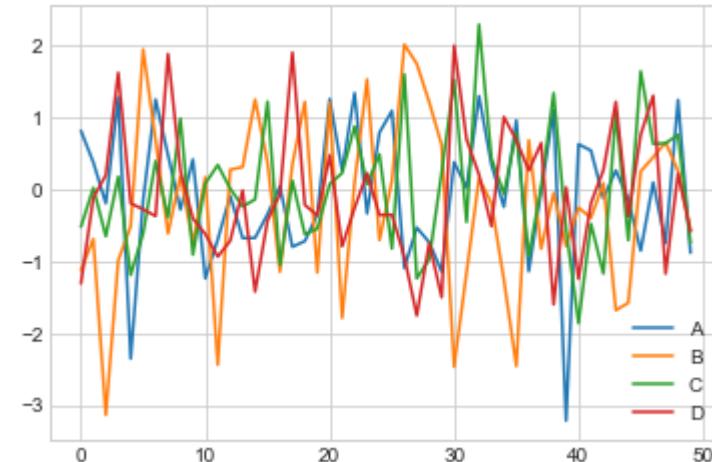
```
penguins["species"].value_counts().plot(kind="bar")  
plt.show()
```

Another example

```
import numpy as np  
n = np.random.randn(5,4)  
D = pd.DataFrame(n, columns=[ "A", "B", "C", "D"  
D.head()
```

	A	B	C	D
0	0.130581	0.269548	-0.448651	-0.797425
1	-0.348450	-0.409104	1.376513	0.747525
2	-1.498935	-0.845907	1.368373	-0.879260
3	-1.502524	0.529177	-0.069634	-0.205654
4	0.061089	1.132279	0.353823	-0.360821

```
D.plot(kind="line")
```



So we can plot multiple variables on a plot (but does it make sense?)

seaborn for statistical visualization

seaborn

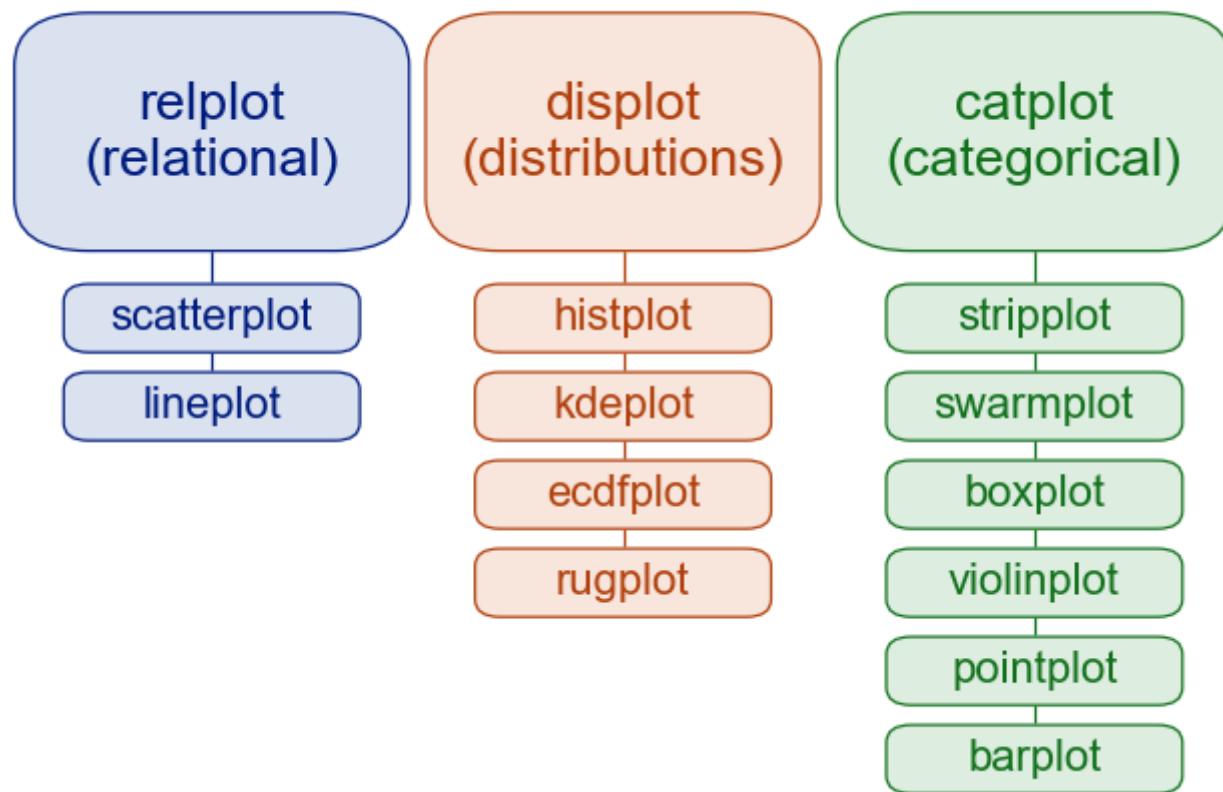
Using [pandas](#), we see basic encodings, basically just `x` and `y`

[seaborn](#) gets us a richer set of visual encodings, using relatively straightforward code.

We'll see later how we'd do a similar plot using more granular code from [matplotlib](#)

seaborn

The main classes of figures created using [seaborn](#):



seaborn

`seaborn` allows us to make it easier to create

- *facets*, i.e., subplots based on unique values of column(s) of the `DataFrame`
- *overlays*, i.e., plots where we can encode unique values of column(s) of a `DataFrame` on the same plot

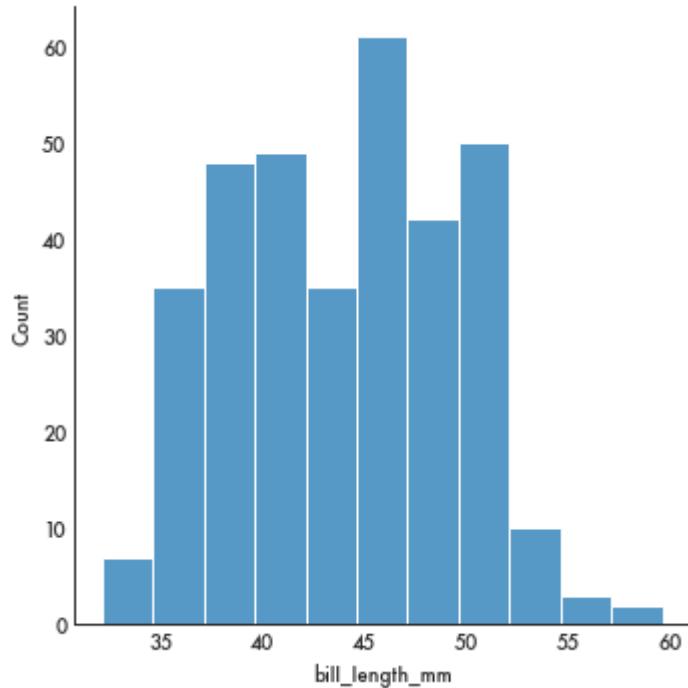
seaborn

`seaborn` code follows a general paradigm, where we

- start with a `DataFrame`
- specify the column(s) we want to plot
- specify the column(s) we want to either facet or overlay
- specify the encodings for the different data elements

