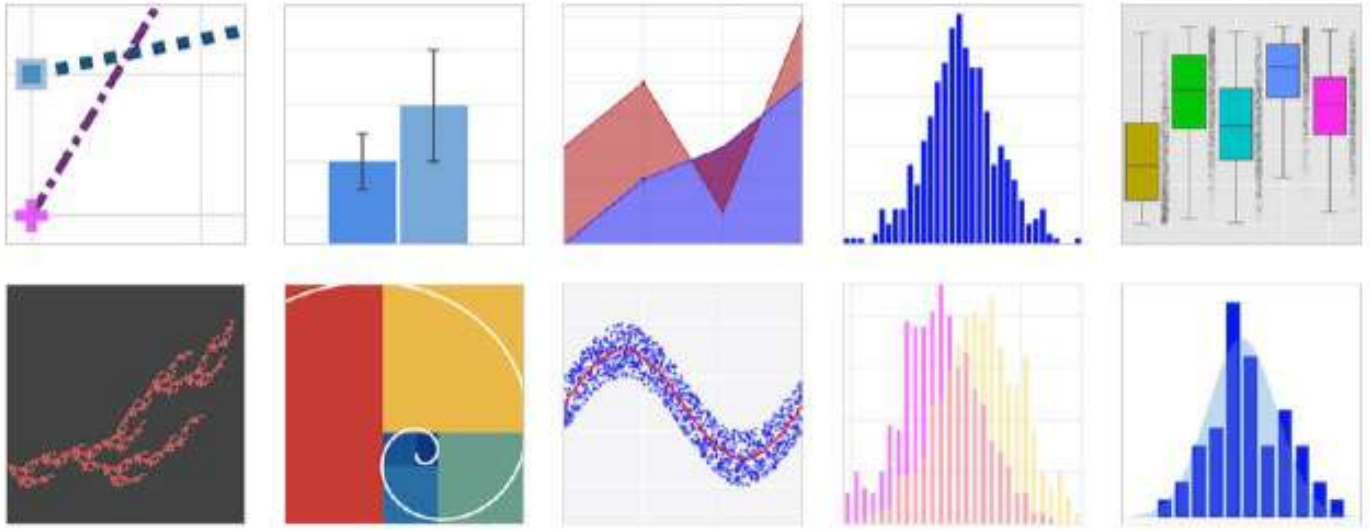
Data Visualizations

Peerapon Vateekul, Ph.D.

peerapon.v@4amconsult.com

Outlines

- Overview
- Exploratory Data Analysis (EDA)
- Scatter Plots
- Line Plots
- Histograms Plots
- Bar Plots
- Box Plots
- Matplotlib Structure

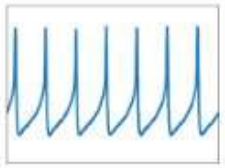


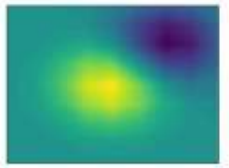
Reference : https://towardsdatascience.com/5-quick-and-easy-data-visualizations-in-python-with-code-a2284bae952f

Overview

- **Matplotlib:** Python 2D plotting library
- **Seaborn:** Statistical data visualization (**On top; Matplotlib**)
- **Folium**: interactive visualization leaflet map

Overview - Matplotlib



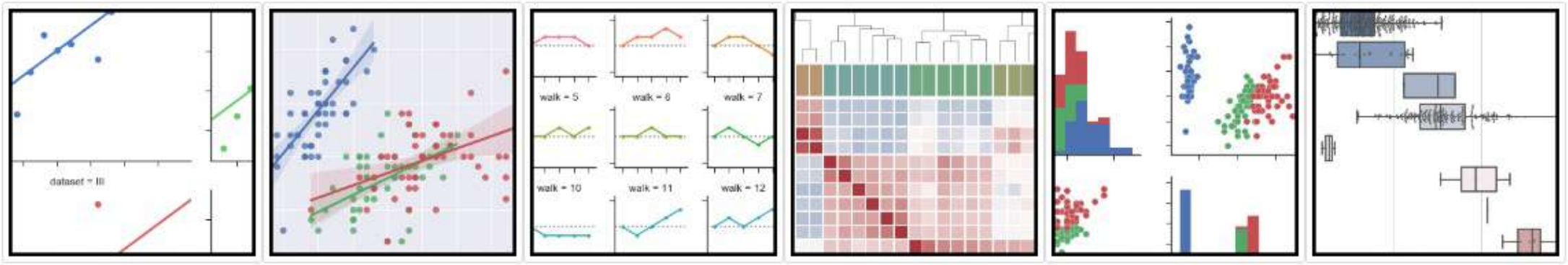



- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.
- Matplotlib tries to make easy things easy and hard things possible.
- You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.
  - For examples, see the sample plots and thumbnail gallery.
- For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython.
- For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Overview - Seaborn

Seaborn

- Seaborn is a Python visualization library based on matplotlib.
- It provides a high-level interface for drawing attractive statistical graphics.
- On top; Matplotib library.



