Statistical data visualizations (static)

BIOF 440

Abhijit Dasgupta

Data visualization

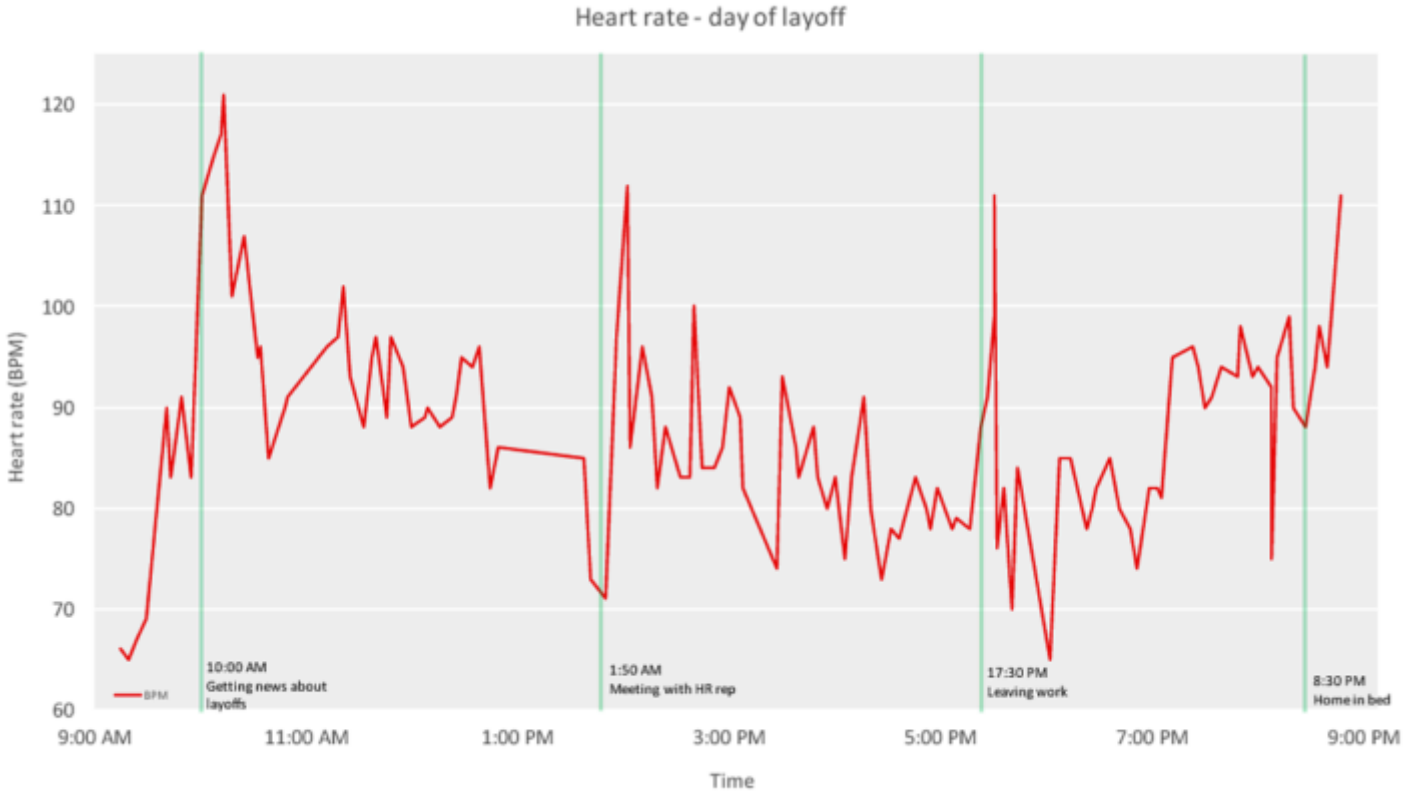
Visualization for analysis

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

Context is important

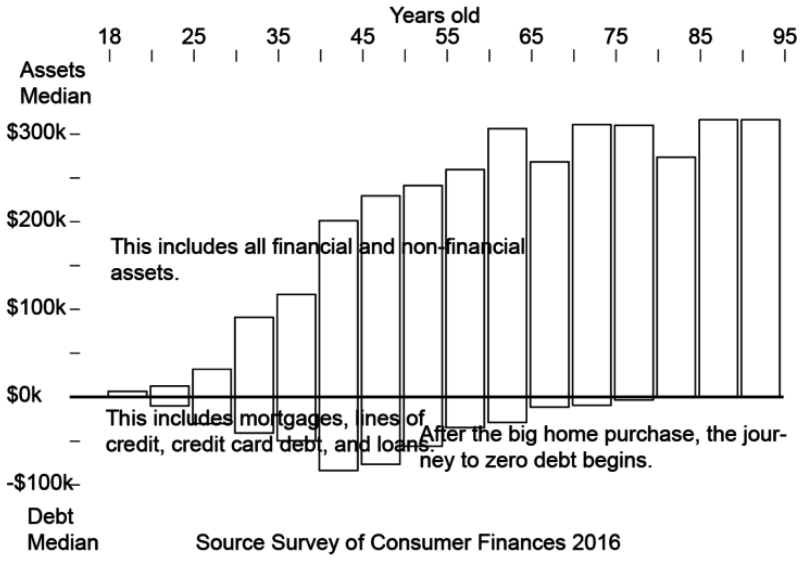
A data visualization should be self-contained and be able to express the context of the data.

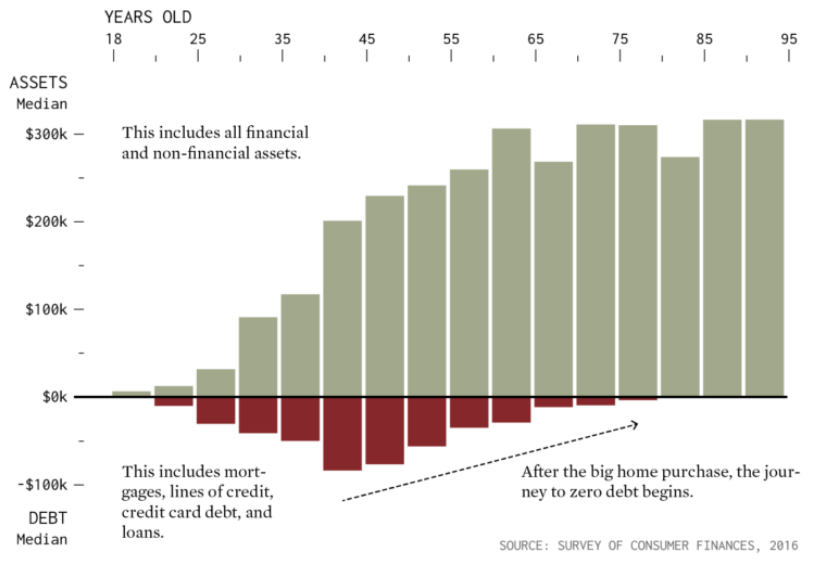
This makes the visualization informative.



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 you 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

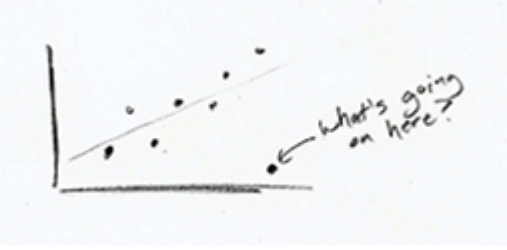
Tufte's Fundamental Principles of Design

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

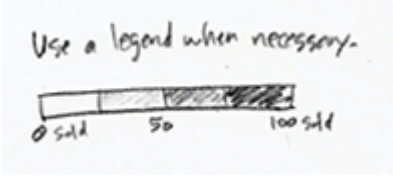
1. Check the data
2. Explain encodings
3. Label axes
4. Include units
5. Keep your geometry in check
6. Include your sources
7. Consider your audience

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

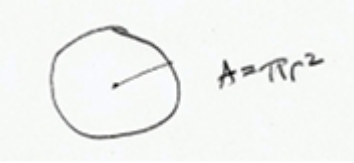
3) Label axes

- 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

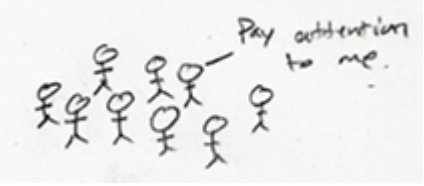


- This is something that is immediately noticeable
- Don't use area to compare two units unless they are an area. An increase in a unit squares the area.
- Tip: size circles and other 2D shapes by area, unless it's a bar chart