Histograms

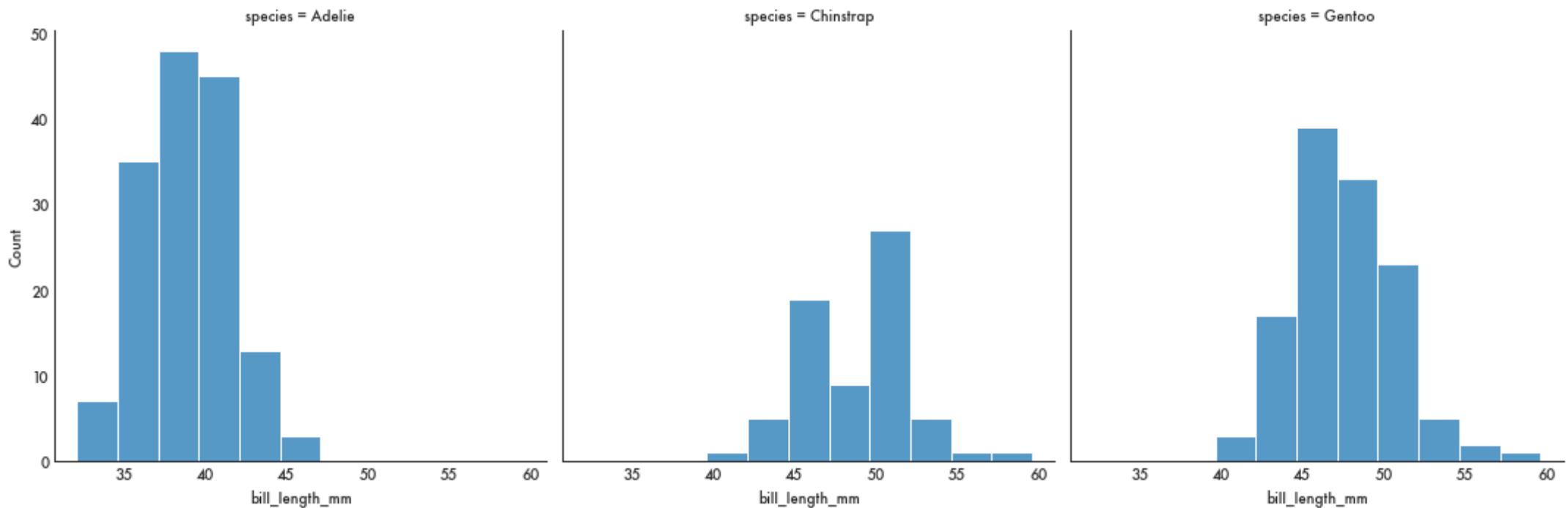
```
sns.displot(  
    data=penguins, # DataFrame  
    x="bill_length_mm", # columns to encode  
    kind="hist", # Type of encoding  
)
```



```
sns.displot(  
    data=penguins, # DataFrame  
    x="bill_length_mm", # columns to encode  
    kind="hist", # Type of encoding  
    hue="species", # Encode species as colors  
)
```

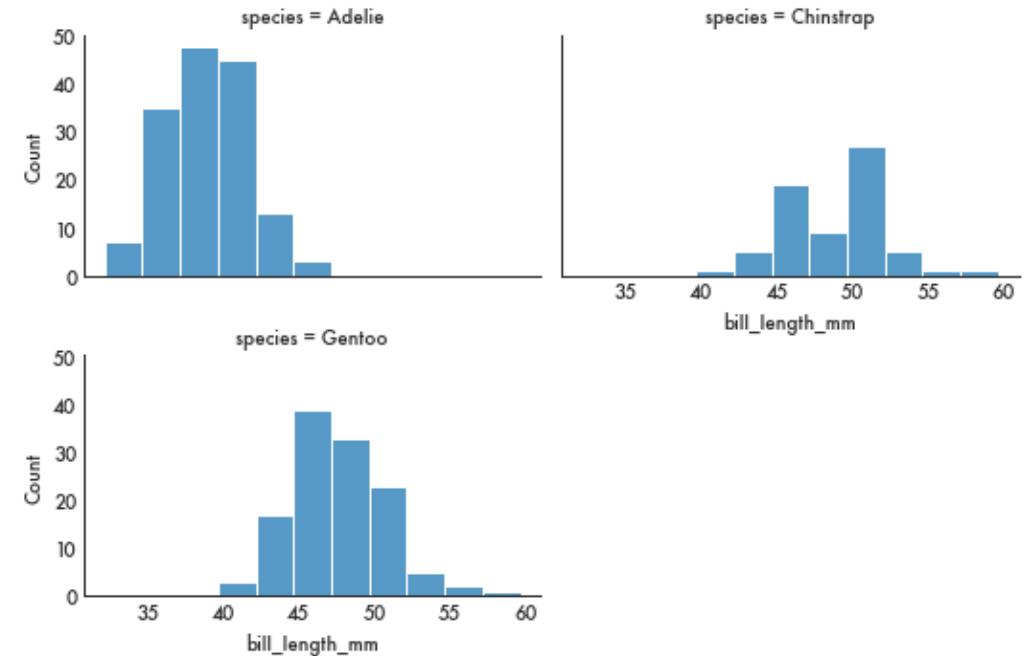
Histograms

```
sns.displot(  
    data=penguins, # DataFrame  
    x="bill_length_mm", # columns to encode  
    kind="hist", # Type of encoding  
    col="species", # Encode species as facets, one per column  
)
```



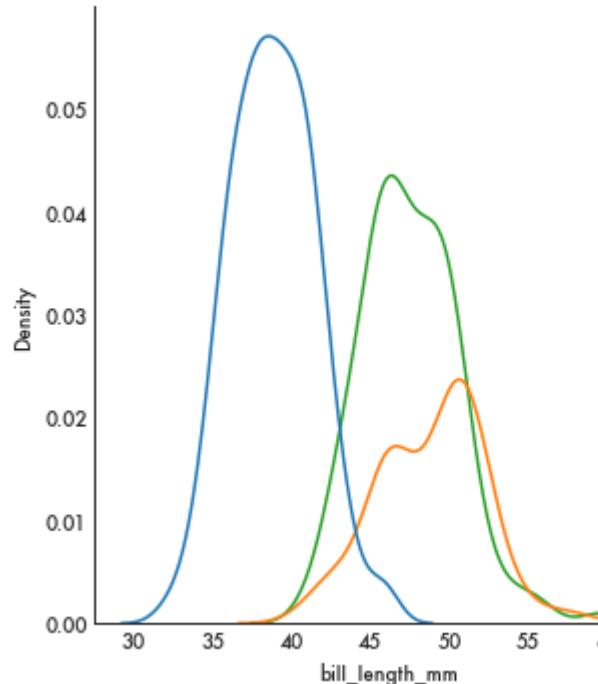
Histograms

```
sns.displot(  
    data=penguins, # DataFrame  
    x="bill_length_mm", # columns to encodeS  
    kind="hist", # Type of encoding  
    col="species", # Encode species as facets  
    col_wrap=2,  
    height=2.5,  
    aspect=1.5,  
)
```