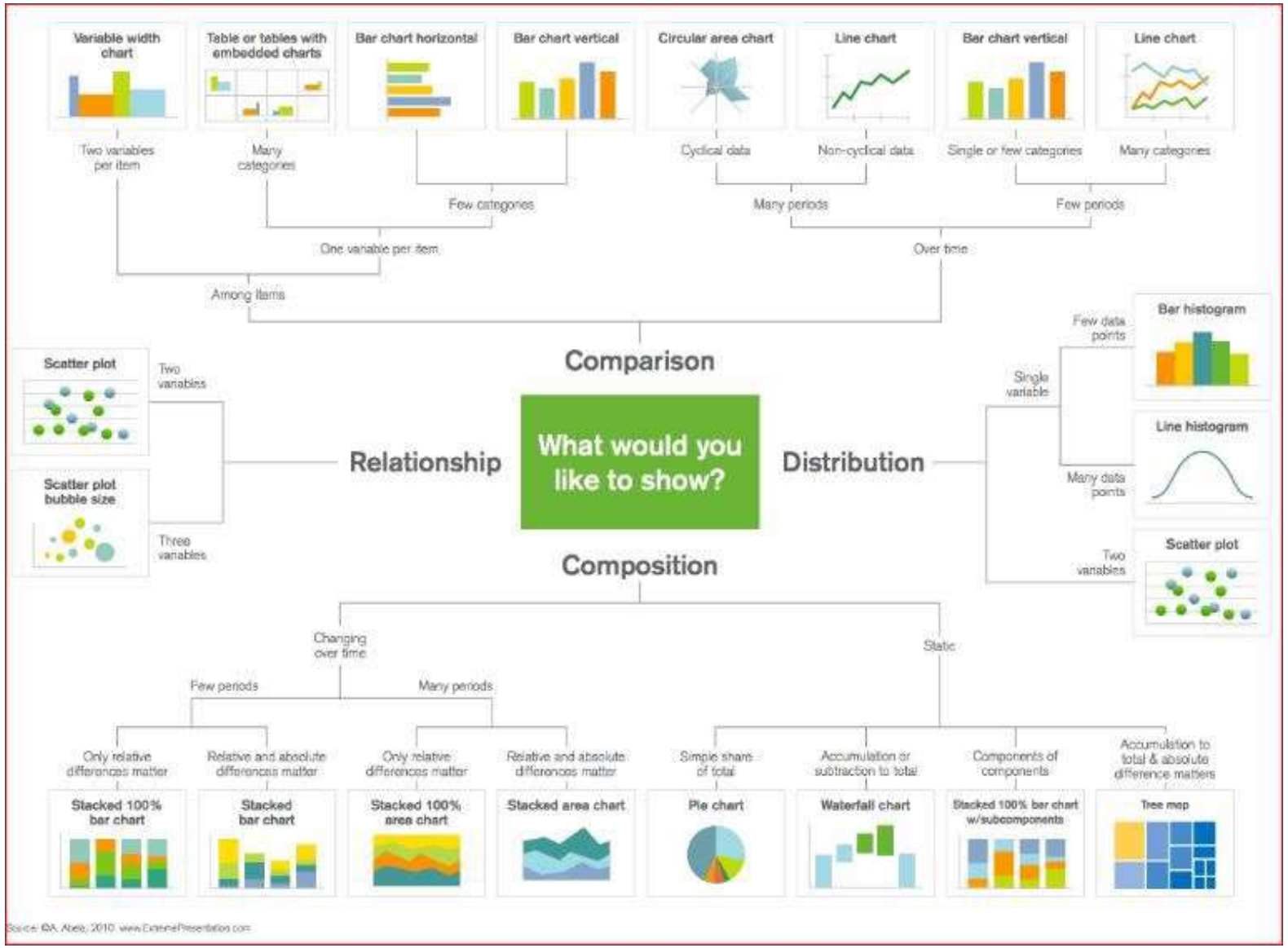
Overview - Plotly

- Plotly creates leading open-source tools for composing, editing, and sharing **interactive** data visualization via the Web.
- Our collaboration servers (available in cloud or on premises) allow data scientists to showcase their work, make graphs without coding, and collaborate with business analysts, designers, executives, and clients.

Exploratory Data Analysis (EDA)

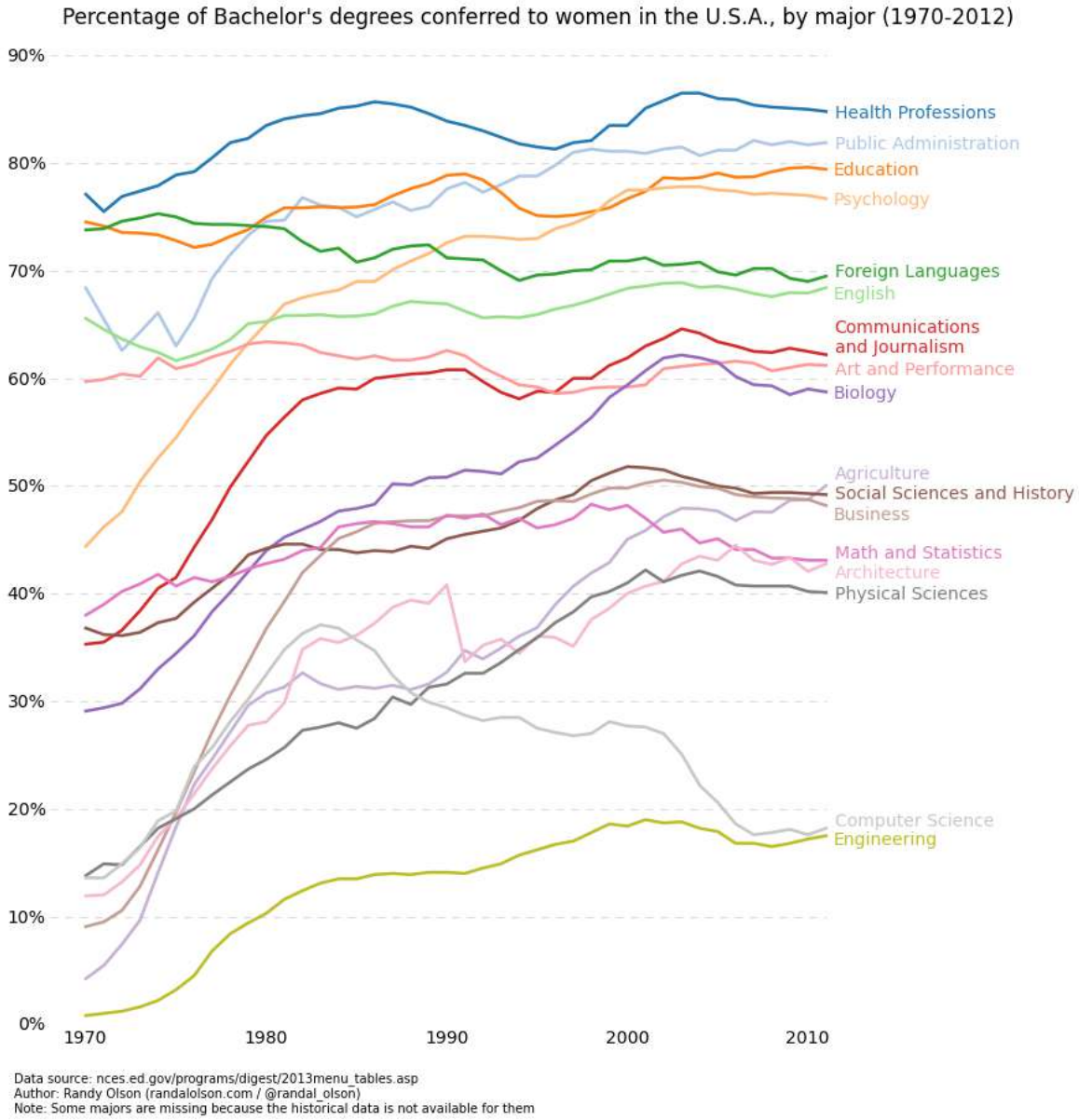
- Data Visualization is a big part of a data scientist's jobs.
- In the early stages of a project, you'll often be doing an **Exploratory Data Analysis (EDA)** to gain some insights into your data.
- Creating visualizations really helps make things clearer and easier to understand, especially with larger, high dimensional datasets.
- Towards the end of your project, it's important to be able to present your final results in a clear, concise, and compelling manner that your audience, whom are often non-technical clients, can understand.