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

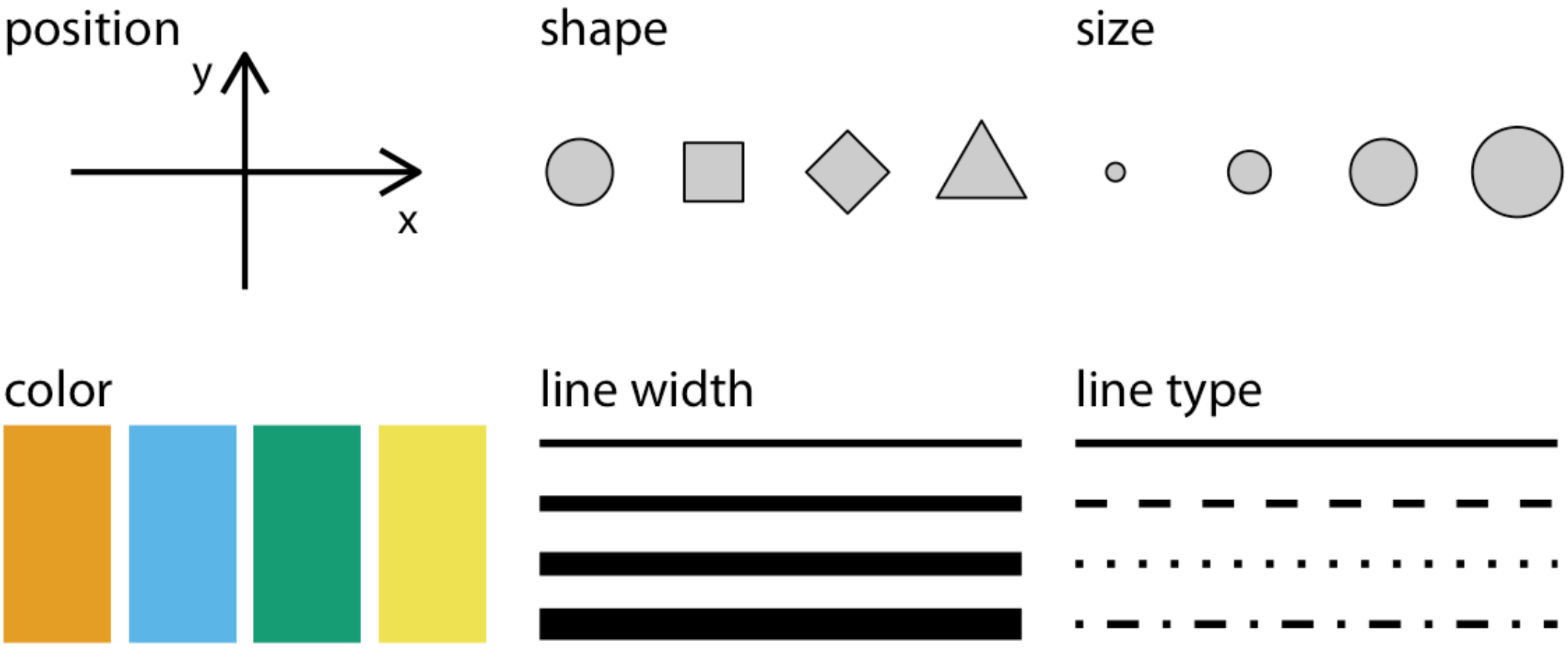
Visual encoding

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



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

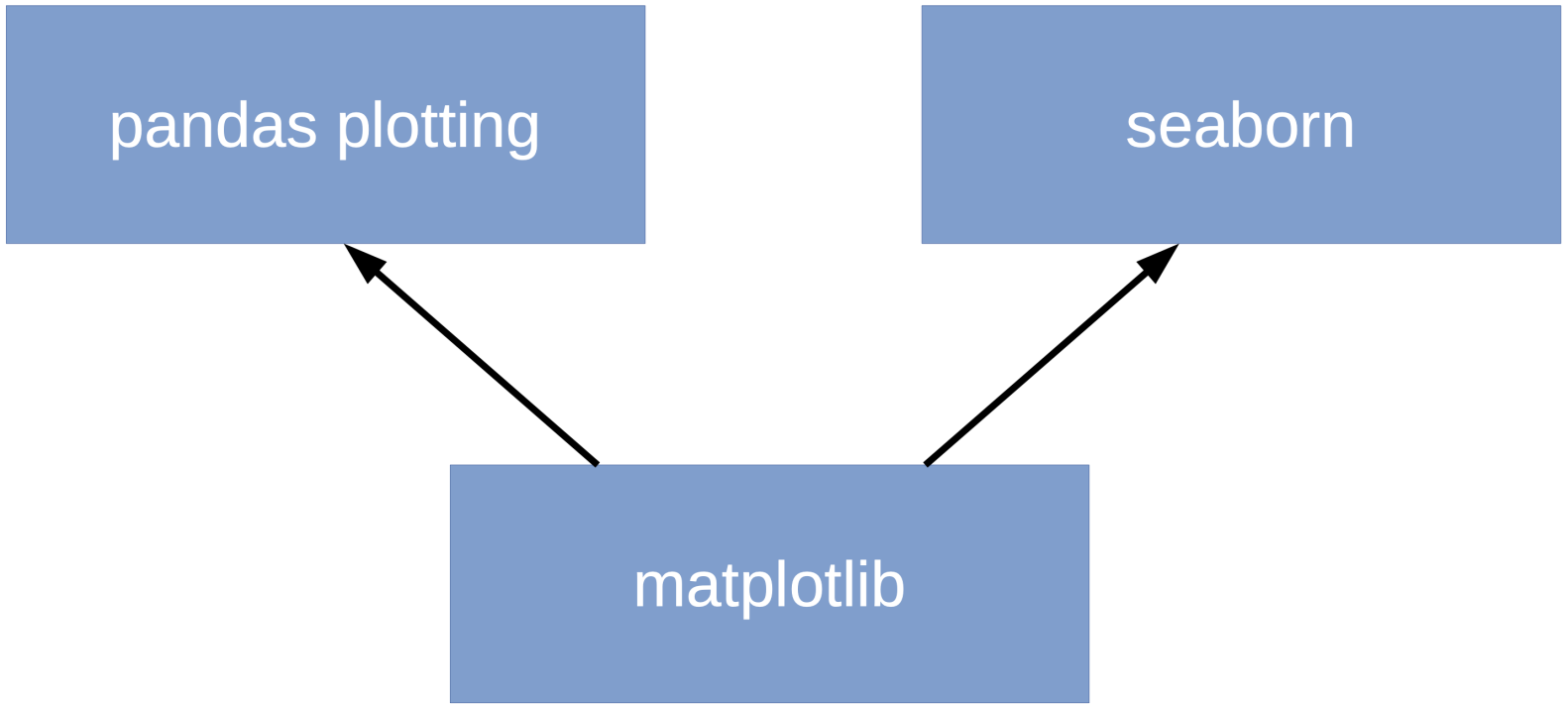
Statistical data visualization

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

%matplotlib inline
plt.style.use("seaborn-whitegrid") # this is a built-in style
mpl.rcParams['figure.figsize'] = (8,6) # Sets default size of graphics in inches
from IPython.display import Image # import images into Jupyter notebooks
```

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

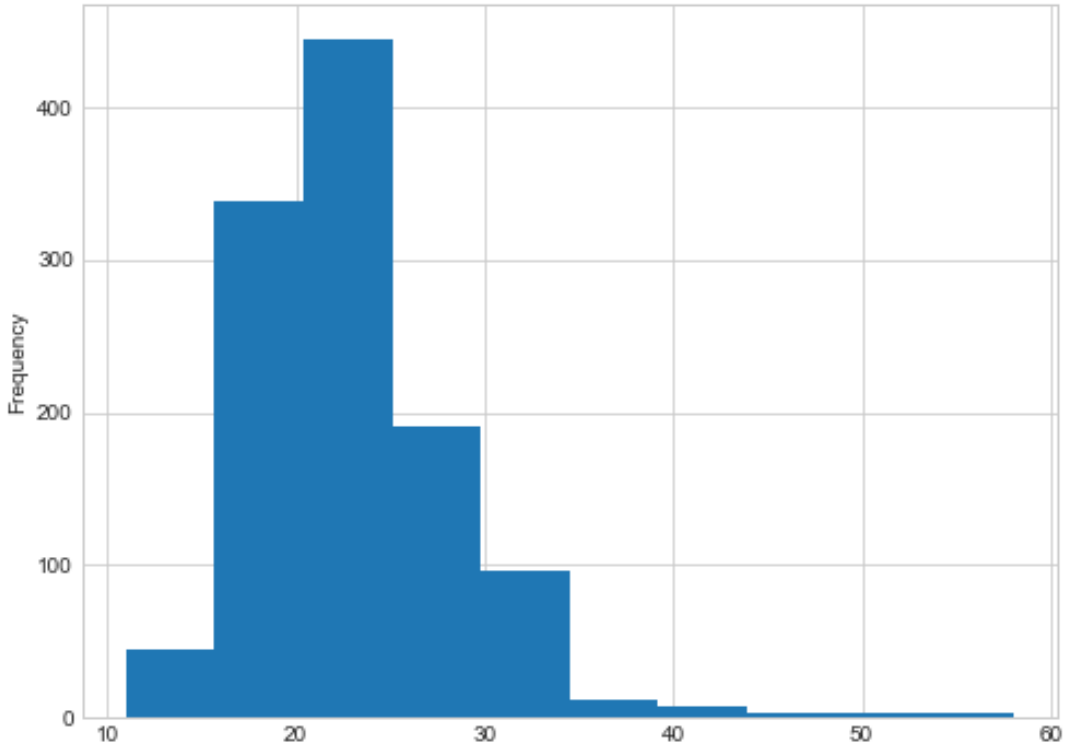
```
cars = pd.read_csv("data/cars2018.csv")
cars.head()
```

	Model	Model Index	Displacement	Cylinders	Gears	Transmission	MPG	Aspiration	Lockup Torque Converter	Drive	Max Ethanol	Recommended Fuel	Intake Valves Per Cyl	Exhaust Valves Per Cyl	Fuel injection
0	Acura NSX	57	3.5	6	9	Manual	21	Turbocharged/Supercharged	Y	All Wheel Drive	10	Premium Unleaded Required	2	2	Direct ignition
1	ALFA ROMEO 4C	410	1.8	4	6	Manual	28	Turbocharged/Supercharged	Y	2-Wheel Drive, Rear	10	Premium Unleaded Required	2	2	Direct ignition
2	Audi R8 AWD	65	5.2	10	7	Manual	17	Naturally Aspirated	Y	All Wheel Drive	15	Premium Unleaded Recommended	2	2	Direct ignition
3	Audi R8 RWD	71	5.2	10	7	Manual	18	Naturally Aspirated	Y	2-Wheel Drive, Rear	15	Premium Unleaded Recommended	2	2	Direct ignition
4	Audi R8 Spyder AWD	66	5.2	10	7	Manual	17	Naturally Aspirated	Y	All Wheel Drive	15	Premium Unleaded Recommended	2	2	Direct ignition

Visualizing one variable (Continuous)

Histogram

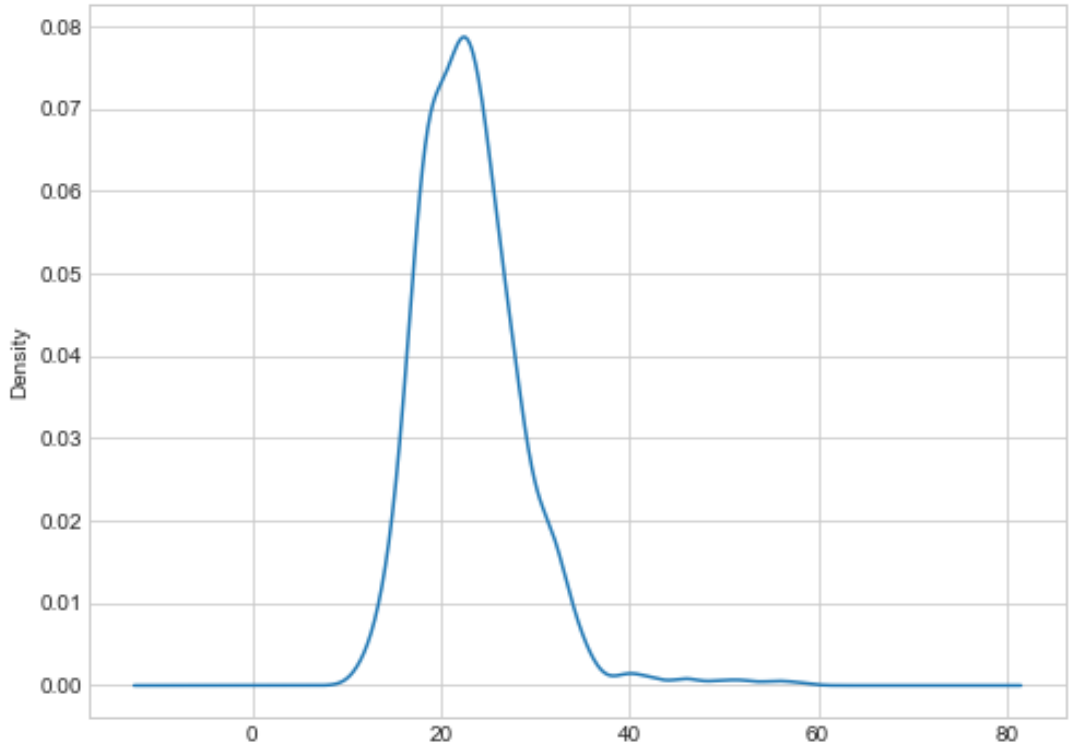
```
cars["MPG"].plot(kind="hist");
```



> Note: Notice the semi-colon after the command. This is to ensure that there is no textual meta-data about the plot that is printed. This is an inheritance from Matlab that has remained.

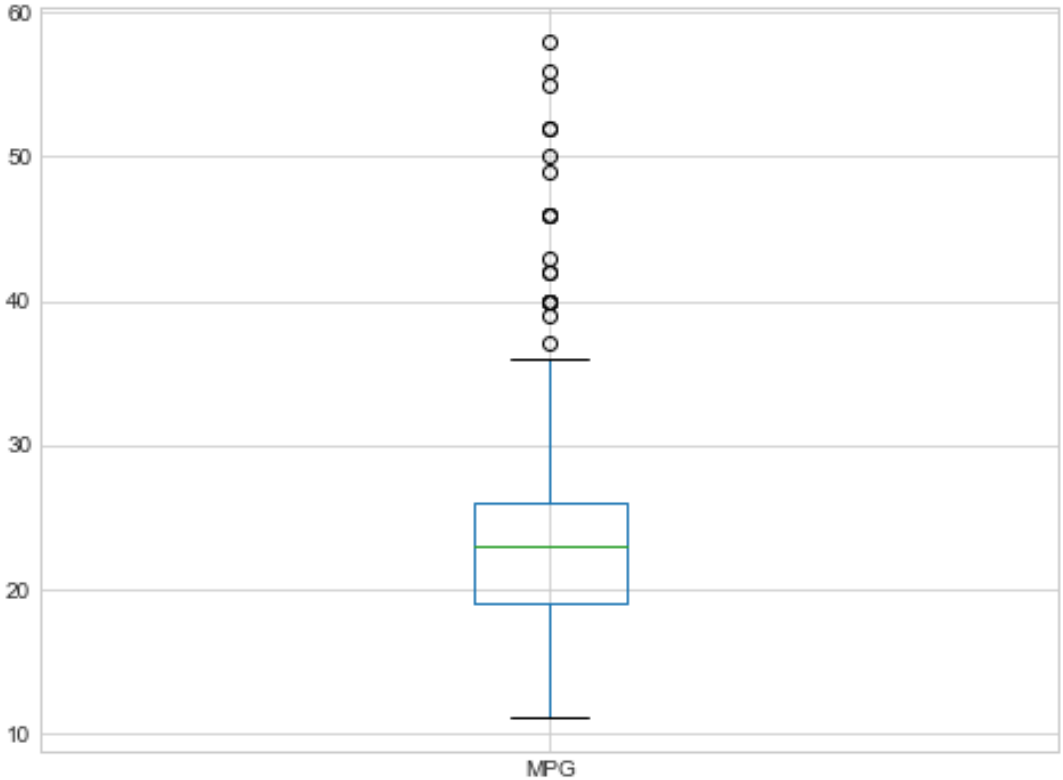
Density plot

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



Box plot

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



Visualizing one variable (categorical)

Frequency barplot

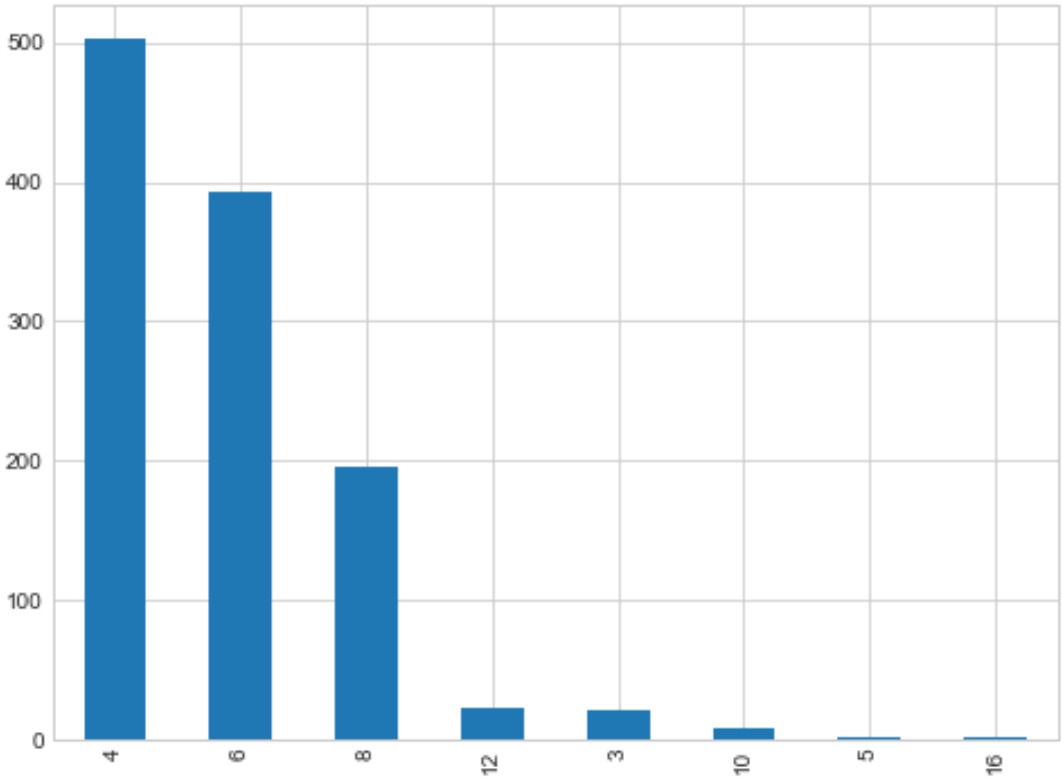
For a frequency barplot, you need to do a bit of data summarization using pandas