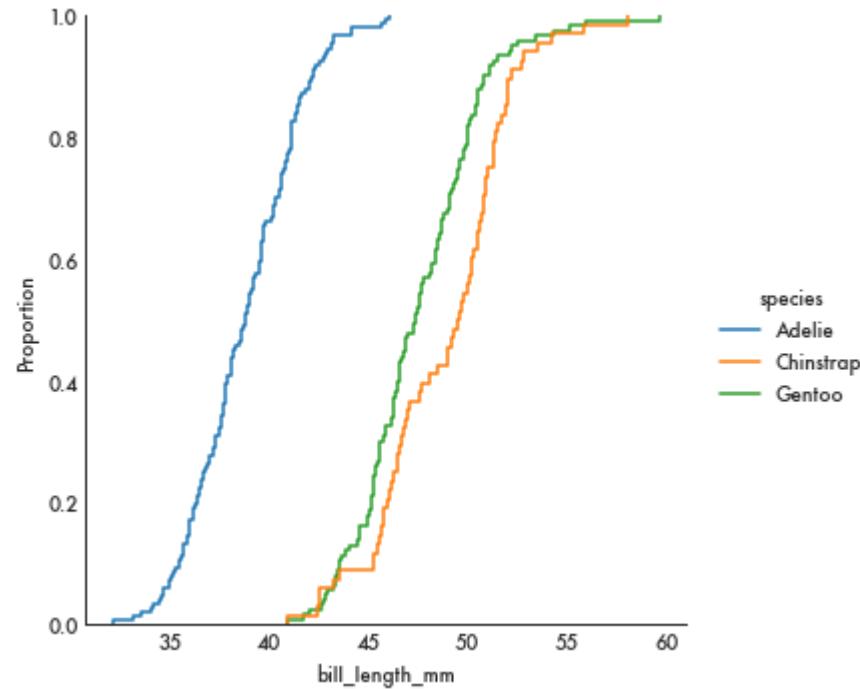
Density plot

```
sns.displot(data=penguins, x="bill_length_mm", hue="species", kind="kde")
```



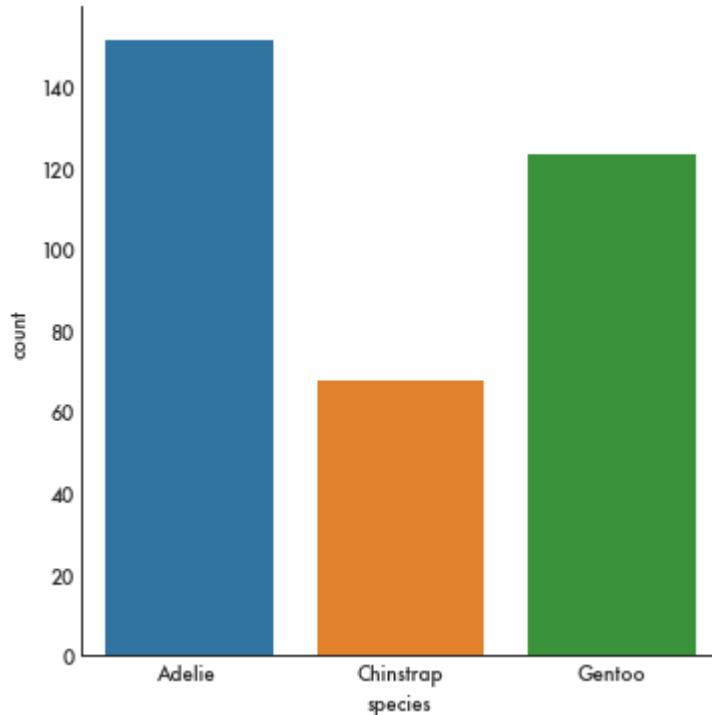
Empirical cumulative distribution plots

```
sns.displot(data=penguins, x="bill_length_mm", hue="species", kind="ecdf")  
plt.show()
```



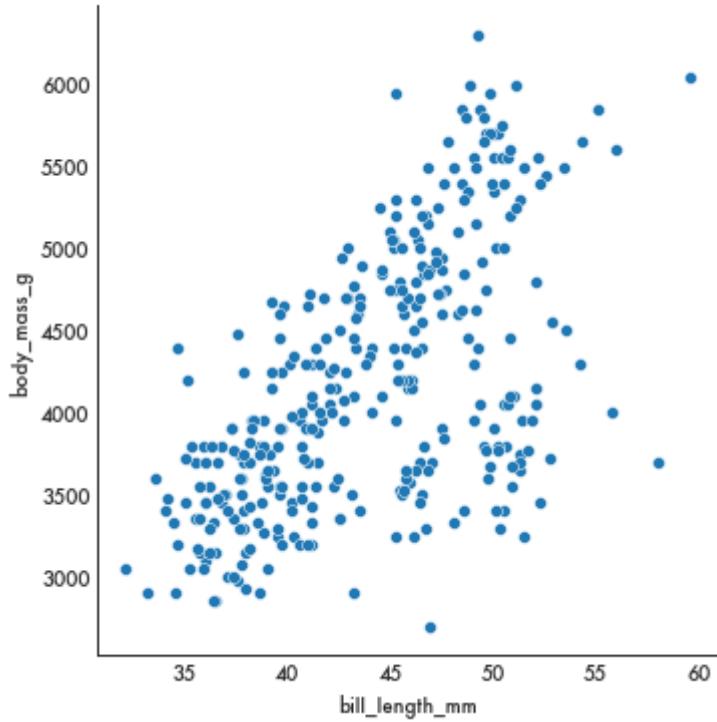
Categorical plots

```
sns.catplot(data=penguins, x="species", kind="count")
```

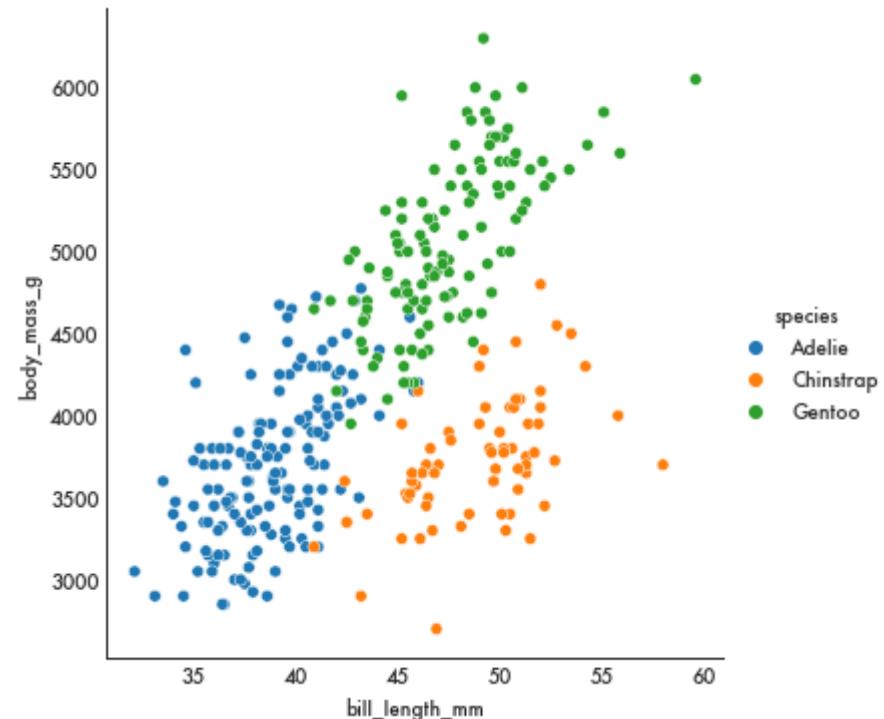


Relating two continuous variables

```
sns.relplot(data=penguins,  
            x="bill_length_mm",  
            y="body_mass_g")
```

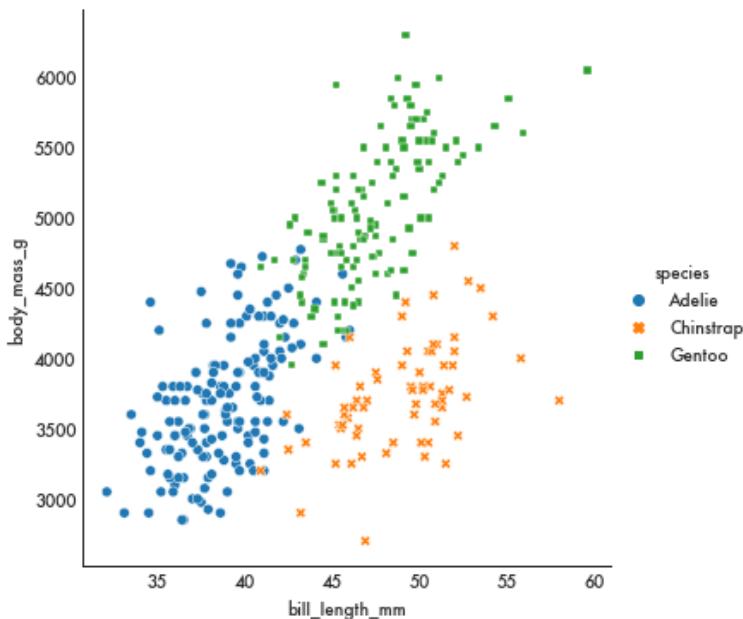


```
sns.relplot(data=penguins,  
            x="bill_length_mm",  
            y="body_mass_g",  
            hue="species")
```