Exploratory Data Analysis (EDA)



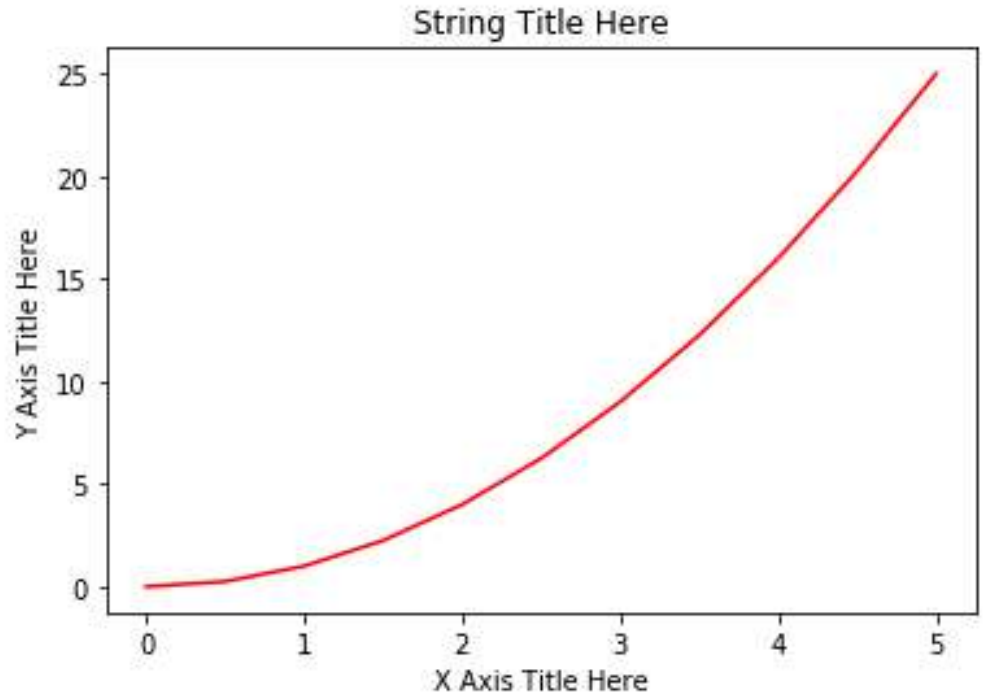
Line Plots

- Line plots are best used when you can clearly see that one variable varies greatly with another i.e., they have a high covariance. Let's look at the figure below to illustrate.
- We can clearly see that there is a large amount of variation in the percentages over time for all majors.
- Line plots are perfect for this situation because they basically give us a quick summary of the covariance of the two variables (percentage and time). Again, we can also use grouping by color encoding.



Line Plots

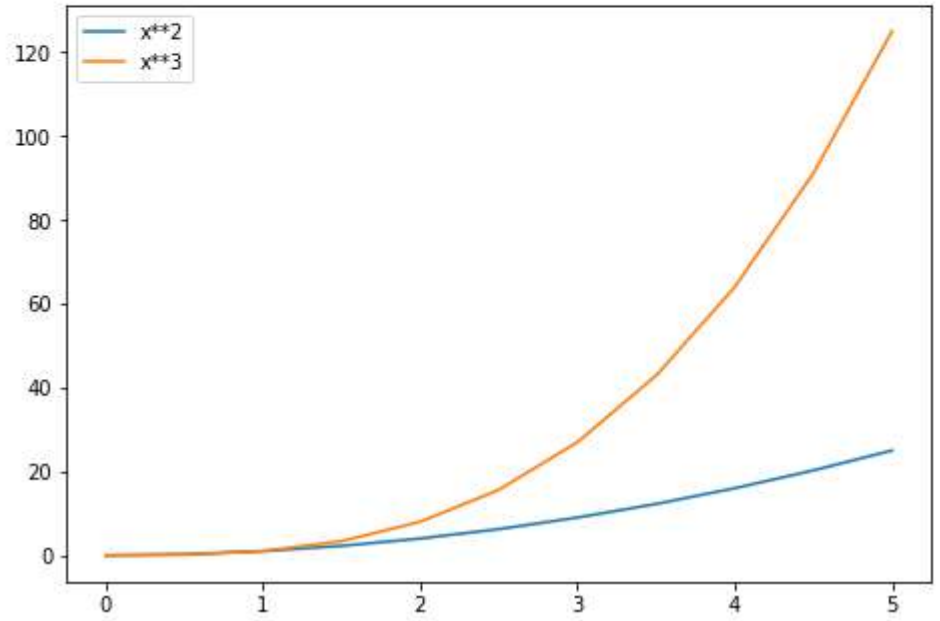
```
plt.plot(x, y, 'r') # 'r' is the color red
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



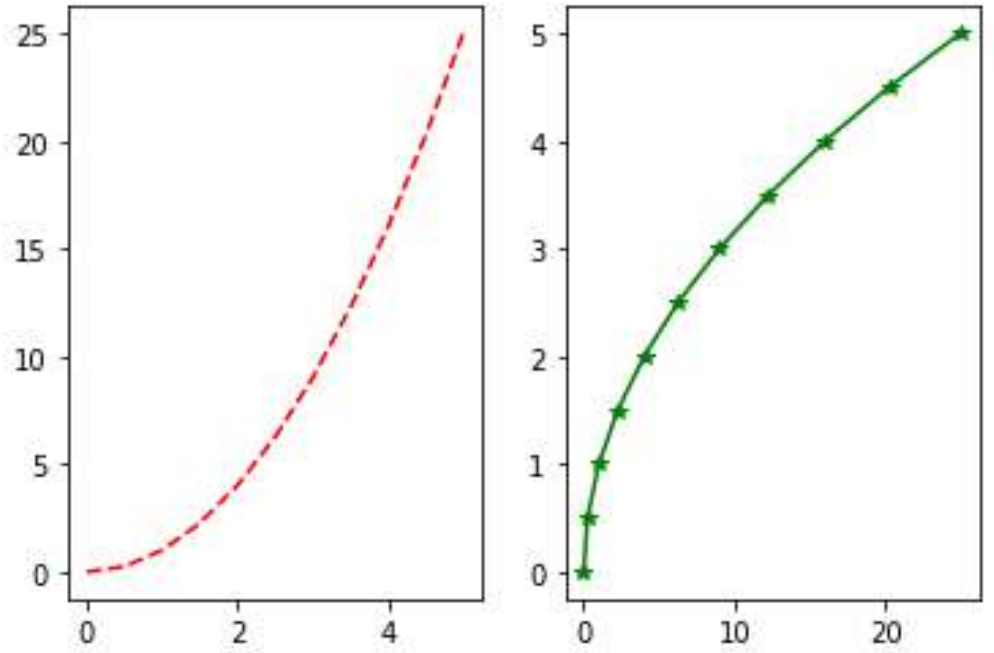
Line Plots