```
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

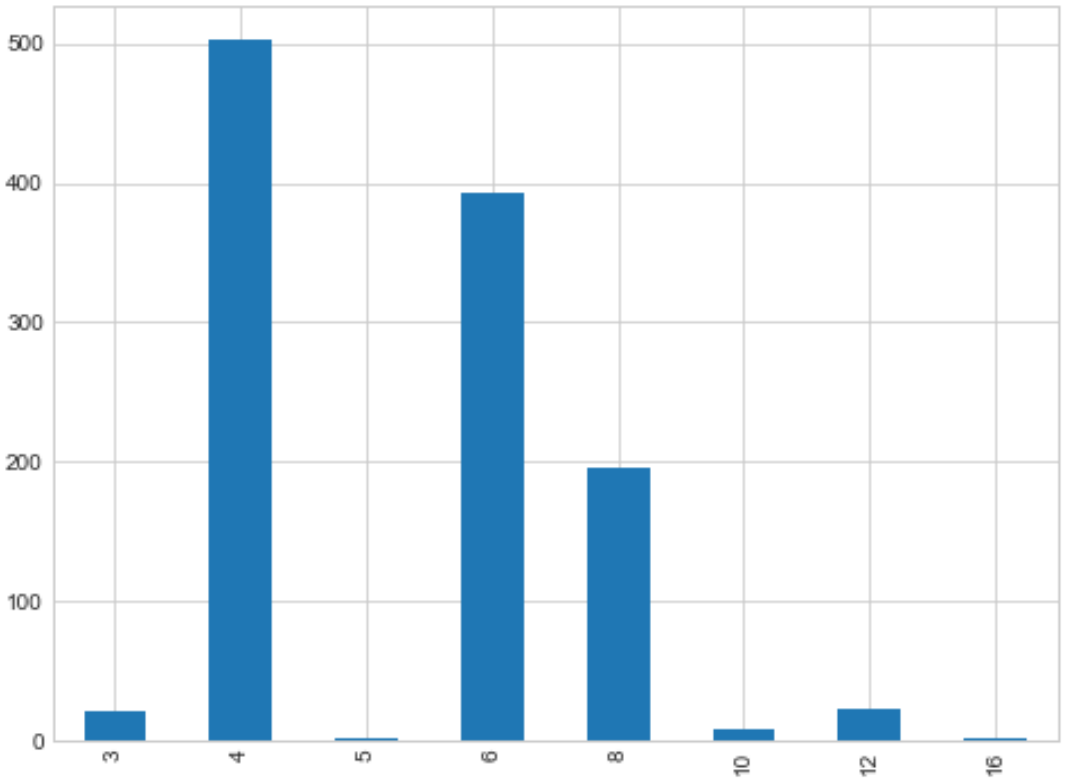
```
cars["Cylinders"].value_counts().plot(kind="bar");
```



Frequency barplot

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

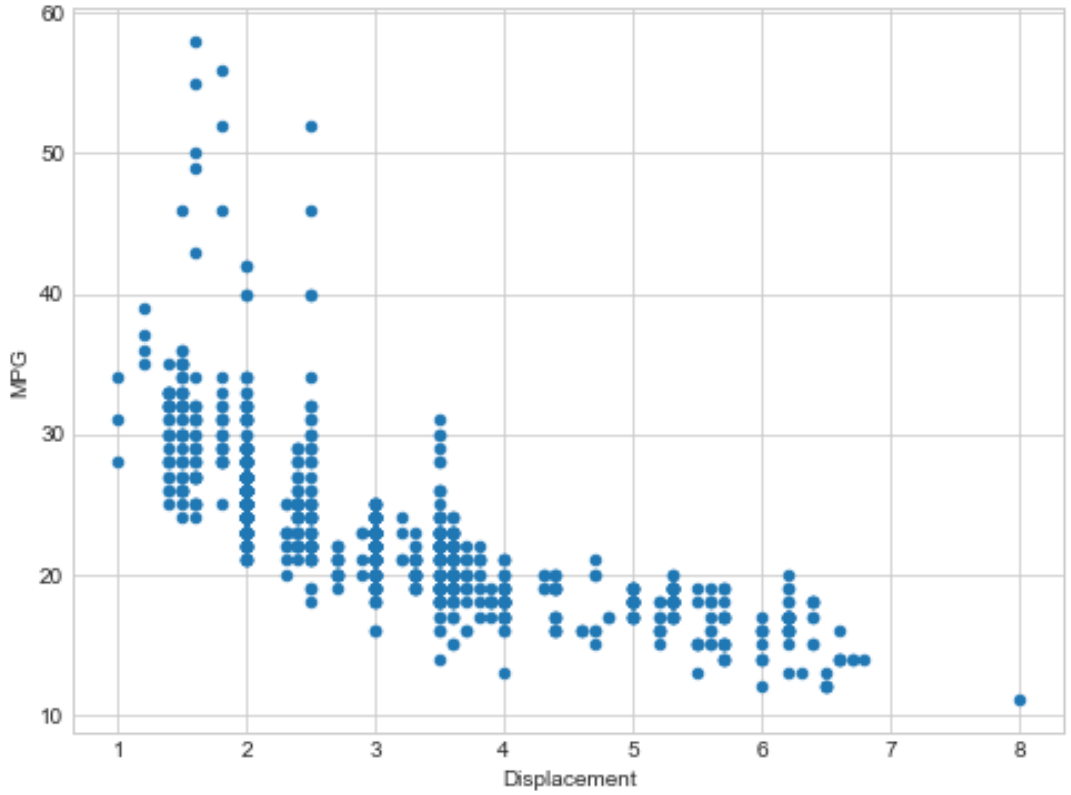
```
cars["Cylinders"].value_counts(sort=False).plot(kind="bar");
```



Visualizing bivariate relationships

Scatter plot (2 continuous variables)

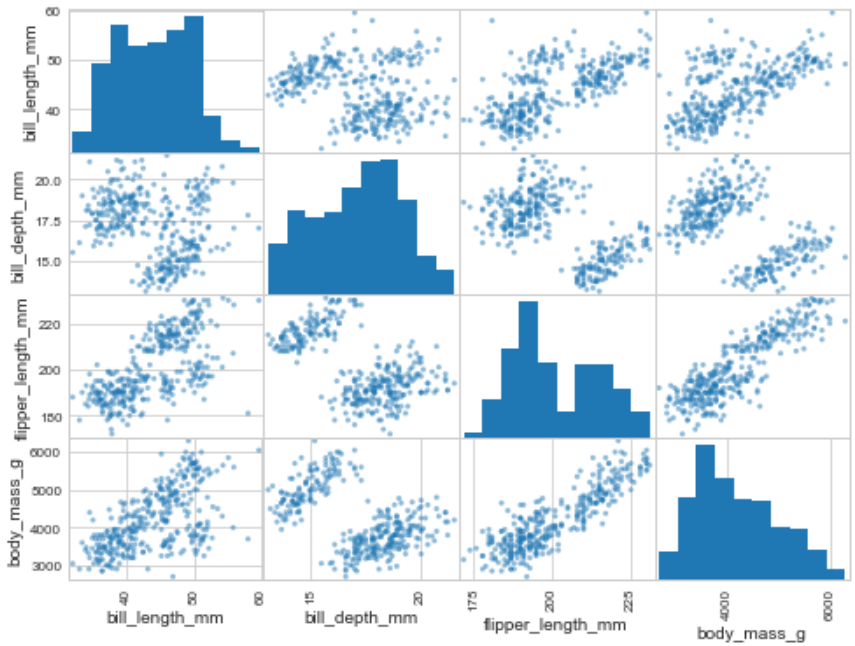
```
cars.plot(x="Displacement", y="MPG", kind="scatter");
```



Scatterplot matrix

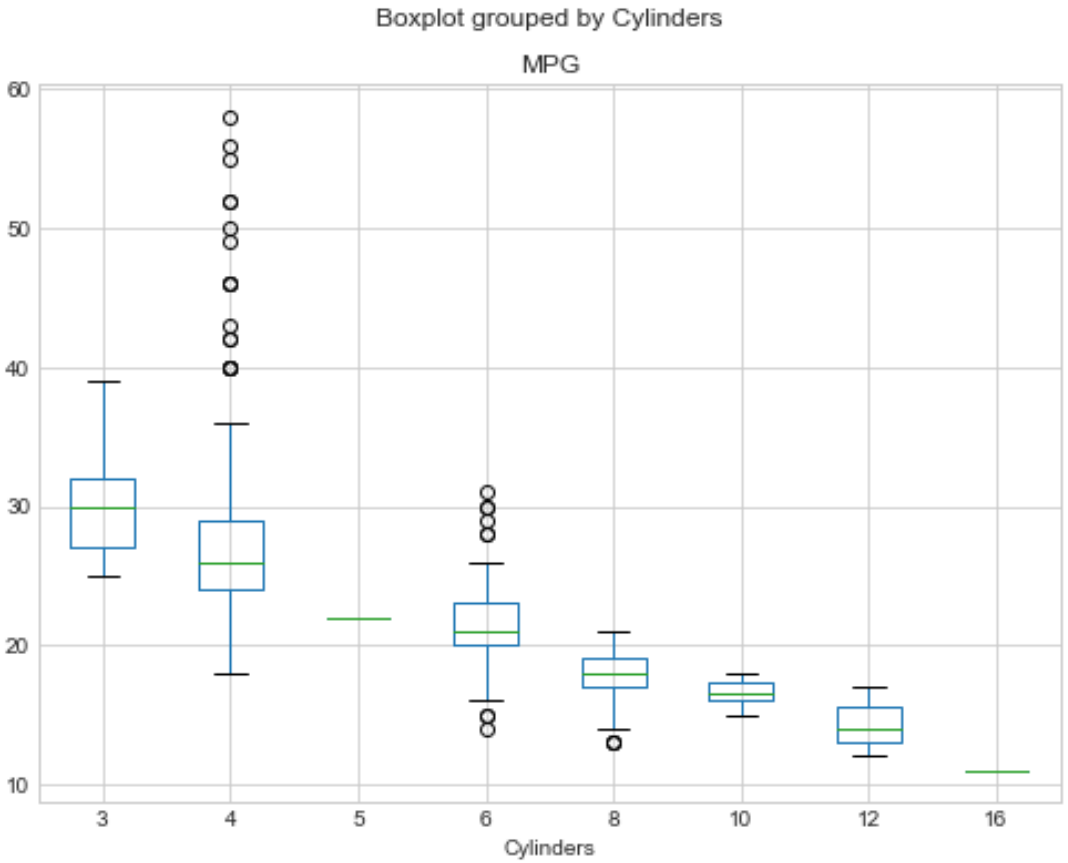
You can look at inter-relationships between all the continuous variables in a dataset using a scatterplot matrix. We'll use the *penguins* data, which we can load through the seaborn package.

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



Box plots (continuous x categorical)

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



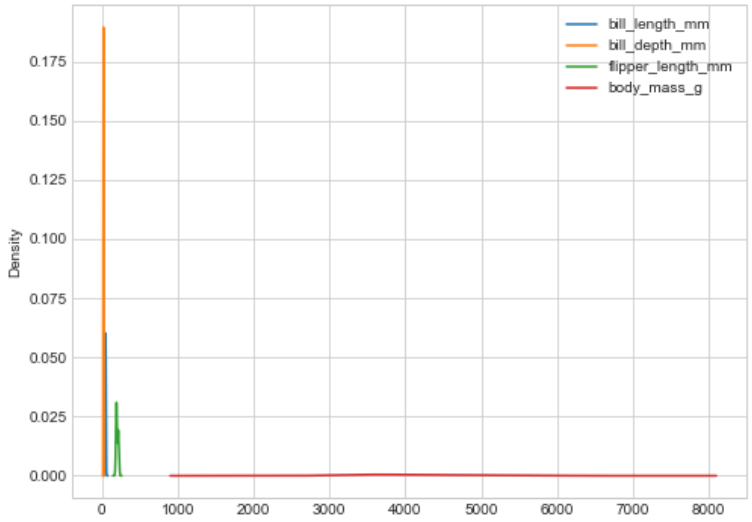
An aside about pandas plotting

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

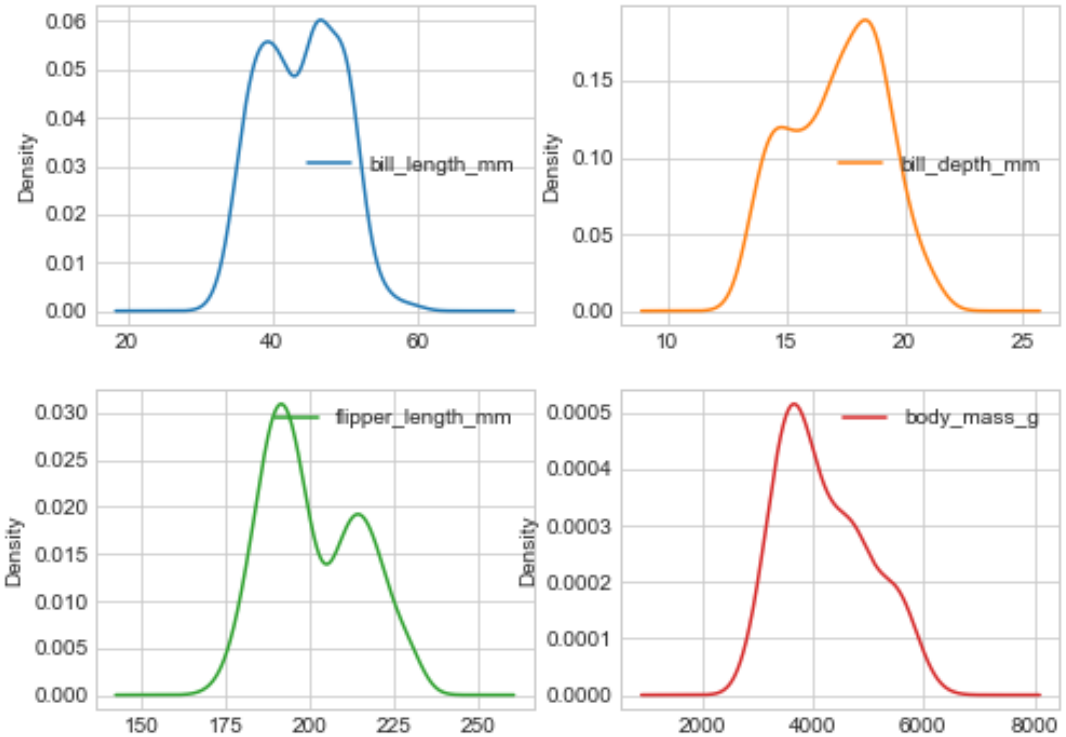
We'd do better by putting each variable into a separate plot

