Chapter 6 Notes

- Advanced plotting with seaborn
    - Seaborn provides additional functionality for more complex visualizations and makes creating visualizations with long-format data easier
    - Categorical data
        - In matplotlib, scatter plots are limited to two numeric variables
        - Seaborn adds `stripplot()` and `swarmplot()` which lets you use one categorical and one numeric variable
            - `stripplot()` can have overlaped points while `swarmplot()` has none
            - `swarmplot()` gives a better idea of the point distribution
        - Enhanced box plots with `boxenplot()` gives a better idea of data distribution, showing additional quantiles at the tails
            - Box plots lose data about distributions
        - Violin plots, with `violinplot()`, add kernel density estimates to the box plot
    - Correlations and heatmaps
        - `heatmap()` can be used to create heatmaps based on correlations between variables
        - `pairplot()` is an alternative to heatmaps, showing scatter plots instead
            - The scatter plots help show correlated variables at a quick glance
            - Histograms are plotted on the diagonals
            - KDEs can be plotted instead of histograms
            - `jointplot()` lets you compare two variables, with histograms on the borders
                - Using the `hex` input to the `kind` argument generates a hexplot, which is a two dimensional histogram
                - The data can be viewed as contours with `kind = 'kde'`
                - Regressions can also be added to the plots with `kind = 'reg'`
    - Regression plots
        - Regression lines can be created with `regplot()` and regressions with residuals can be created with `residplot()`
        - Iterables are used in this example
            - The `itertools` module can be imported to make iterating easier
            - Iterables are objects that can be iterated over
            - When starting a loop with an iterable, an iterator is created
                - The iterator provides the next value until exhausted
                - Iterators can be reused after the first use
            - Lists are an iterable that can be reused
            - This example also relies heavily on the `zip()` function
        - `lmplot()` can be used to plot regressions across different subsets of data
    - Faceting
        - Allows plotting of subsets of data across subplots with `FacetGrid()`
        - `FacetGrid()` sets the data to be used in the grids, `map()` can be used to specify the plot type
- Formatting plots with matplotlib
    - Titles and labels
        - Titles can be added with matplotlib using `plt.title()`
            - The title can be positioned with x and y values and format can be controlled
            - This behaves different for subplots
        - Axes can be labeled with `plt.xlabel()` and `plt.ylabel()`
        - When working with subplots, `plt.title()` generates a title for the last plot to be generated
            - `plt.suptitle()` will generate a title for the overall plot, above all of the subplots
            - Individual axis labels can be generated using the `set_ylabel()` function on the specified Axes object
    - Legends
        - Legend aspects can be controlled through the `plt.legend()` function and `Axes.legend()` method

      - Can control legend location and look, which includes fonts and colors
    - Commonly used parameters include `loc`, `bbox_to_anchor`, `ncol`, `framealpha`, and `title`
    - By default, `matplotlib` will find the best location for the legend
      - The `loc` argument can be used to customize the legend location
      - Location strings can be passed to `loc`
    - `ncol` can help control the shape of the legend with many entries by adding columns
  - Formatting axes
    - Formatting axes in `matplotlib` can be done with either the `plt.xlim()` / `plt.ylim()` functions or the `set_xlim()` / `set_ylim()` methods on Axes objects
      - Min and max values are passed to both of these
    - The scale of the axes can be changed with `plt.xscale()` / `plt.yscale()`
      - Ex. `plt.yscale('log')` changes the y scale from linear to log
    - Tick mark visibility and location can be set by `plt.xticks()` / `plt.yticks()`
    - Axis label format can be modified using the `PercentFormatter` class in the `matplotlib.ticker` module
      - There are many other formatting options in this module, like `EngFormatter` for engineering notation
      - The `MultipleLocator` class allows for placement of ticks at a multiple of a number of locations
- Customizing visualizations
  - Adding reference lines
    - Sometimes specific values or areas need to be highlighted
    - Horizontal reference lines can be added with `plt.axhline()`
      - These are useful for y axis related boundaries
    - Vertical reference lines can be drawn with the `plt.axvline()` method
  - Shading regions
    - Line the reference lines, vertical and horizontal shaded regions can be drawn by calling the `axvspan()` and `axhspan()` methods on `Axes` objects
    - The area between curves can be shaded using the `plt.fill_between()` or `plt.fill_betweenx()` functions
      - The `plt.fill_between()` function fills between two y values, `plt.fill_betweenx()` fills between two x values
      - Boolean masks can be provided with the `where` argument so that only certain sections are filled
  - Annotations
    - Add labels to values can be done with the `plt.annotate()` function
      - Annotations can be placed with `xy` and `xytext` arguments
      - `xy` can be used to place arrows and pointers, `xytext` can be used to place text
      - Arrows can be added using the `arrowprops` argument
  - Colors
    - Plot colors should be consistent, but color palettes can be easily adopted
    - For example, the `color` argument can be used in the `df.plot()` method
      - Colors can be specified by single character names, `matplotlib` colors, hex code strings, and rbg and alpha codes (in the range of [0,1])
    - Colormaps can be provided, these can be cycled through automatically
      - There are three different types of colormaps: qualitative, sequential, and diverging
        - Qualitative colormaps have no ordering or relationship between colors
        - Sequential colormaps are used for information with ordering
        - Diverging colormaps has a middle value between two extremes
      - Colormap names can be found with the `cm.datad.keys()` function from the `cm` module in the `matplotlib` package
        - Colormap names in reverse order are colormaps in reverse order
      - Colormap colors can be chosen by passing a value in the range [0,1] to the `colormap` object callable
        - This can be passed onto the `color` argument
      - Custom colormaps can be generated using the `color_util.py` package
      - The example provided on pg. 378 to 380
      - Conditional coloring can also be applied using colormaps

        - The example on pg. 383 provides a method to create a generator to determine plot color based on the data
            - The advantage of the generator is that it calculates the conditions and color only when it's asked to
- Textures
    - Textures can be used to enhance plots beyond color, especially where color isn't useful, i.e. color blindness or where differences in colors are difficult to discern
    - Textures can be applied in the `df.plot()` and `plt.fill_between()` methods using the `hatch` argument