```
fig = plt.figure()

ax = fig.add_axes([0,0,1,1])

ax.plot(x, x**2, label="x**2")
ax.plot(x, x**3, label="x**3")
ax.legend()
```

```
<matplotlib.legend.Legend at 0x1c6b9a4ff88>
```



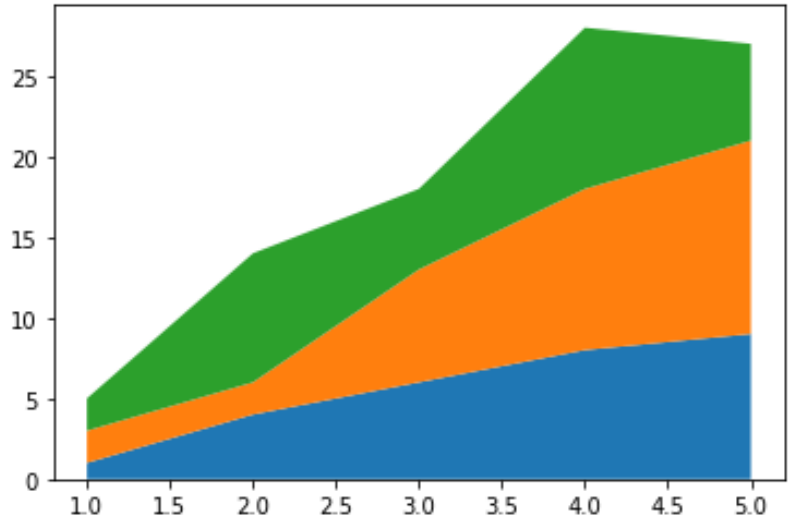

```
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1,2,1)
plt.plot(x, y, 'r--') # More on color options later
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```



Stack Area

```
x=range(1,6)
y1=[1,4,6,8,9]
y2=[2,2,7,10,12]
y3=[2,8,5,10,6]

plt.stackplot(x,y1, y2, y3, labels=['A','B','C'])
```

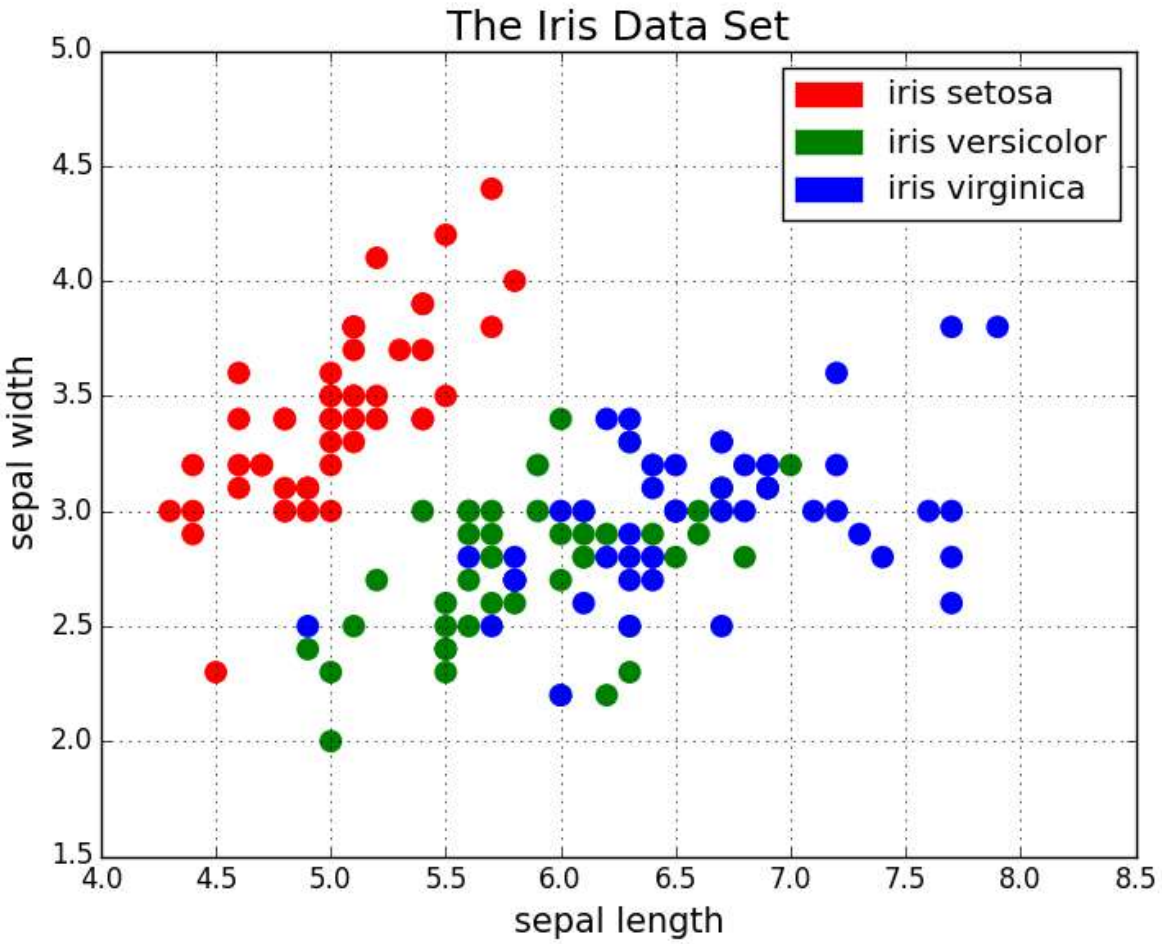
```
[<matplotlib.collections.PolyCollection at 0x1f8baecec88>,
 <matplotlib.collections.PolyCollection at 0x1f8baf79388>,
 <matplotlib.collections.PolyCollection at 0x1f8baf79888>]
```