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

Plotting several columns in a DataFrame

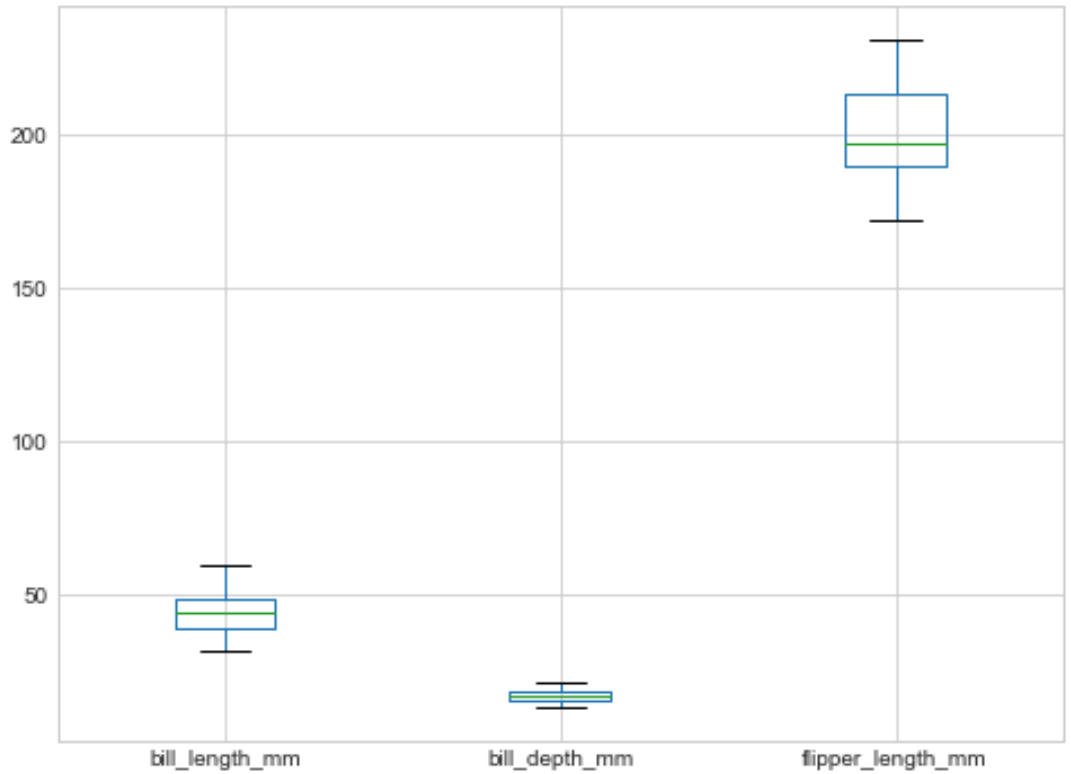
Let's put each subplot on its own scale

```
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");
```



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

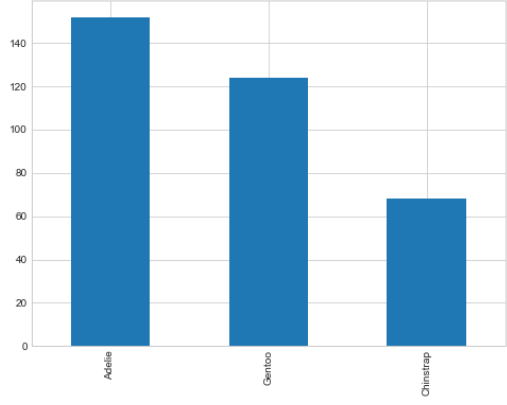
```
freqs = penguins["species"].value_counts()
freqs.index
```

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

```
freqs.values
```

array([152, 124, 68])

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



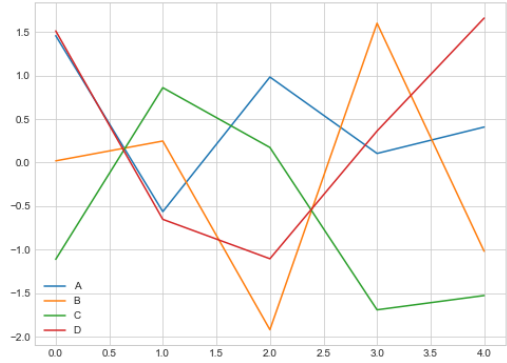
Another example

```
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	1.456751	0.020078	-1.110089	1.511507
1	-0.562469	0.247687	0.859611	-0.651064
2	0.982073	-1.917526	0.173585	-1.103150
3	0.105097	1.599340	-1.689247	0.362538
4	0.408049	-1.017161	-1.525200	1.658364

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



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

seaborn for statistical visualization

seaborn

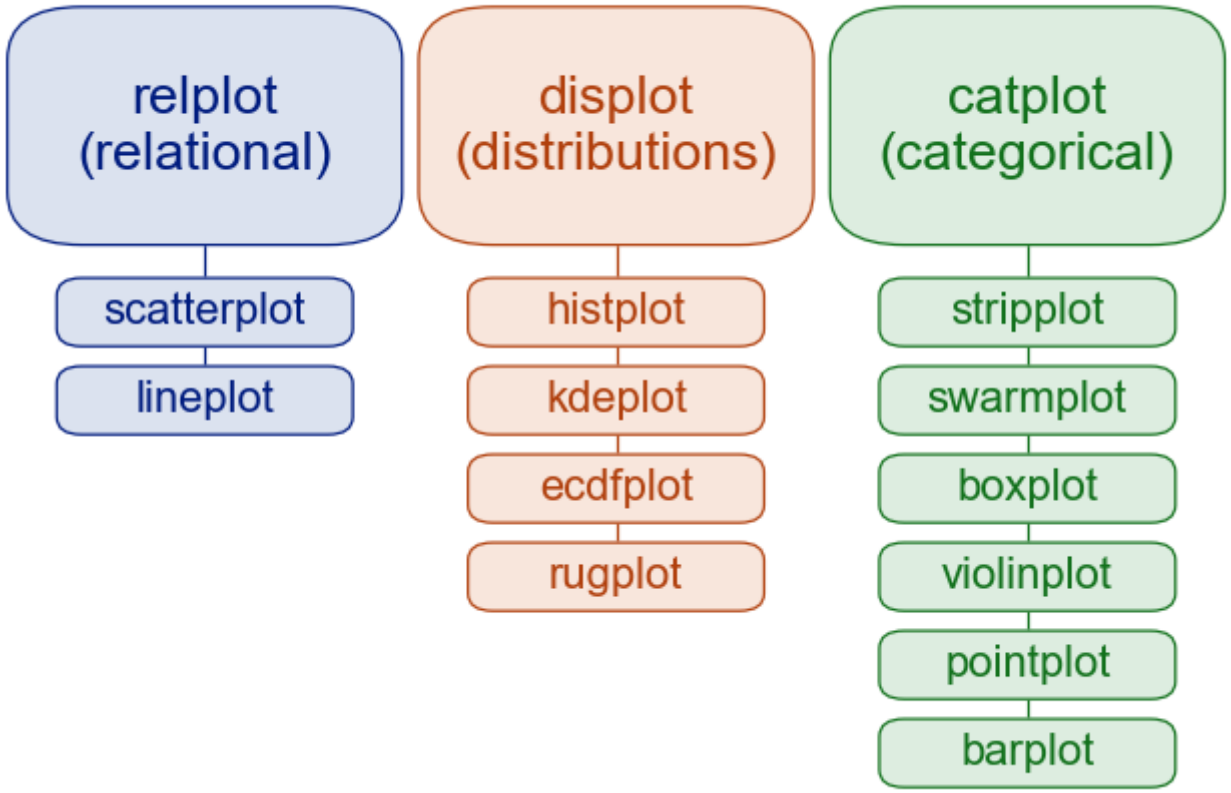
Using pandas, we see basic encodings, basically just x and y

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

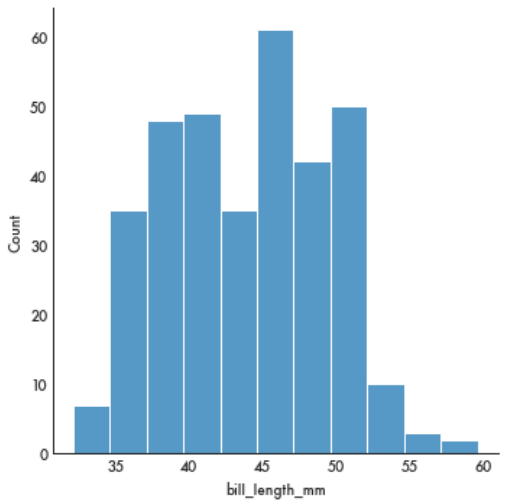
seaborn code follows a general paradigm, where we

- 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

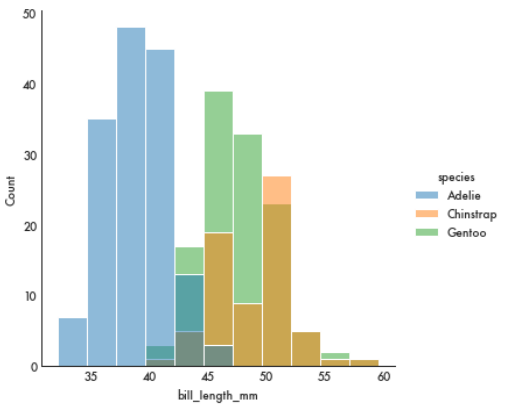
```
sns.set_style("white", {"font.family": "Futura"})
```

Histograms

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

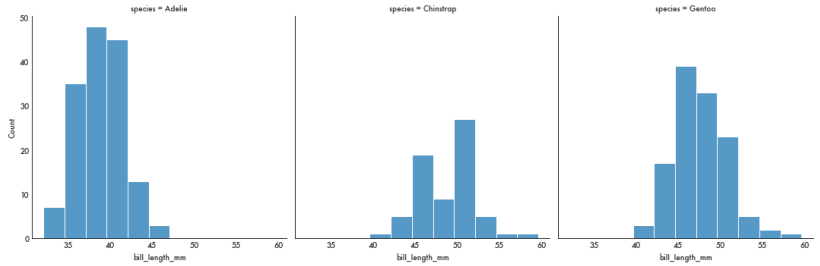



```
sns.displot(
    data=penguins,  # DataFrame
    x="bill_length_mm",  # columns to encode
    kind="hist",  # Type of encoding
    hue="species",  # Encode species as colors
);
```



Histograms

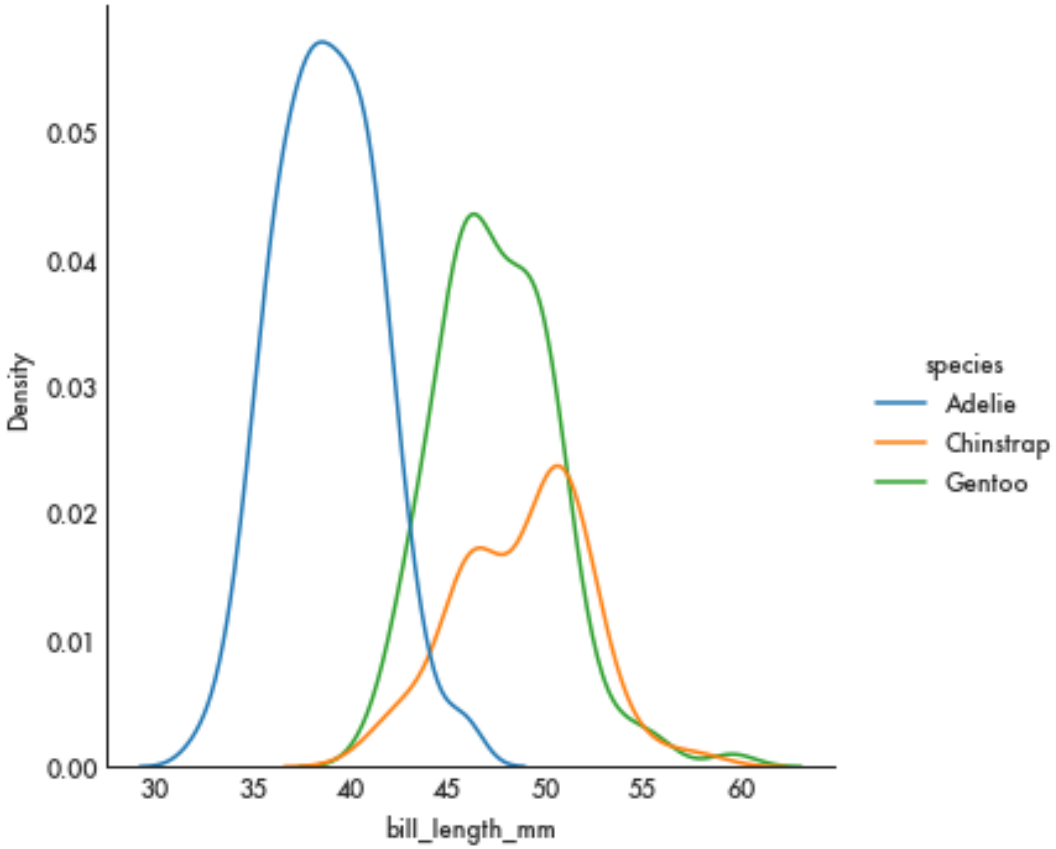
```
sns.displot(
    data=penguins,  # DataFrame
    x="bill_length_mm",  # columns to encode
    kind="hist",  # Type of encoding
    col="species",  # Encode species as facets
);
```