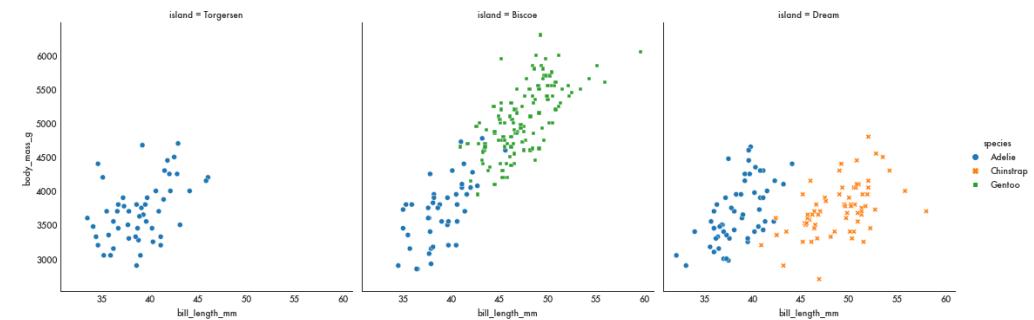


Relating two continuous variables

```
sns.relplot(  
    data=penguins,  
    x="bill_length_mm",  
    y="body_mass_g",  
    hue="species",  
    style="species",  
)
```

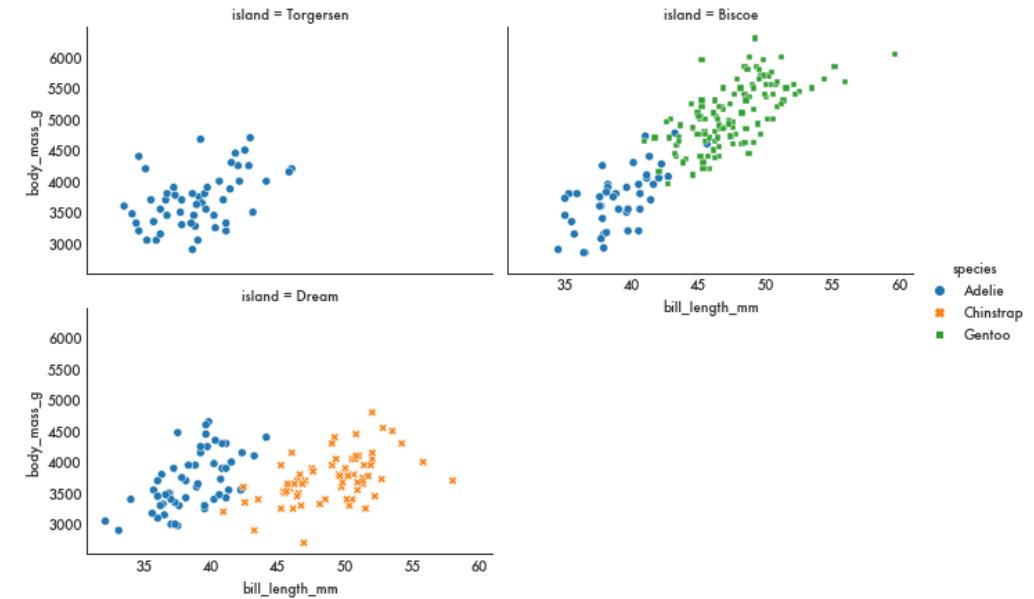


```
sns.relplot(  
    data=penguins,  
    x="bill_length_mm",  
    y="body_mass_g",  
    hue="species",  
    style="species",  
    col="island",  
)
```



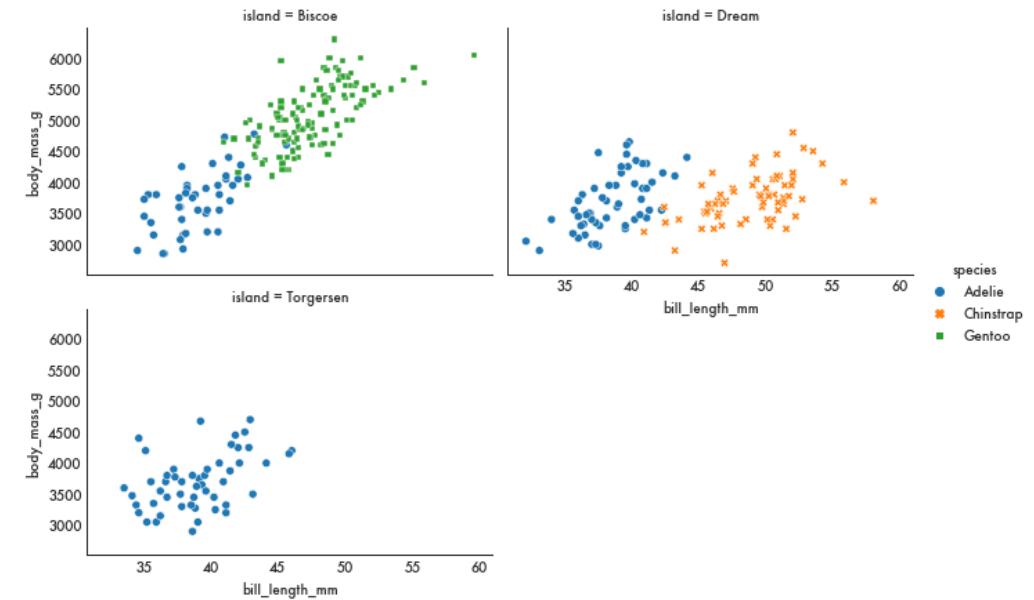
Relating two continuous variables

```
sns.relplot(  
    data=penguins,  
    x="bill_length_mm",  
    y="body_mass_g",  
    hue="species",  
    style="species",  
    col="island",  
    col_wrap=2,  
)
```



Relating two continuous variables

```
sns.relplot(  
    data=penguins,  
    x="bill_length_mm",  
    y="body_mass_g",  
    hue="species",  
    style="species",  
    col="island",  
    col_wrap=2,  
    col_order=["Biscoe", "Dream", "Torgersen"]  
    height=3,  
    aspect=1.5,  
)
```