Reference : https://python-graph-gallery.com/250-basic-stacked-area-chart/

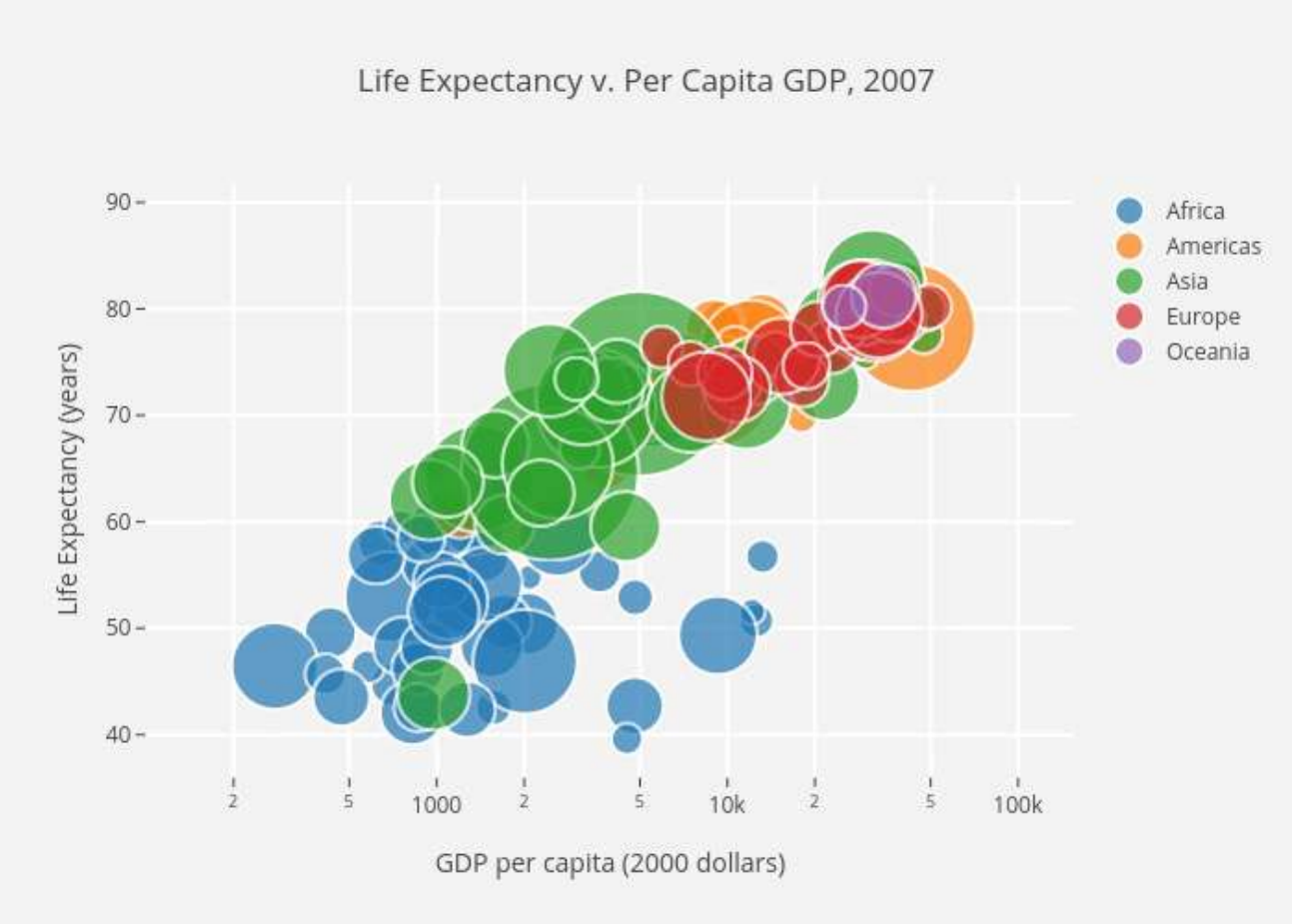
Scatter Plots

- Scatter plots are great for showing the relationship between two variables since you can directly see the raw distribution of the data.
- You can also view this relationship for different groups of data simple by colour coding the groups as seen in the first figure below.
- Want to visualize the relationship between three variables? No problemo! Just use another parameters, like point size, to encode that third variable as we can see in the second figure below. b