```
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,
);
```

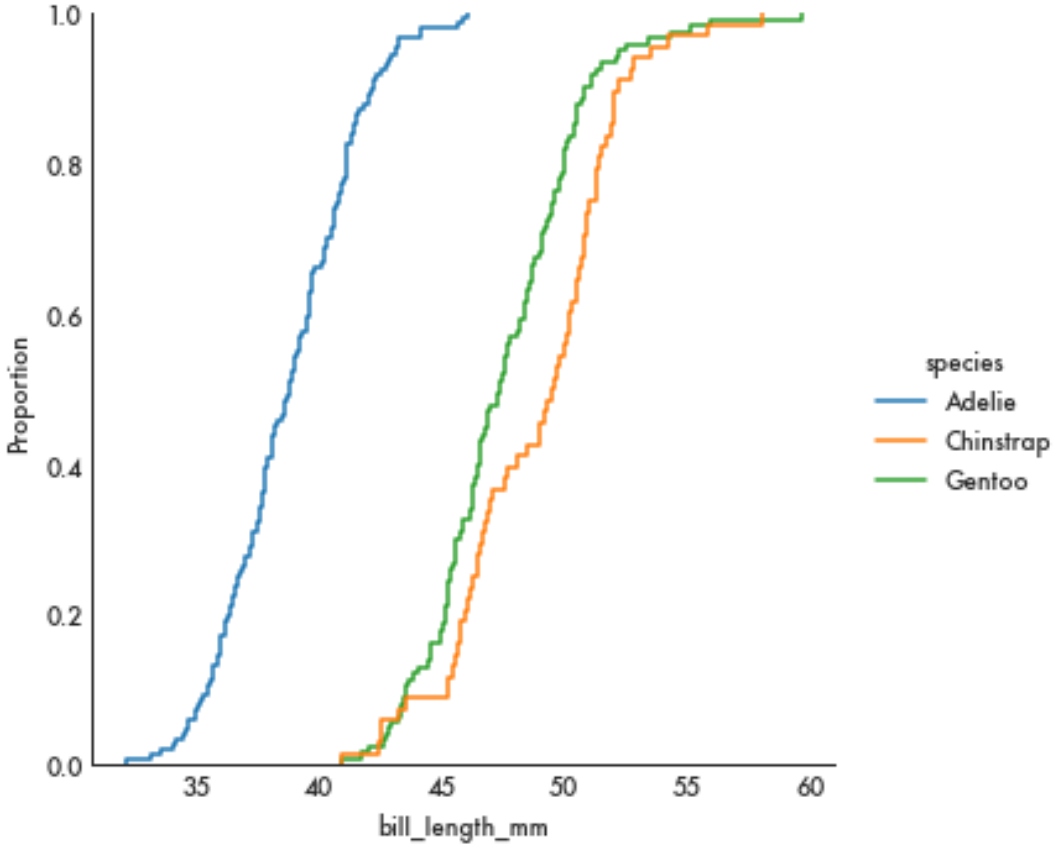
Density plot

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



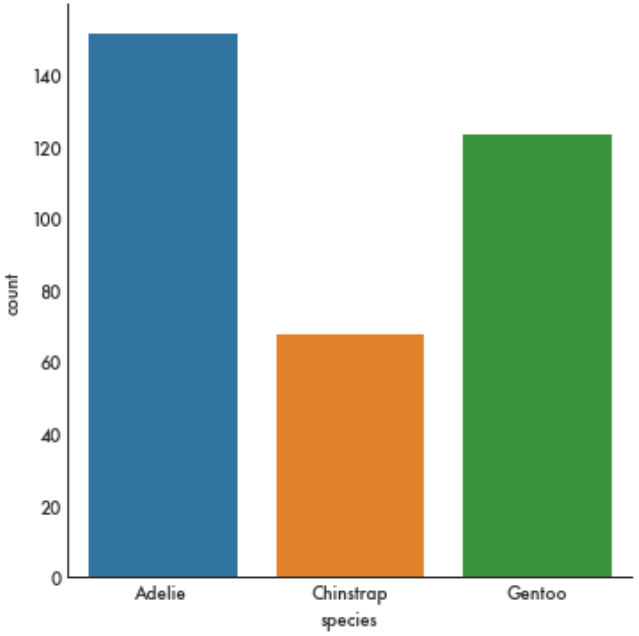
Empirical cumulative distribution plots

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



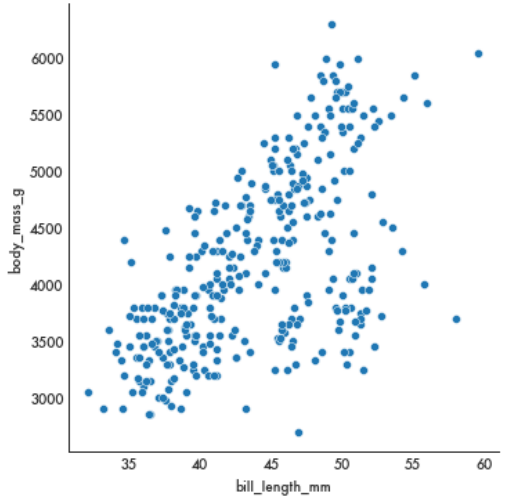
Categorical plots

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

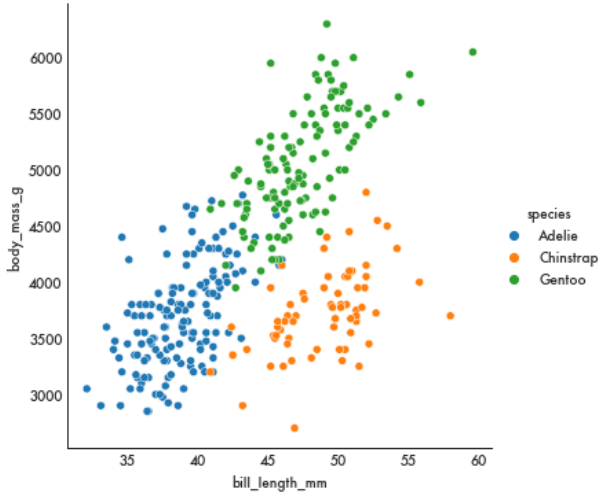


Relating two continuous variables

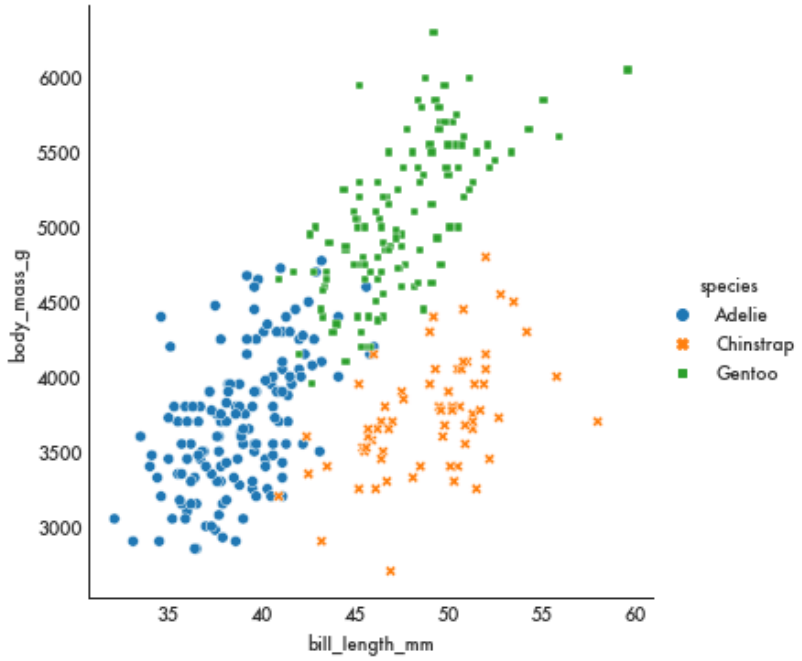
```
sns.relplot(data=penguins, x="bill_length_mm",
```



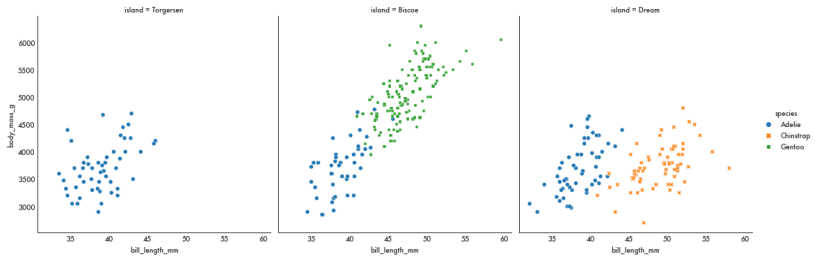

```
sns.relplot(data=penguins, x="bill_length_mm",
```



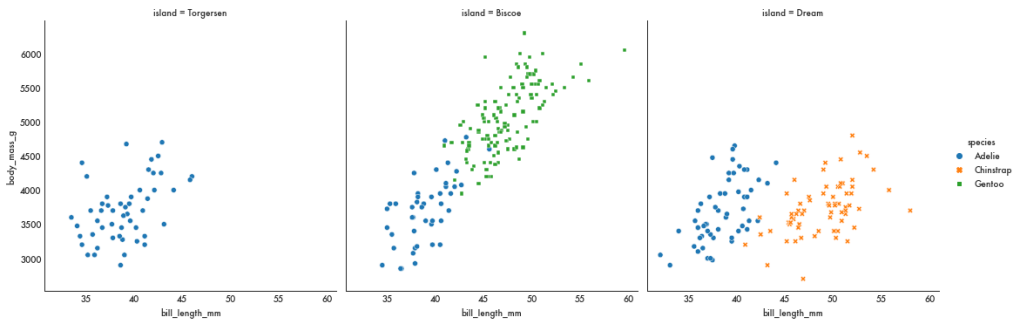
```
sns.relplot(
    data=penguins,
    x="bill_length_mm",
    y="body_mass_g",
    hue="species",
    style="species",
);
```



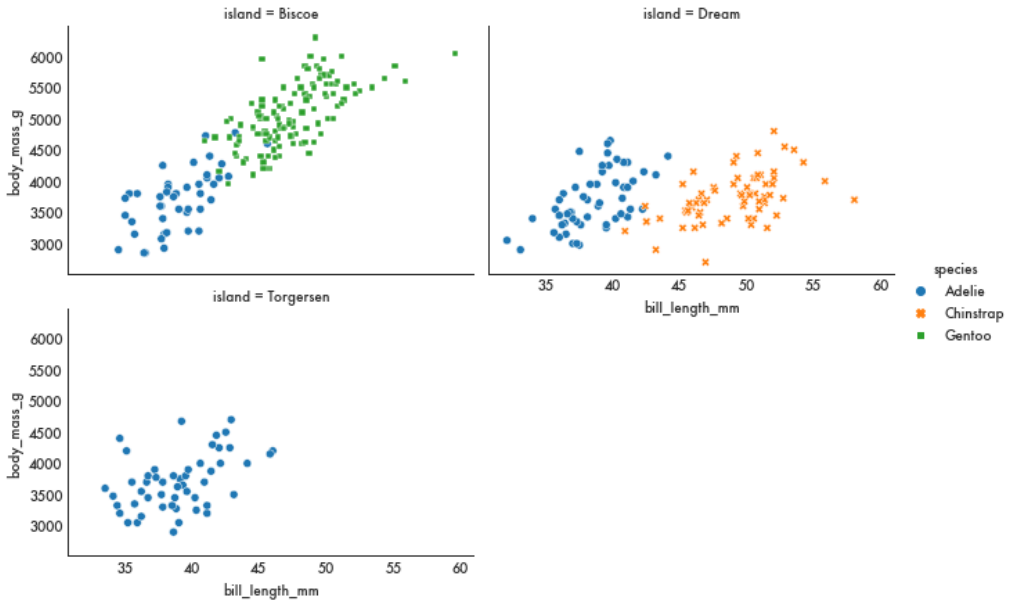

```
sns.relplot(
    data=penguins,
    x="bill_length_mm",
    y="body_mass_g",
    hue="species",
    style="species",
    col="island",
);
```



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




```
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,
);
```

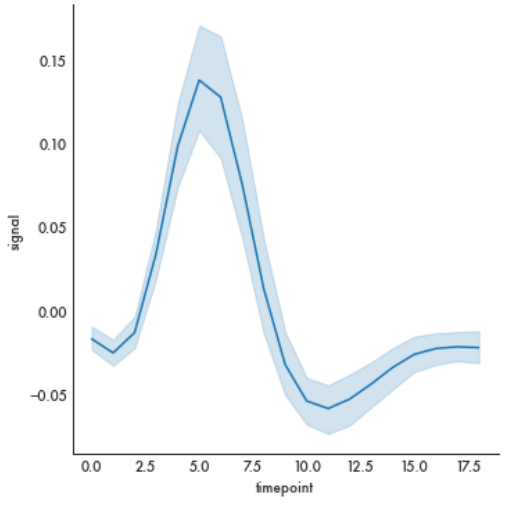


Line plots

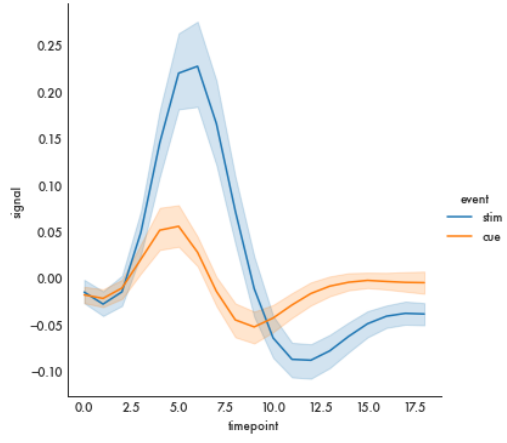
```
fmri = sns.load_dataset("fmri")
fmri.head()
```

	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

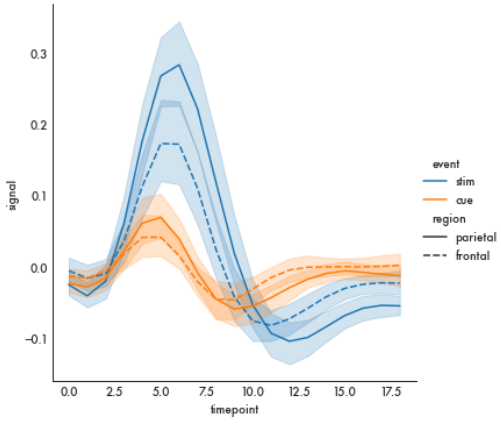
```
sns.relplot(data=fmri, x="timepoint", y="signa
```



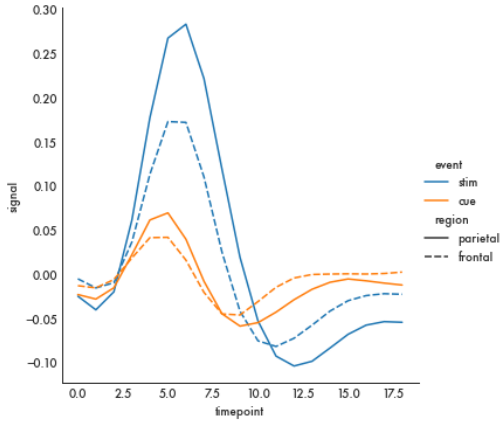