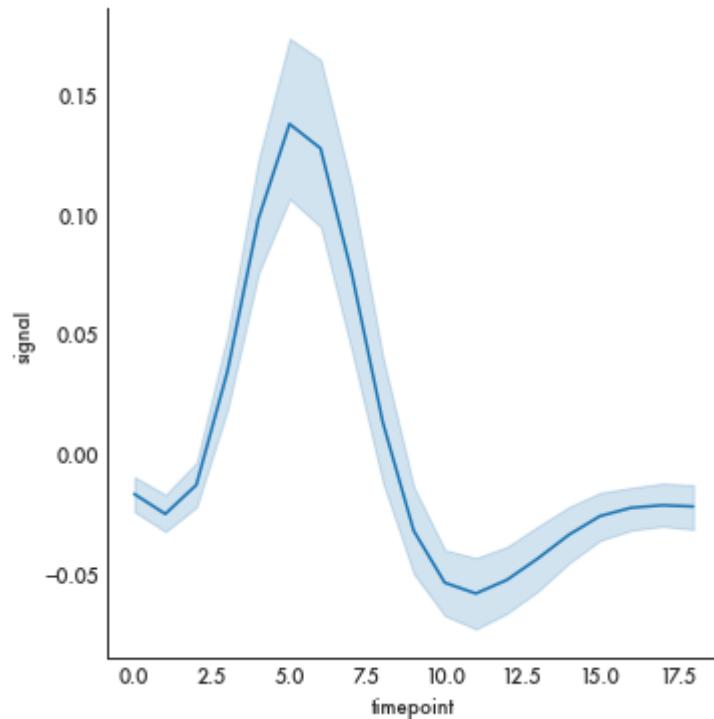


Line plots

```
fmri = sns.load_dataset("fmri")
fmri.head()
```

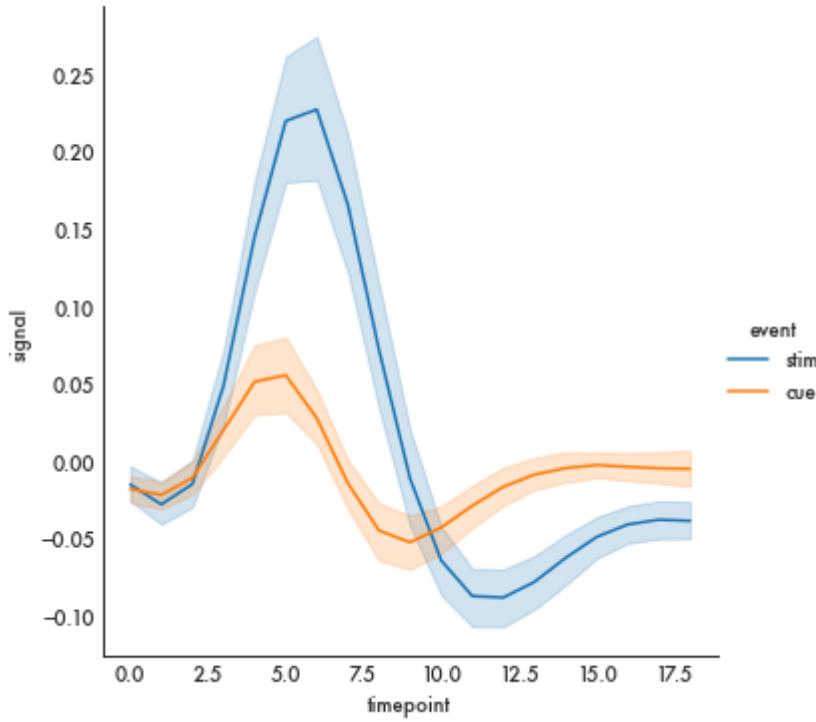
```
   subject  timepoint  event    region      signal
0       s13        18  stim  parietal -0.017552
1       s5         14  stim  parietal -0.080883
2       s12        18  stim  parietal -0.081033
3       s11        18  stim  parietal -0.046134
4       s10        18  stim  parietal -0.037970
```

```
sns.relplot(data=fmri,
             x="timepoint",
             y="signal",
             kind="line")
```

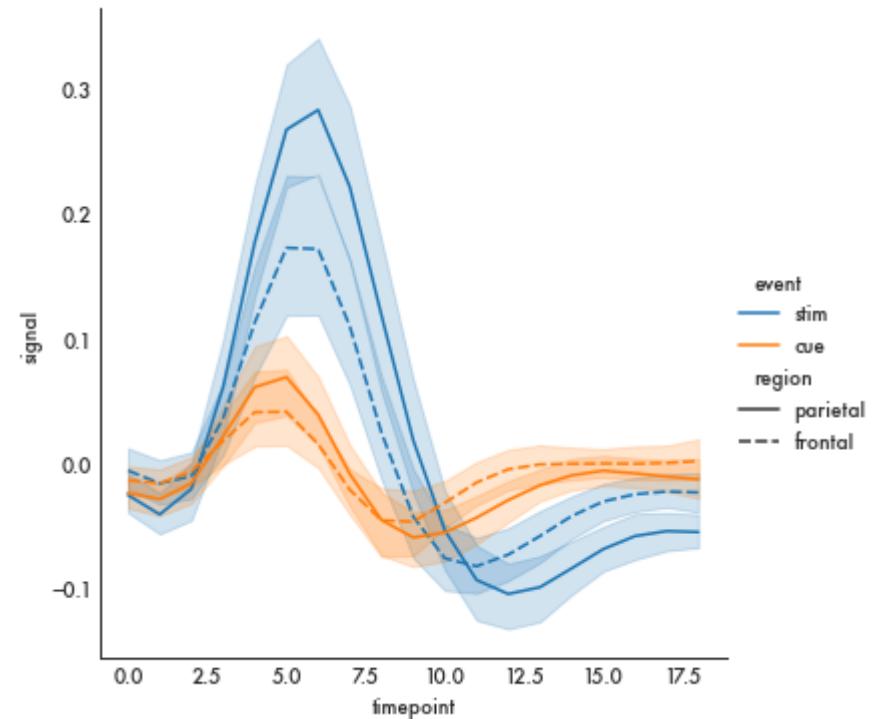


Line plots

```
sns.relplot(data=fmri,  
            x="timepoint", y="signal",  
            kind="line", hue="event")
```

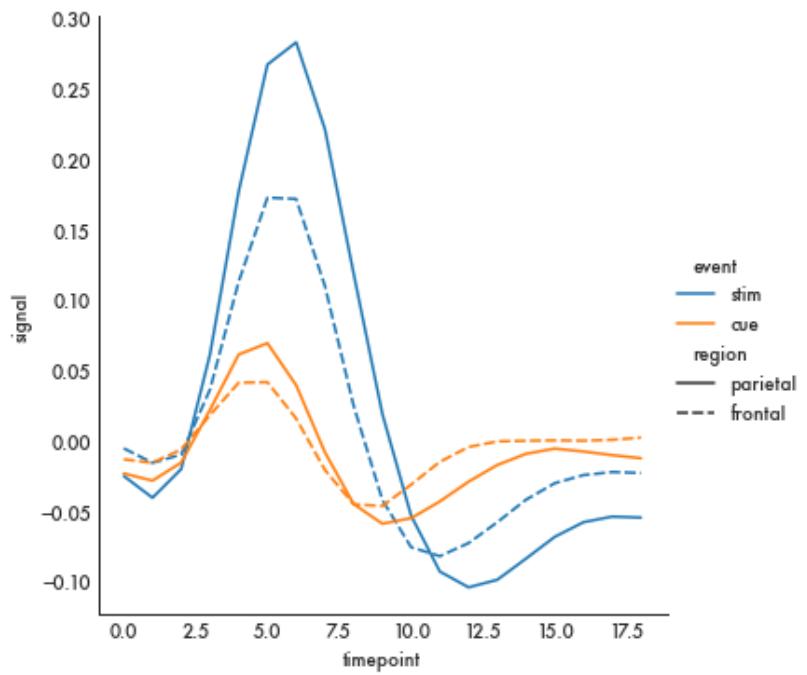


```
sns.relplot(  
    data=fmri, x="timepoint", y="signal",  
    kind="line", hue="event", style="region"  
)
```