Scatter Plots

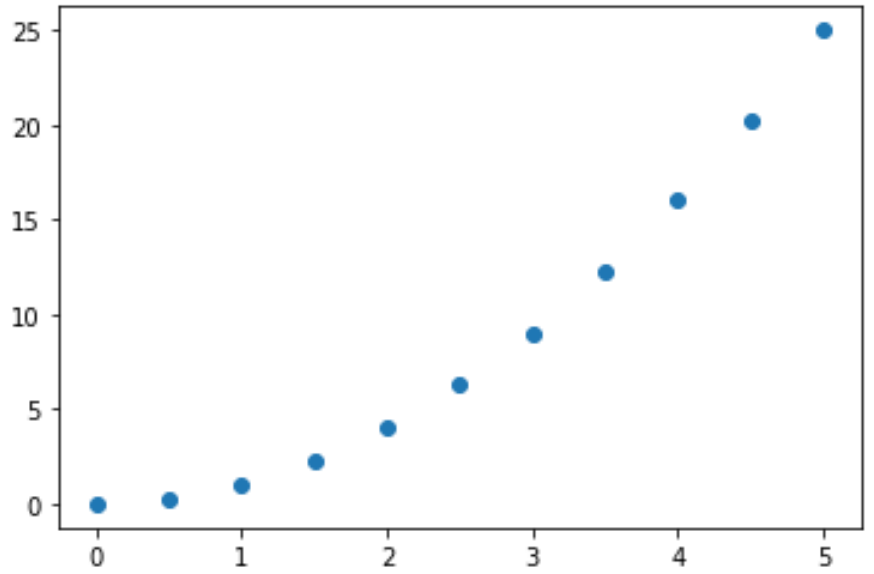
- Scatter plots are great for showing the relationship between two variables since you can directly see the raw distribution of the data.
- You can also view this relationship for different groups of data simple by colour coding the groups as seen in the first figure below.
- Want to visualize the relationship between three variables? No problemo! Just use another parameters, like point size, to encode that third variable as we can see in the second figure below. b



Scatter Plots

```
plt.scatter(x,y)
```

```
<matplotlib.collections.PathCollection at 0x1f8b95c7e48>
```



Scatter Plots

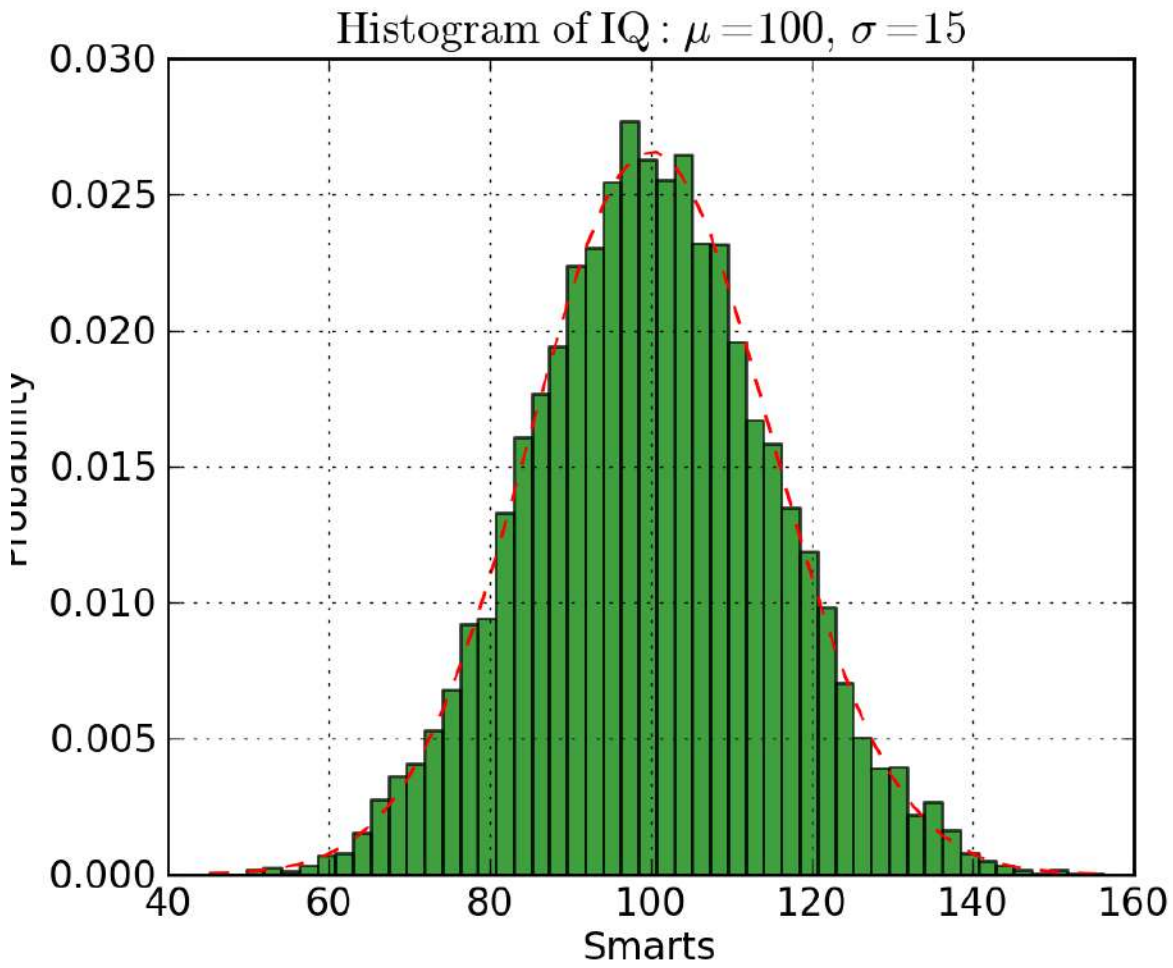
- We can also set the point size, point color, and alpha transparency.
- You can even set the y-axis to have a logarithmic scale.
- The title and axis labels are then set specifically for the figure.
- That's an easy to use function that creates a scatter plot end to end!

```python
import matplotlib.pyplot as plt
import numpy as np

def scatterplot(x_data, y_data, x_label="", y_label="", title="", color = "r", yscale_log=F

    # Create the plot object
    _, ax = plt.subplots()

    # Plot the data, set the size (s), color and transparency (alpha)
    # of the points
    ax.scatter(x_data, y_data, s = 10, color = color, alpha = 0.75)

    if yscale_log == True:
        ax.set_yscale('log')

    # Label the axes and provide a title
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
```