```
sns.relplot(data=fmri, x="timepoint", y="signa
```



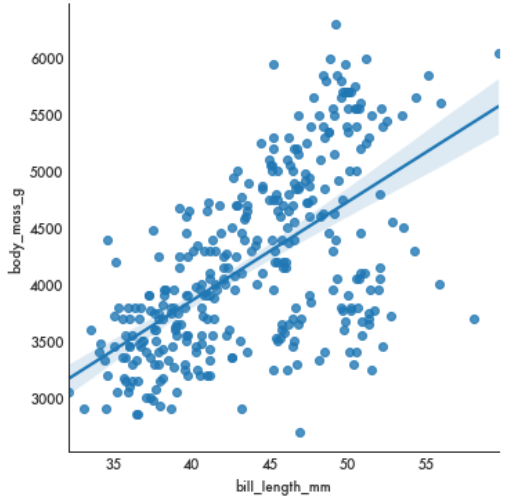
```
sns.relplot(
    data=fmri, x="timepoint", y="signal", kind
);
```



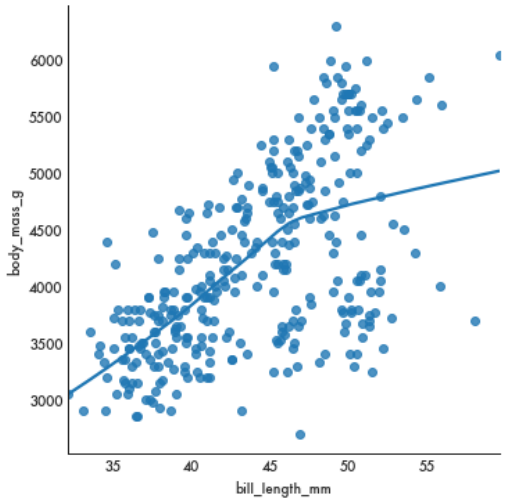

```
sns.relplot(
    data=fmri,
    x="timepoint",
    y="signal",
    kind="line",
    hue="event",
    style="region",
    ci=None,
);
```



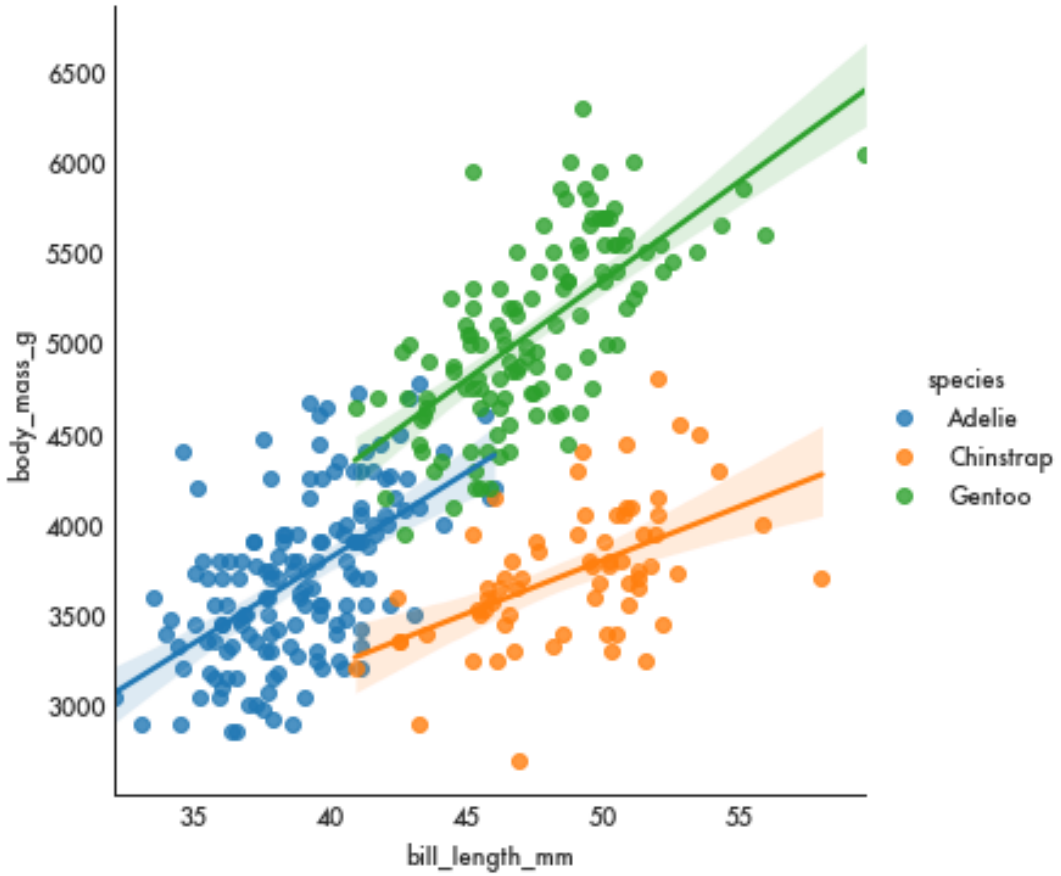
```
sns.lmplot(
    data=penguins,
    x="bill_length_mm",
    y="body_mass_g",
);
```




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



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

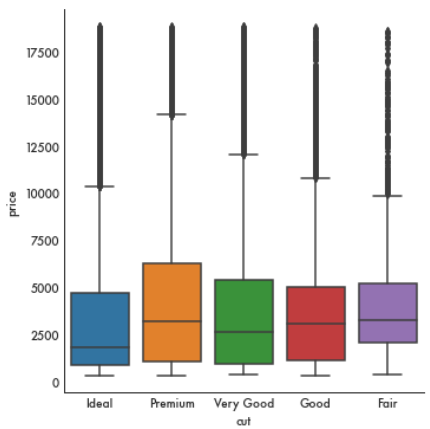


Categorical plots

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

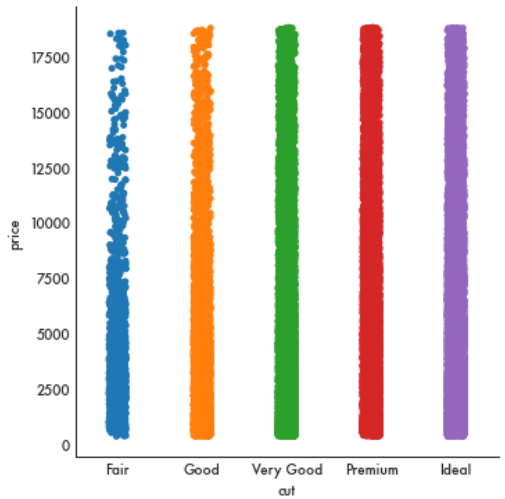
```
(53940, 10)
```

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

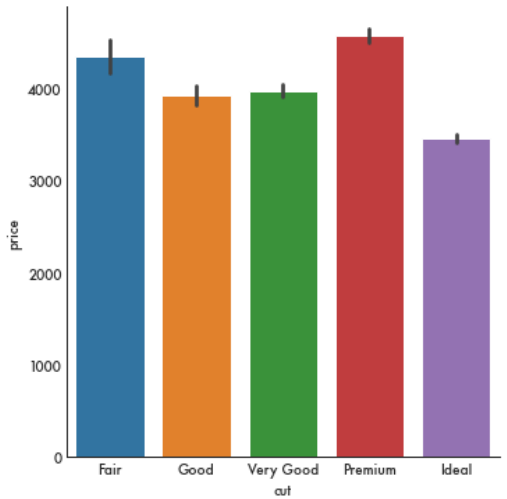


Categorical plots

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

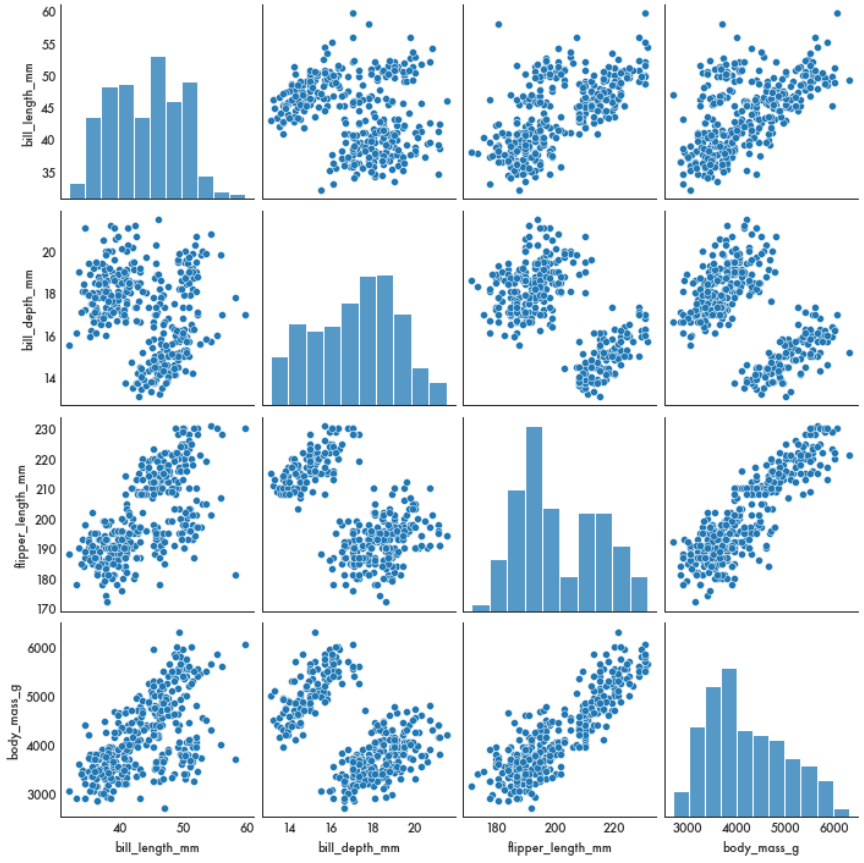



```
cat_order = ["Fair", "Good", "Very Good", "Prer
sns.catplot(
    data=diamonds,
    x="cut",
    y="price",
    kind="bar",
    order=cat_order,
);
```

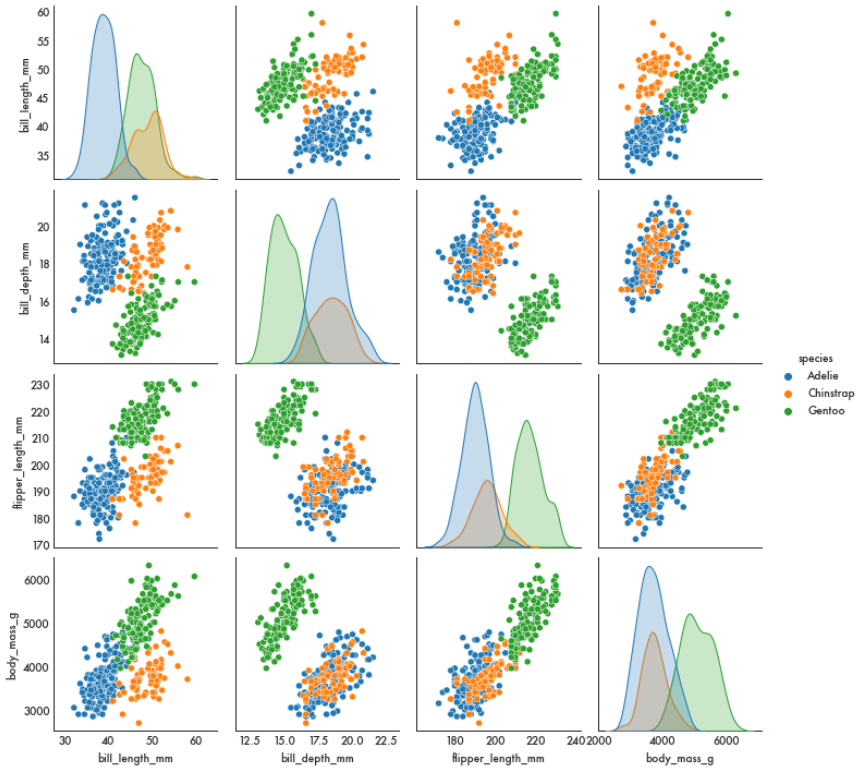


Pair plot

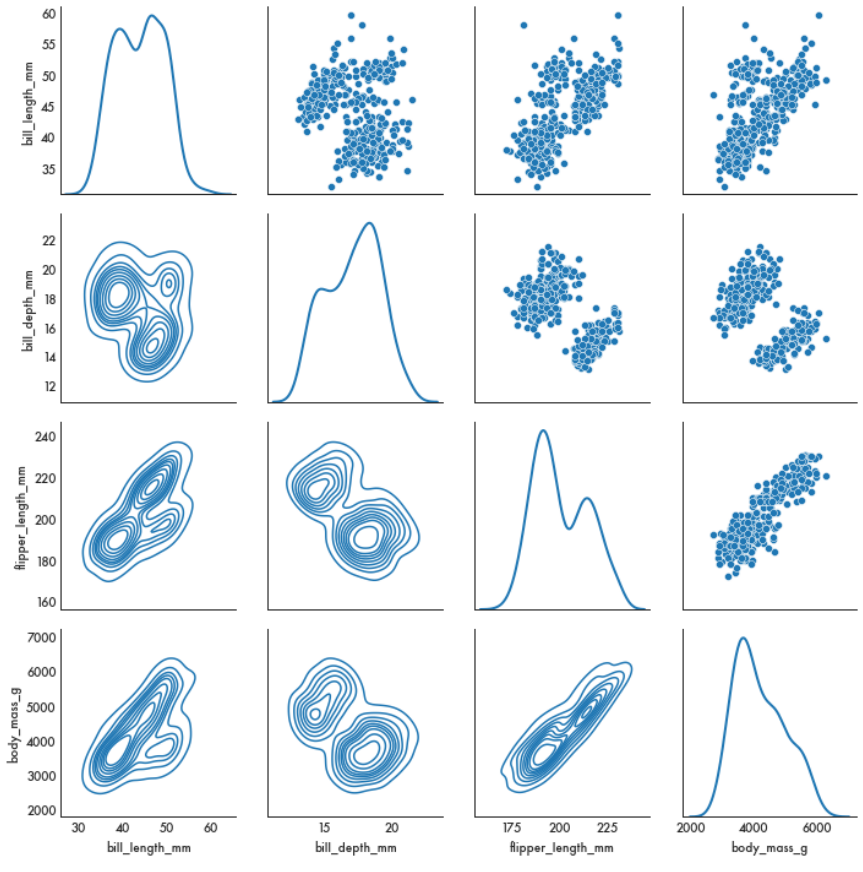
```
sns.pairplot(data=penguins);
```



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

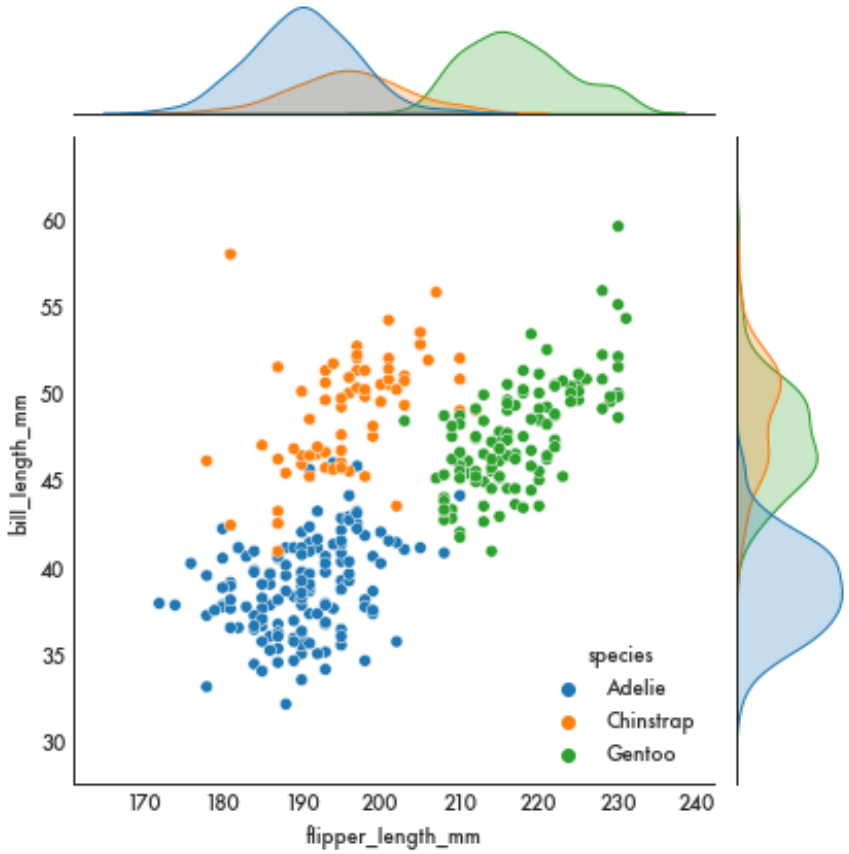