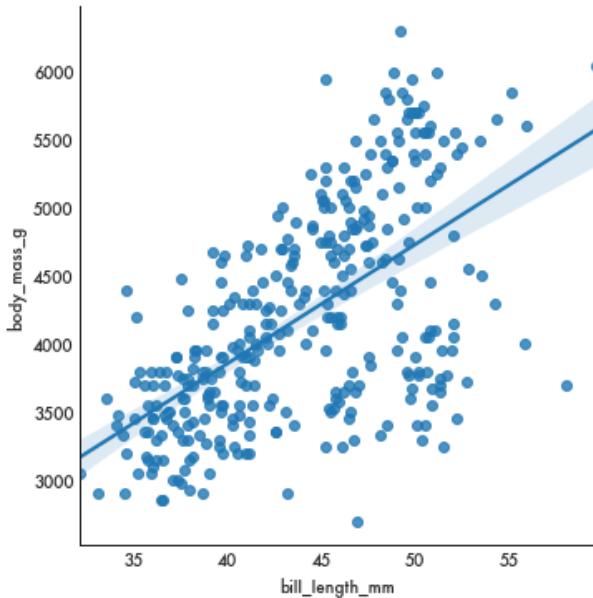
Line plots

```
sns.relplot(  
    data=fmri,  
    x="timepoint", y="signal",  
    kind="line", hue="event",  
    style="region", ci=None,  
)
```

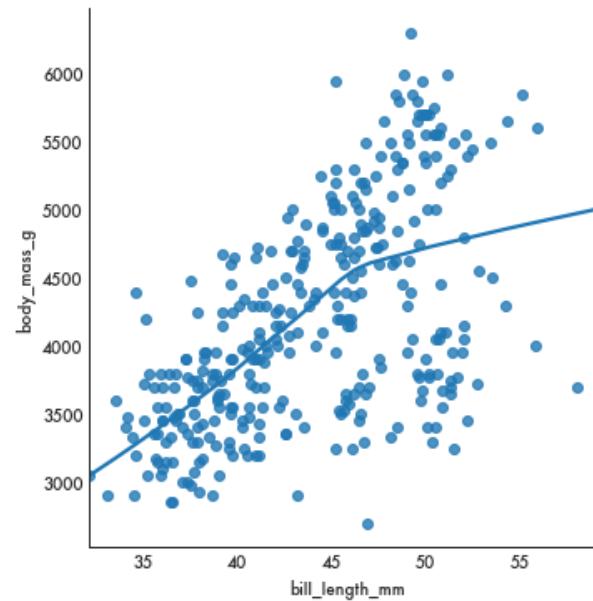


Regression lines

```
sns.lmplot(  
    data=penguins,  
    x="bill_length_mm", y="body_mass_g",  
)
```

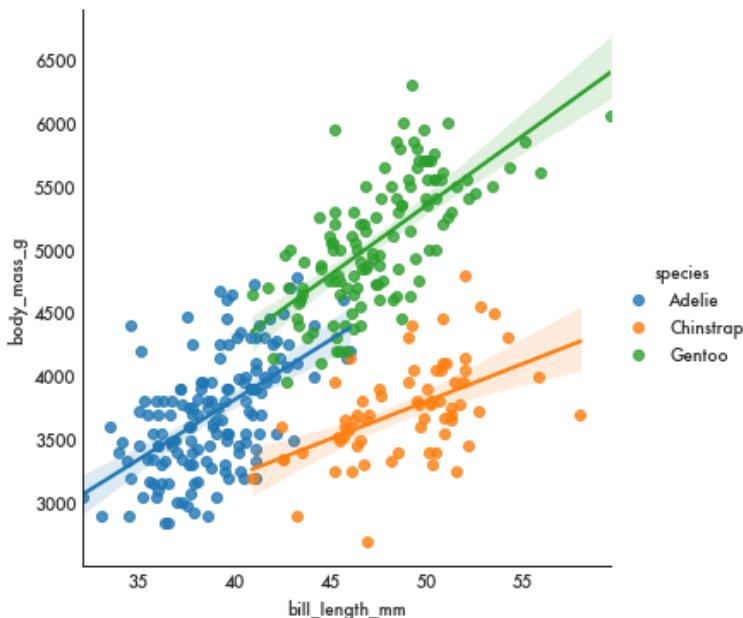


```
sns.lmplot(  
    data=penguins,  
    x="bill_length_mm", y="body_mass_g",  
    lowess=True,  
)
```



Regression lines

```
sns.lmplot(  
    data=penguins,  
    x="bill_length_mm", y="body_mass_g",  
    hue="species",  
)
```

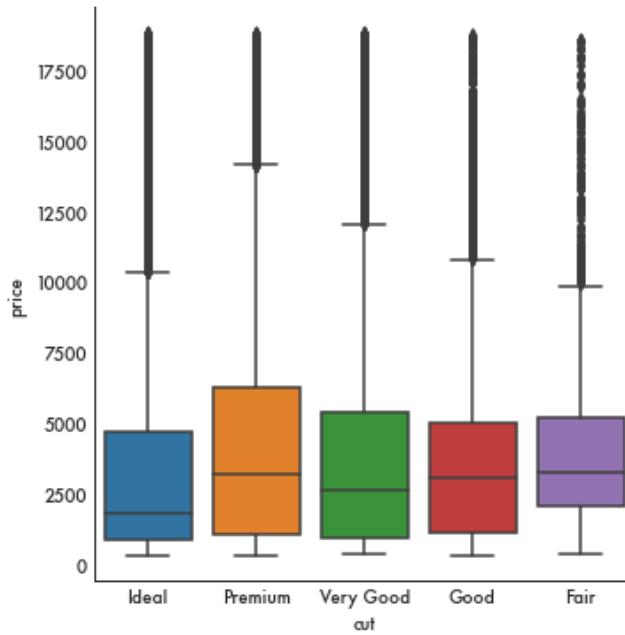


Categorical plots

```
diamonds = sns.load_dataset("diamonds")  
diamonds.shape
```

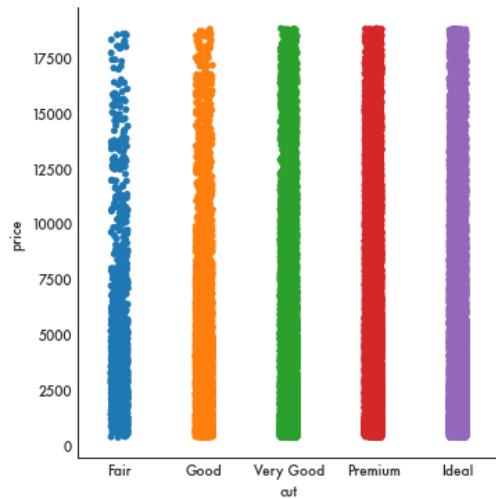
```
(53940, 10)
```

```
sns.catplot(data=diamonds, x="cut", y="price", kind="box")
```

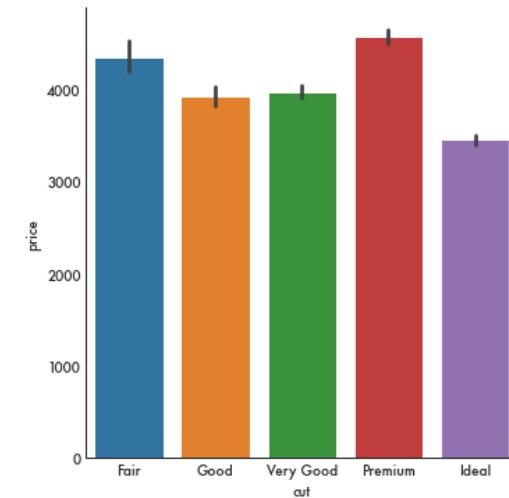


Categorical plots

```
cat_order = ["Fair", "Good", "Very Good",
             "Premium", "Ideal"]
sns.catplot(
    data=diamonds,
    x="cut", y="price",
    kind="strip",
    order=cat_order,
)
```