scatter_plot.py hosted with ❤ by GitHub view raw

Histograms Plots

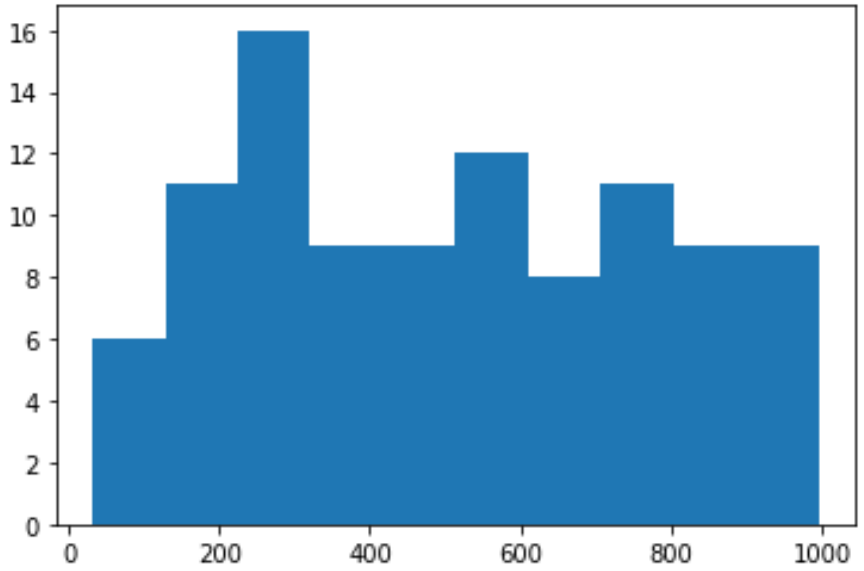
- Histograms are useful for viewing (or really discovering)the distribution of data points.
- Check out the histogram below where we plot the frequency vs IQ histogram.
- We can clearly see the concentration towards the center and what the median is.
- We can also see that it follows a Gaussian distribution.
- Using the bars (rather than scatter points, for example) really gives us a clearly visualization of the relative difference between the frequency of each bin.
- The use of bins (discretization) really helps us see the "bigger picture" where as if we use all of the data points without discrete bins, there would probably be a lot of noise in the visualization, making it hard to see what is really going on.



Histograms Plots

```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

```
(array([ 6., 11., 16.,  9.,  9., 12.,  8., 11.,  9.,  9.]),
 array([ 32. , 128.3, 224.6, 320.9, 417.2, 513.5, 609.8, 706.1, 802.4,
        898.7, 995. ]),
 <a list of 10 Patch objects>)
```



Histograms Plots

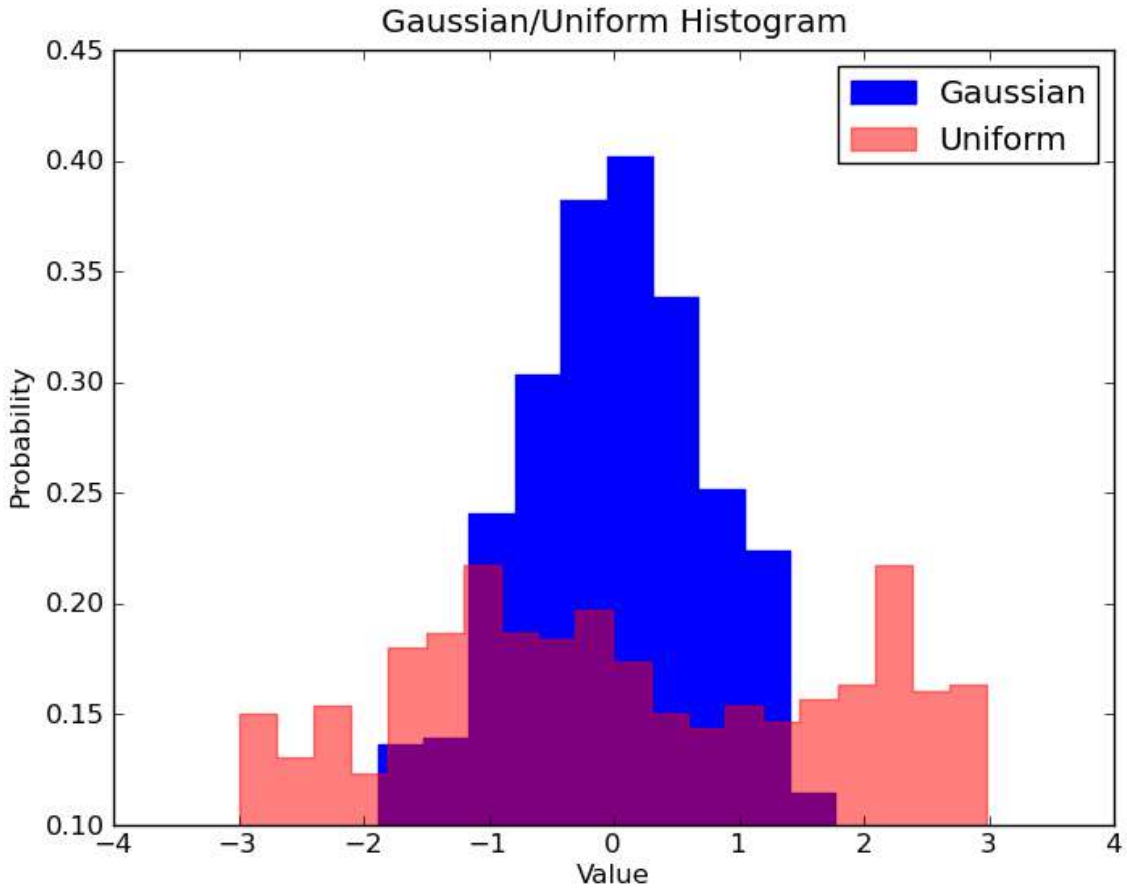
- The code for the histogram in Matplotlib is shown below. There are two parameters to take note of.
- Firstly, the n_bins parameters controls how many discrete bins we want for our histogram.
- More bins will give us finer information but may also introduce noise and take us away from the bigger picture; on the other hand, less bins gives us a more "birds eye view" and a bigger picture of what's going on without the finer details.
- Secondly, the cumulative parameter is a boolean which allows us to select whether our histogram is cumulative or not. This is basically selecting either the Probability Density Function (PDF) or the Cumulative Density Function (CDF).