```
g = sns.PairGrid(penguins, diag_sharey=False)
g.map_upper(sns.scatterplot)
g.map_lower(sns.kdeplot)
g.map_diag(sns.kdeplot, lw=2);
```



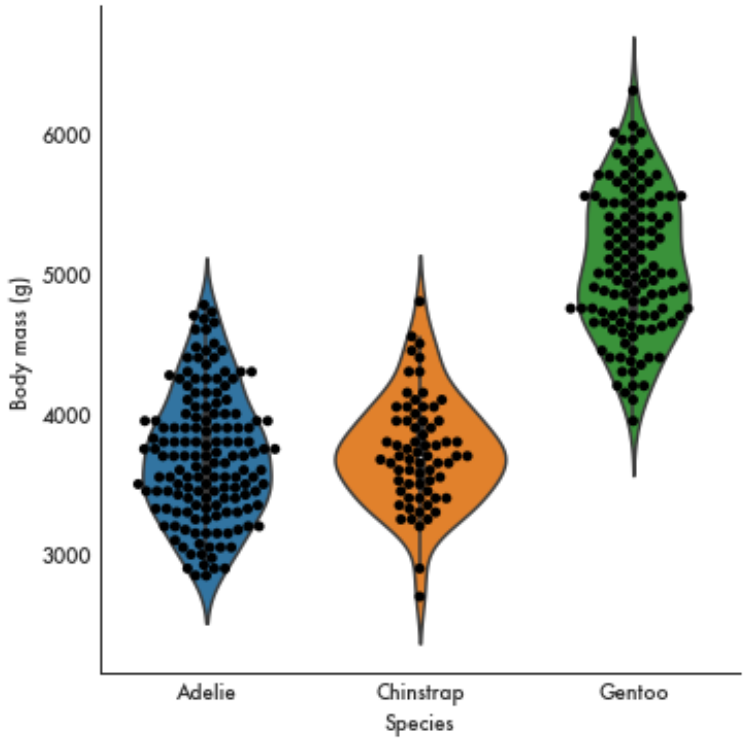
Joint plot

```
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)");
```



Saving your work

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.

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

```
---
class:middle,center,inverse

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

The type of file will be automatically determined by the last three letters of the file name

- penguins.png = PNG file
- penguins.tif = TIFF file
- penguins.pdf = PDF file
- penguins.svg = SVG file

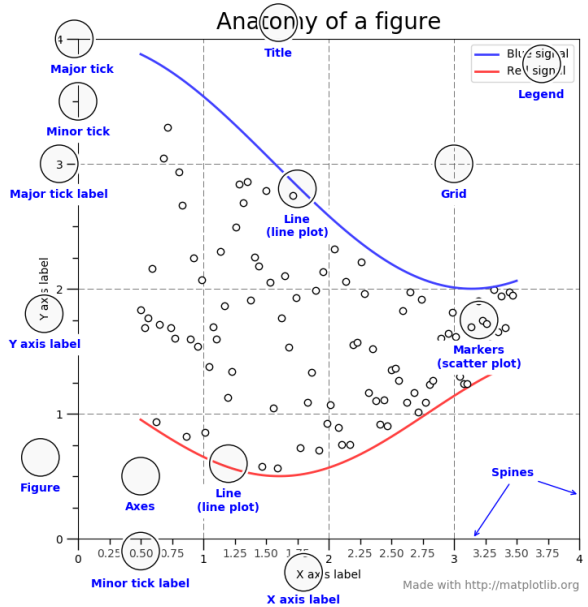
See `help(plt.savefig)` or the online documentation for details.

matplotlib

Granular control

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

We can build a visualization from the ground up



Matplotlib

Both `seaborn` and `pandas` plotting methods actually are creating `matplotlib` plots, so knowing `matplotlib` is useful

- to add features or annotations to a plot
- to customize aspects of the plot
- to easily compose sets of plots
- to export and save plots
- to do visualizations that are not "baked in" to `seaborn` or `pandas`

We've seen some of the elements of `matplotlib` syntax already, but we'll take a deeper dive now.

Matplotlib

- `seaborn` and `pandas` makes things easier to do quickly
  - Nicer, more expressive code for creating common visualizations (better API)
  - based on `pandas.DataFrame`
  - Less things to learn and remember
  - Nicer for data visualization
- `matplotlib` is getting into the trenches, in some ways
  - powerful visualization tool in its own right
  - based on `numpy.array`
  - can create graphs of functions and other kinds of constructs
  - more involved syntax

Matplotlib

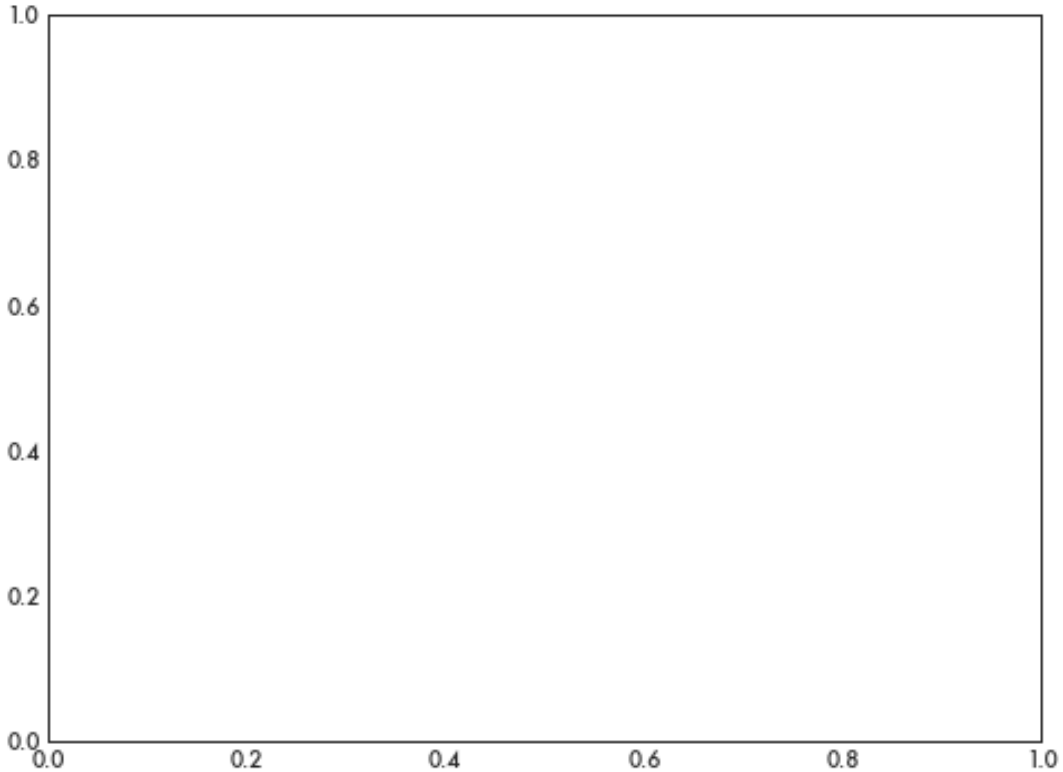
We'll see the *object-oriented API* of matplotlib

> There is another way to create functions in matplotlib, that mimicks Matlab That API is considered outdated in favor of the OO one.

Matplotlib

Let's start with creating a figure "canvas"

```
fig, ax = plt.subplots() # default is 1 row and 1 col, so a single plot
```



Matplotlib

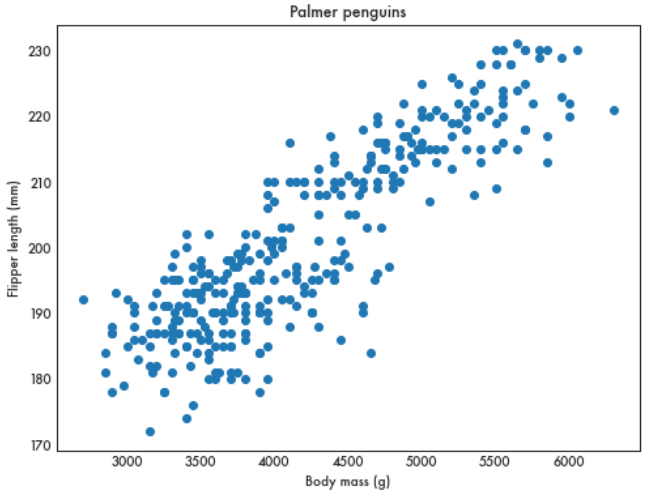
- Figure refers to the topmost layer of a plot, which can contain axes, titles, labels, subplots
- Axes define a subplot
  - We can write our own x-axis limits, y-axis limits, their labels, the type of graph.
  - It controls every detail inside the subplot

Typically we will control aspects of our data visualization at the axis level.

Matplotlib

Now lets add some data to this plot

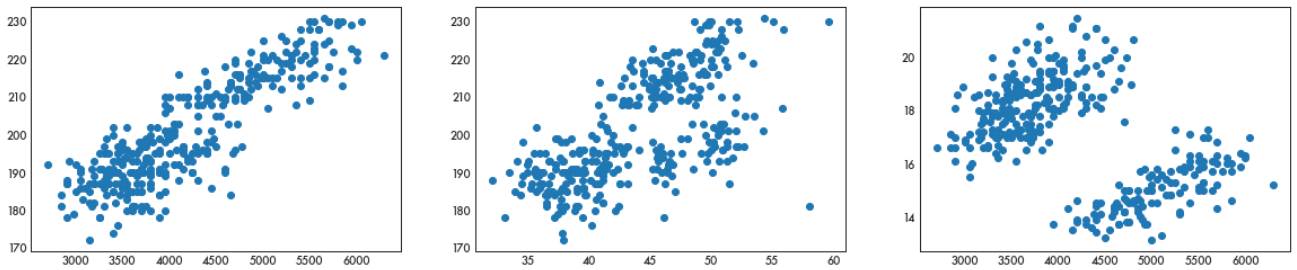
```
fig, ax = plt.subplots()
ax.scatter(penguins['body_mass_g'], penguins['flipper_length_mm'])
ax.set_xlabel('Body mass (g)')
ax.set_ylabel('Flipper length (mm)')
ax.set_title('Palmer penguins');
```



Matplotlib

We can easily do subplots in this paradigm

```
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(20,4))
ax[0].scatter(penguins['body_mass_g'], penguins['flipper_length_mm'])
ax[1].scatter(penguins['bill_length_mm'],penguins['flipper_length_mm'])
ax[2].scatter(penguins['body_mass_g'], penguins['bill_depth_mm']);
```

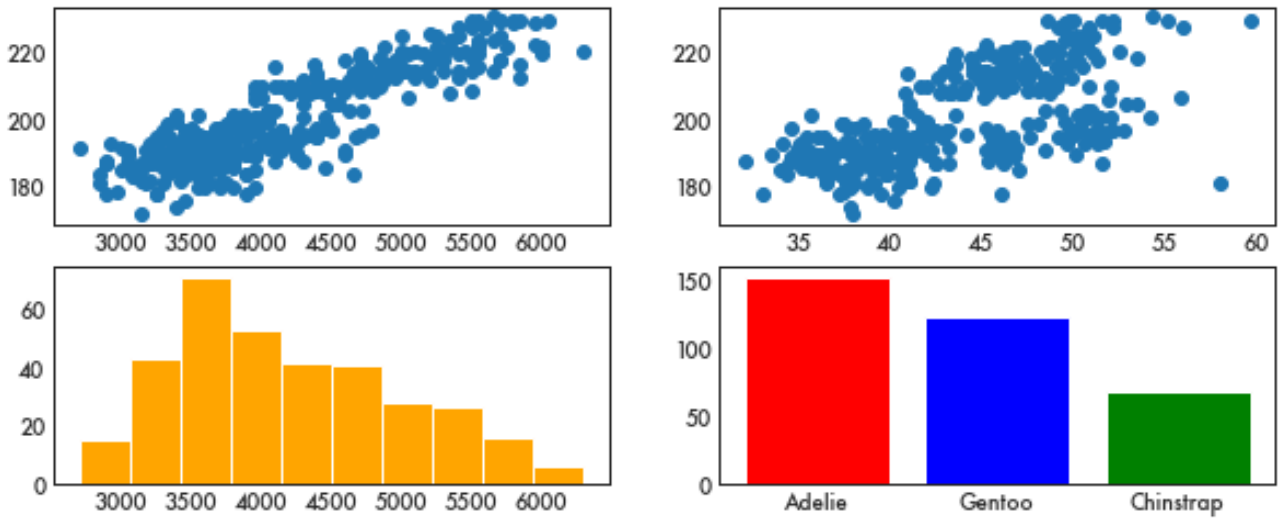


Note that there is no labeling or annotation here. This needs to be done explicitly.