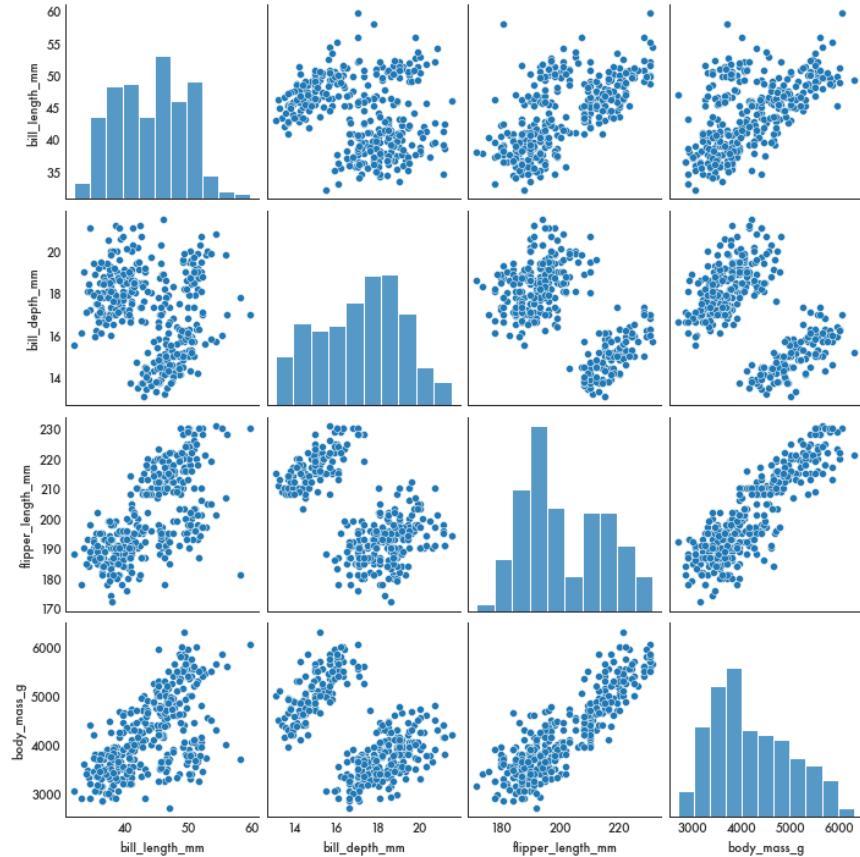


```
cat_order = ["Fair", "Good", "Very Good",
             "Premium", "Ideal"]
sns.catplot(
    data=diamonds,
    x="cut", y="price",
    kind="bar",
    order=cat_order,
)
```



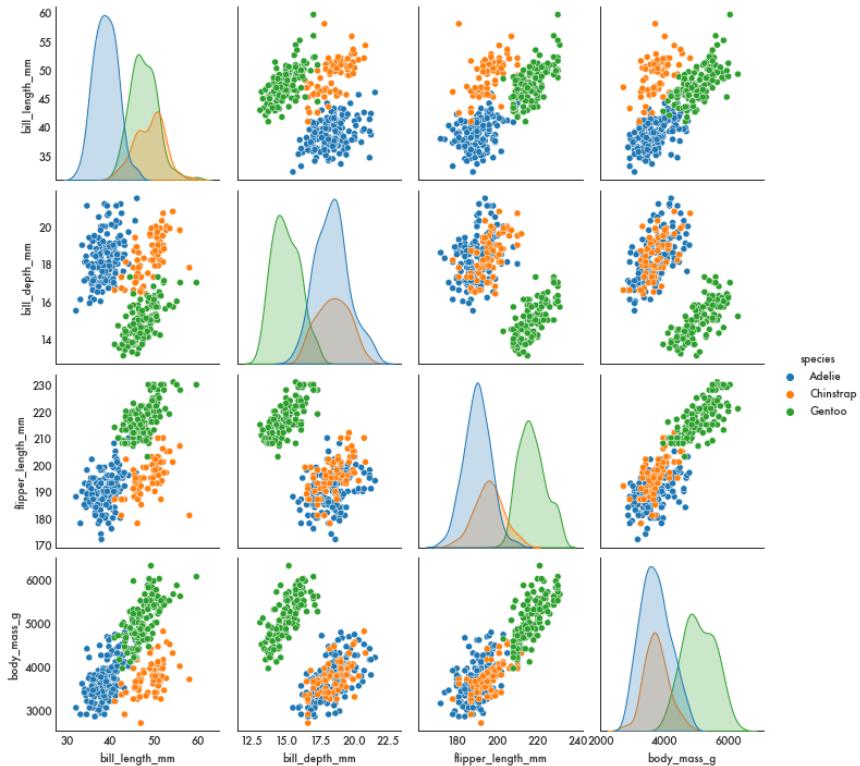
Pair plot

```
sns.pairplot(data=penguins)
```



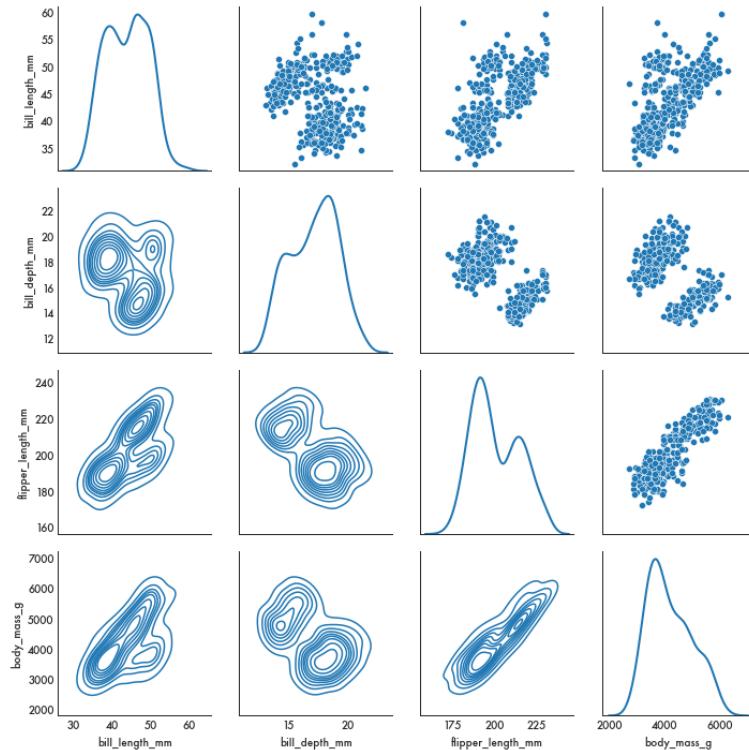
Pair plot

```
sns.pairplot(data=penguins, hue="species")
```