Matplotlib

You can also create a 2-d grid of subplots quite easily

```
cts = penguins['species'].value_counts()
fig,ax = plt.subplots(nrows=2, ncols=2, figsize = (10,4))
ax[0,0].scatter(penguins['body_mass_g'], penguins['flipper_length_mm'])
ax[0,1].scatter(penguins['bill_length_mm'],penguins['flipper_length_mm'])
ax[1,0].hist(penguins['body_mass_g'], color='orange');
ax[1,1].bar(cts.index, cts, color = ['red','blue','green']);
```



Of course you have to label and clean up the figure

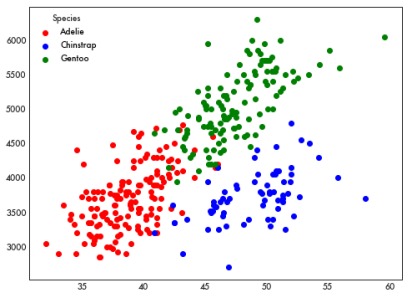
Matplotlib

The part that gets complicated with `matplotlib` are overlays.

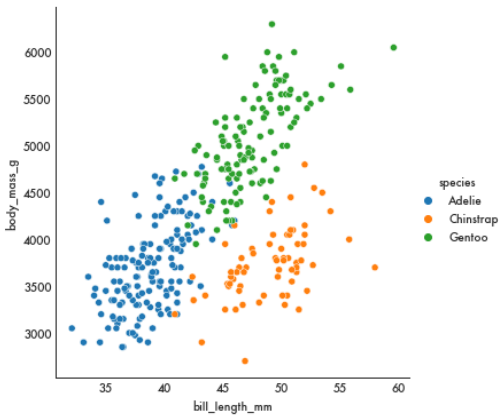
- You will have to layer each overlay on to the canvas using separate commands or loops
- This is where `seaborn`'s API makes things much simpler

Matplotlib

```
fig, ax = plt.subplots()
cols = ["red", "blue", "green"]
for i, u in enumerate(penguins["species"].uniqu
    d = penguins[penguins["species"] == u]
    ax.scatter(d["bill_length_mm"], d["body_mas
ax.legend(title="Species", loc="best", labelco
plt.show();
```

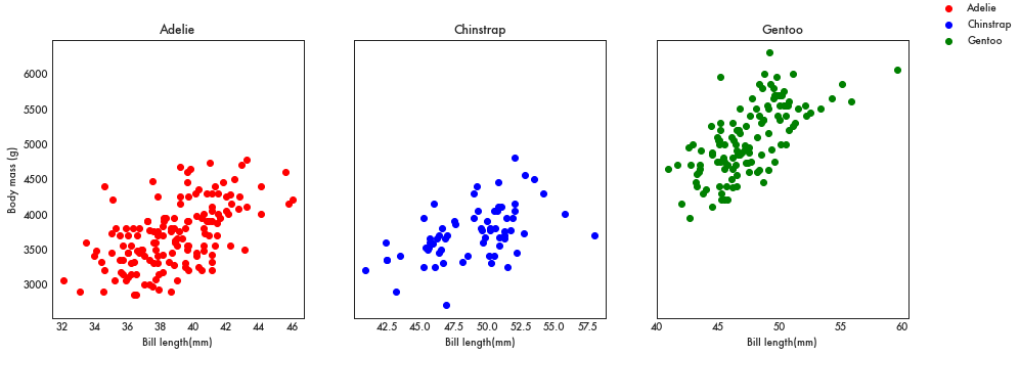



```
sns.relplot(
    data=penguins,
    x='bill_length_mm', y='body_mass_g',
    hue='species');
```

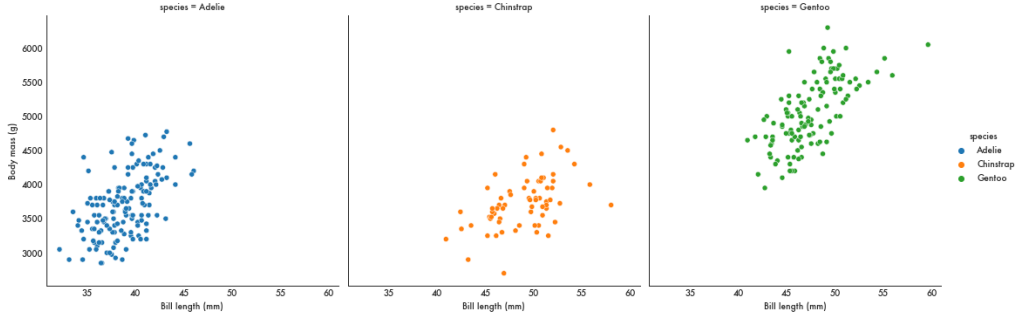


Matplotlib

```
fig, axs = plt.subplots(nrows=1, ncols=3,
                        sharey=True,
                        figsize=[15, 5])
cols = ["red", "blue", "green"]
for i, u in enumerate(penguins["species"].uniqu
    d = penguins[penguins["species"] == u]
    axs[i].scatter(d["bill_length_mm"], d["body
    axs[i].set_title(u)
    axs[i].set_xlabel("Bill length(mm)")
    if i == 0:
        axs[i].set_ylabel("Body mass (g)")
fig.legend();
```



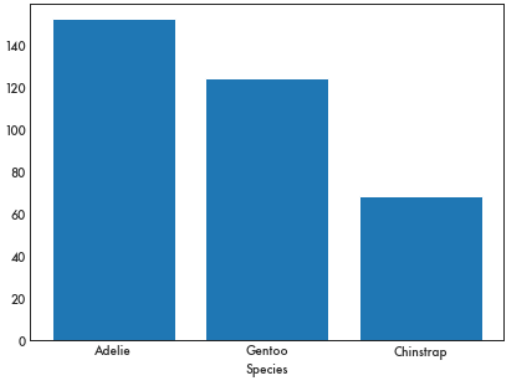

```
g = sns.relplot(
    data=penguins, x="bill_length_mm",
    y="body_mass_g", col="species", hue="speci
)
g.set_xlabels("Bill length (mm)")
g.set_ylabels("Body mass (g)");
```

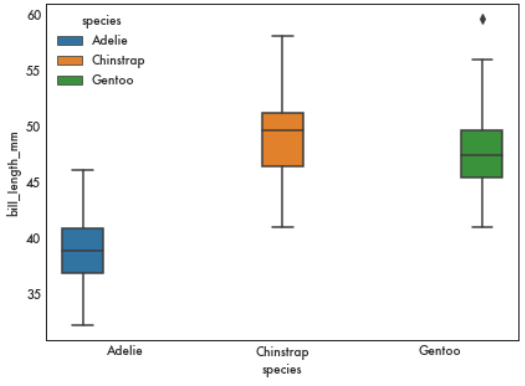


Matplotlib

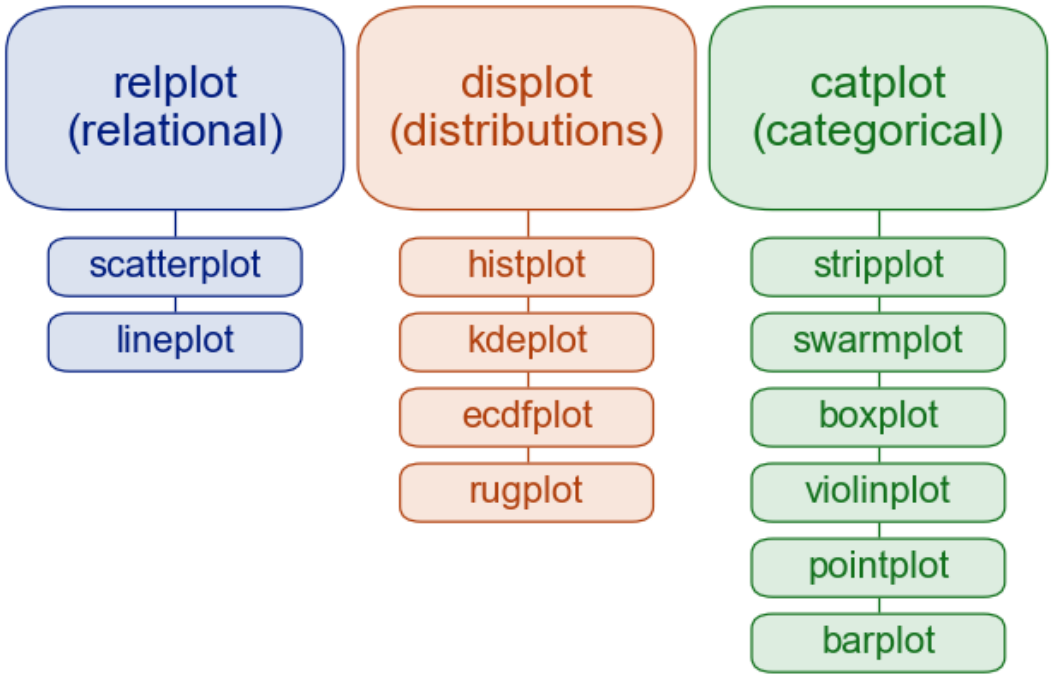
You can mix-and-match different `matplotlib` and `seaborn` axis-level graphics using the axis commands.

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=[15,5])
axs[0].bar(cts.index, cts)
axs[0].set_xlabel('Species')
sns.boxplot(ax = axs[1], data=penguins,
            x = 'species', y = 'bill_length_mm', hue='species',);
```





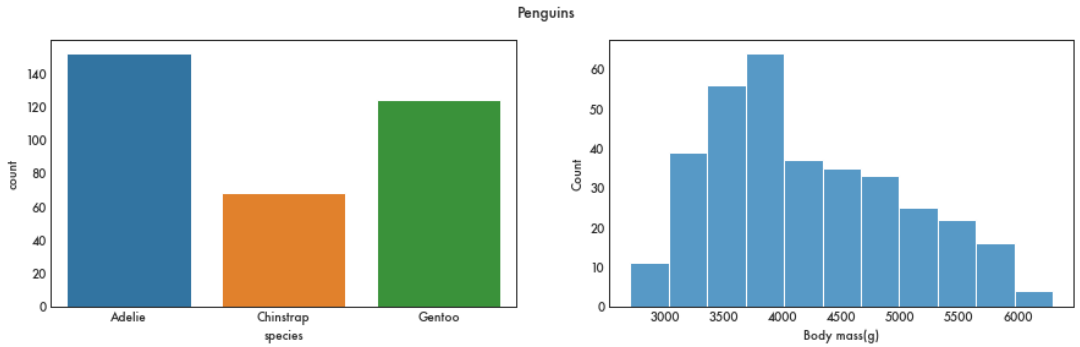
Matplotlib and seaborn compatibility



- The functions at the top (relplot, displot, catplot) are *figure-level* functions and cannot be used with axes
- The functions below them are *axis-level* functions and can be used with axes

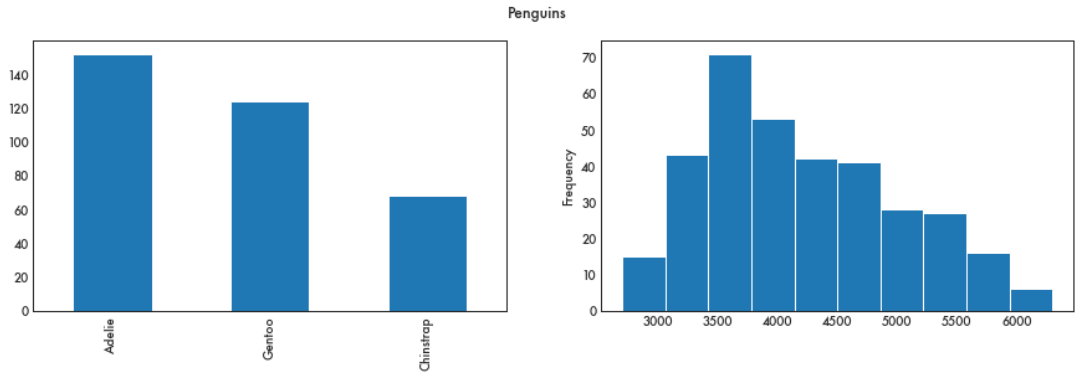
Matplotlib and seaborn compatibility

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=[15,4])
sns.countplot(ax=ax[0], data=penguins, x = 'species')
sns.histplot(ax=ax[1], data=penguins, x = 'body_mass_g');
ax[1].set_xlabel('Body mass(g)');
fig.suptitle('Penguins');
```



Matplotlib and pandas compatibility

```
fig, ax=plt.subplots(nrows=1, ncols=2, figsize=[15,4])
penguins['species'].value_counts().plot(kind='bar', ax=ax[0]);
penguins['body_mass_g'].plot(kind='hist', ax = ax[1]);
fig.suptitle('Penguins');
```



Further reading

Further reading

- An overview of matplotlib plots
- The lifecycle of a matplotlib plot
- The matplotlib usage guide