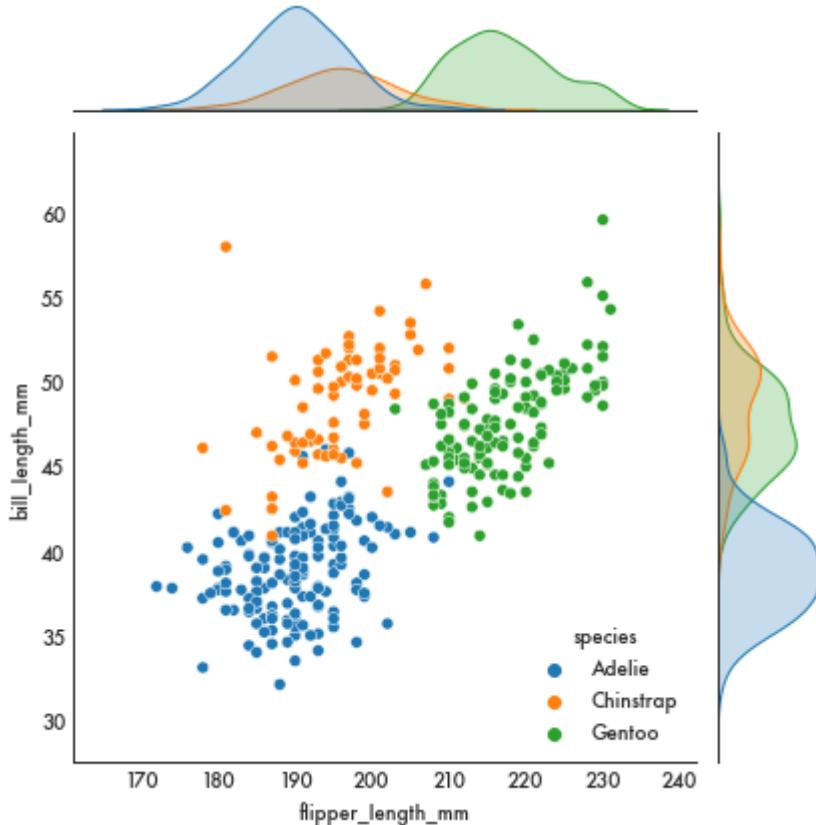
Pair plot

```
g = sns.PairGrid(penguins, diag_sharey=False)
g.map_upper(sns.scatterplot)
g.map_lower(sns.kdeplot)
g.map_diag(sns.kdeplot, lw=2)
```



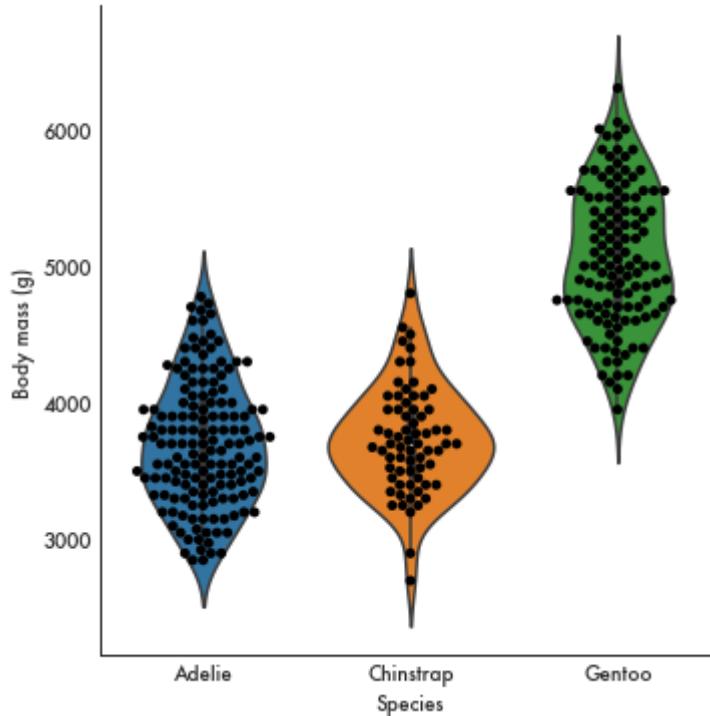
Joint plot

```
sns.jointplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", hue="species")
```



Overlaying aesthetics

```
g = sns.catplot(data=penguins, x="species", y="body_mass_g", kind="violin")
sns.swarmplot(data=penguins, x="species", y="body_mass_g", ax=g.ax, color="black")
g.set_xlabels("Species")
g.set_ylabels("Body mass (g)")
```

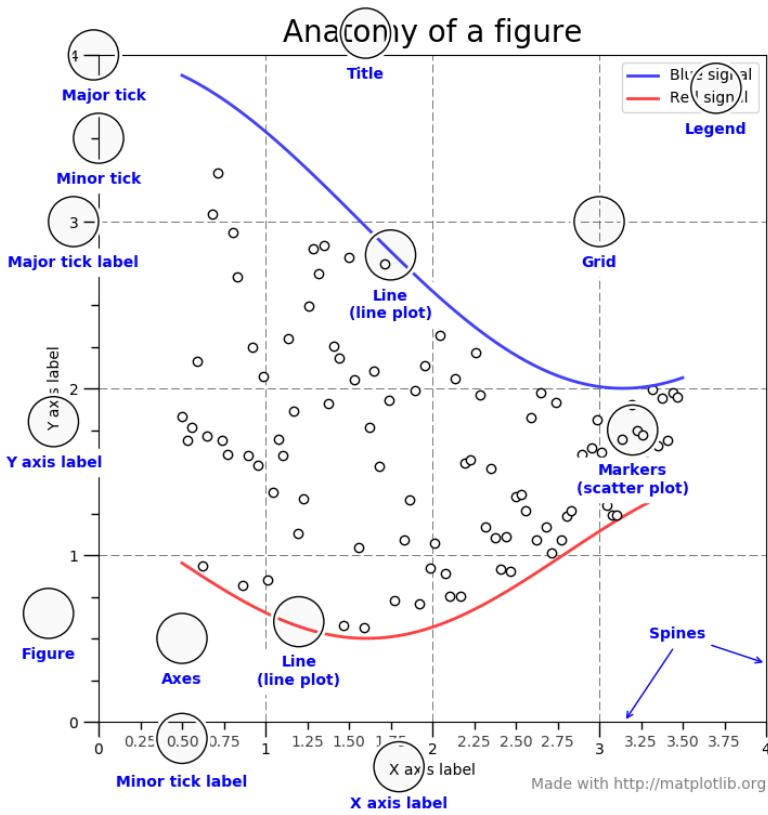


matplotlib

Granular control

matplotlib allows us a lot of granular control of a data visualization.

We can build a visualization from the ground up



Saving your work

`matplotlib`, and, by extension, `pandas` and `seaborn`, has a large number of backend engines that enables one to save their visualizations in several file formats.

```
print(pd.Series(fig.canvas.get_supported_filetypes()))
```

```
eps          Encapsulated Postscript
jpg         Joint Photographic Experts Group
jpeg        Joint Photographic Experts Group
pdf          Portable Document Format
pgf          PGF code for LaTeX
png          Portable Network Graphics
ps           Postscript
raw          Raw RGBA bitmap
rgba         Raw RGBA bitmap
svg          Scalable Vector Graphics
svgz         Scalable Vector Graphics
tif          Tagged Image File Format
tiff         Tagged Image File Format
dtype: object
```

Saving your work

You can save your work using the `plt.savefig` function

```
# This will save the last run figure
plt.savefig("penguins.png", transparent=True, dpi=300)
```

```
!ls -l
```

```
total 3736
-rw-r--r--  1 abhijit  staff   28084 Apr  5 07:54 03_01_data_visualization.Rmd
-rw-r--r--@ 1 abhijit  staff   31217 Apr  5 06:34 03_01_data_visualization.html
-rw-r--r--  1 abhijit  staff  1615051 Apr  5 07:54 03_01_data_visualization.ipynb
-rw-r--r--  1 abhijit  staff   26327 Apr  5 07:54 03_01_data_visualization.py
drwxr-xr-x  4 abhijit  staff    128 Apr  5 06:34 03_01_data_visualization_files[m|m
-rw-r--r--  1 abhijit  staff    324 Apr  5 05:41 _output.yml
lrwxr-xr-x  1 abhijit  staff     44 Apr  5 01:54 data -> /Users/abhijit/Dropbox/teaching/BIOF439/c
drwxr-xr-x  7 abhijit  staff   224 Apr  5 01:24 img
-rw-r--r--  1 abhijit  staff  145047 Apr  5 07:55 penguins.png
```