```python
def histogram(data, n_bins, cumulative=False, x_label = "", y_label = "", title = ""):
    _, ax = plt.subplots()
    ax.hist(data, n_bins = n_bins, cumulative = cumulative, color = '#539caf')
    ax.set_ylabel(y_label)
    ax.set_xlabel(x_label)
    ax.set_title(title)
```

histogram.py hosted with ❤ by GitHub view raw

Histograms Plots

- Imagine we want to compare the distribution of two variables in our data.
- One might think that you'd have to make two separate histograms and put them side-by-side to compare them.
- But, there's actually a better way: we can overlay the histograms with varying transparency.
- Check out the figure below. The Uniform distribution is set to have a transparency of 0.5 so that we can see what's behind it. This allows use to directly view the two distributions on the same figure.



Histograms Plots

- There are a few things to set up in code for the overlaid histograms.
- First, we set the horizontal range to accommodate both variable distributions.
- According to this range and the desired number of bins we can actually computer the width of each bin.
- Finally, we plot the two histograms on the same plot, with one of them being slightly more transparent.

```python
# Overlay 2 histograms to compare them
def overlaid_histogram(data1, data2, n_bins = 0, data1_name="", data1_color="#539caf", data
    # Set the bounds for the bins so that the two distributions are fairly compared
    max_nbins = 10
    data_range = [min(min(data1), min(data2)), max(max(data1), max(data2))]
    binwidth = (data_range[1] - data_range[0]) / max_nbins


    if n_bins == 0
        bins = np.arange(data_range[0], data_range[1] + binwidth, binwidth)
    else:
        bins = n_bins

    # Create the plot
    _, ax = plt.subplots()
    ax.hist(data1, bins = bins, color = data1_color, alpha = 1, label = data1_name)
    ax.hist(data2, bins = bins, color = data2_color, alpha = 0.75, label = data2_name)
    ax.set_ylabel(y_label)
    ax.set_xlabel(x_label)
    ax.set_title(title)
    ax.legend(loc = 'best')
```

overlaid_histogram.py hosted with ❤ by GitHub view raw

Bar Plots

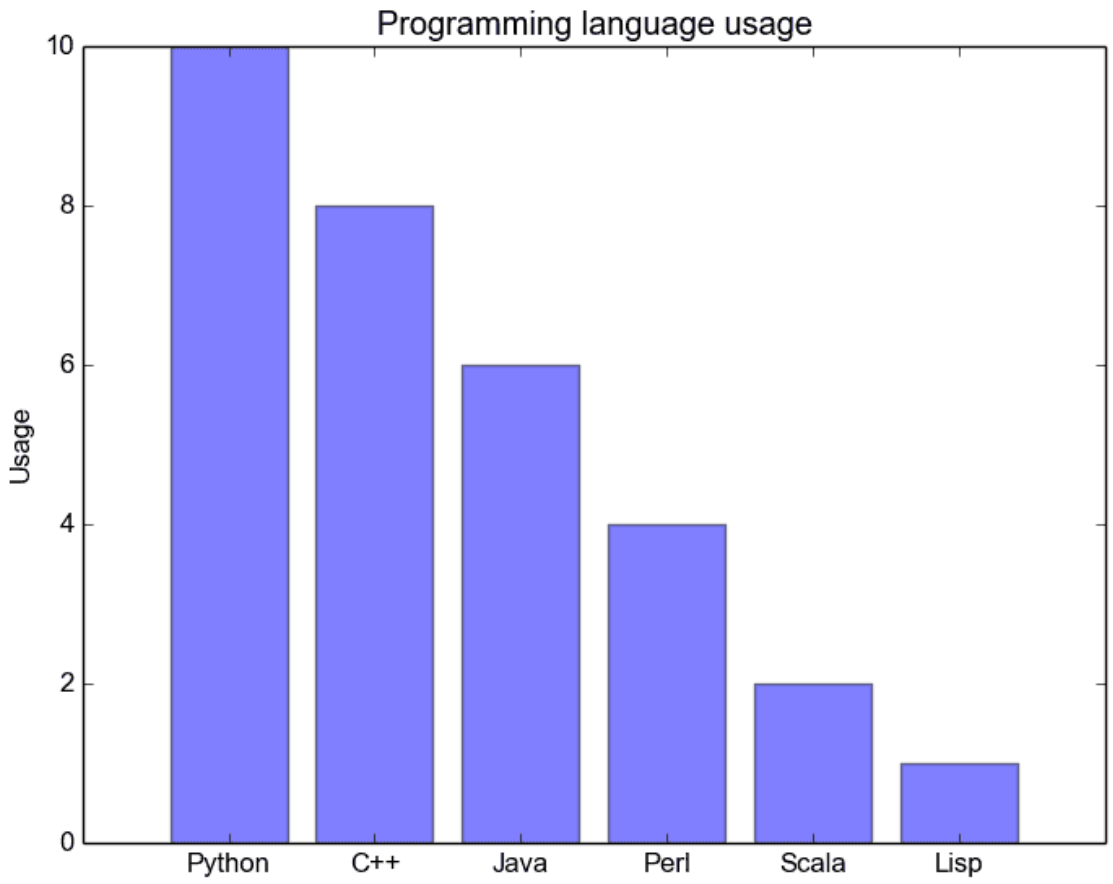
- Bar plots are most effective when you are trying to visualize categorical data that has few (probably < 10) categories.
- If we have too many categories then the bars will be very cluttered in the figure and hard to understand.
- They're nice for categorical data because you can easily see the difference between the categories based on the size of the bar (i.e magnitude); categories are also easily divided and colour coded too.
- There are 3 different types of bar plots we're going to look at: regular, grouped, and stacked. Check out the code below the figures as we go along.

Bar Plots

- The regular barplot is in the next figure.
- In the barplot() function, x_data represents the tickers on the x-axis and y_data represents the bar height on the y-axis.
- The error bar is an extra line centered on each bar that can be drawn to show the standard deviation.
- Grouped bar plots allow us to compare multiple categorical variables. Check out the second bar plot below.
- The first variable we are comparing is how the scores vary by group (groups G1, G2, ... etc). We are also comparing the genders themselves with the colour codes.
- Taking a look at the code, the y_data_list variable is now actually a list of lists, where each sublist represents a different group. We then loop through each group, and for each group we draw the bar for each tick on the x-axis; each group is also colour coded.

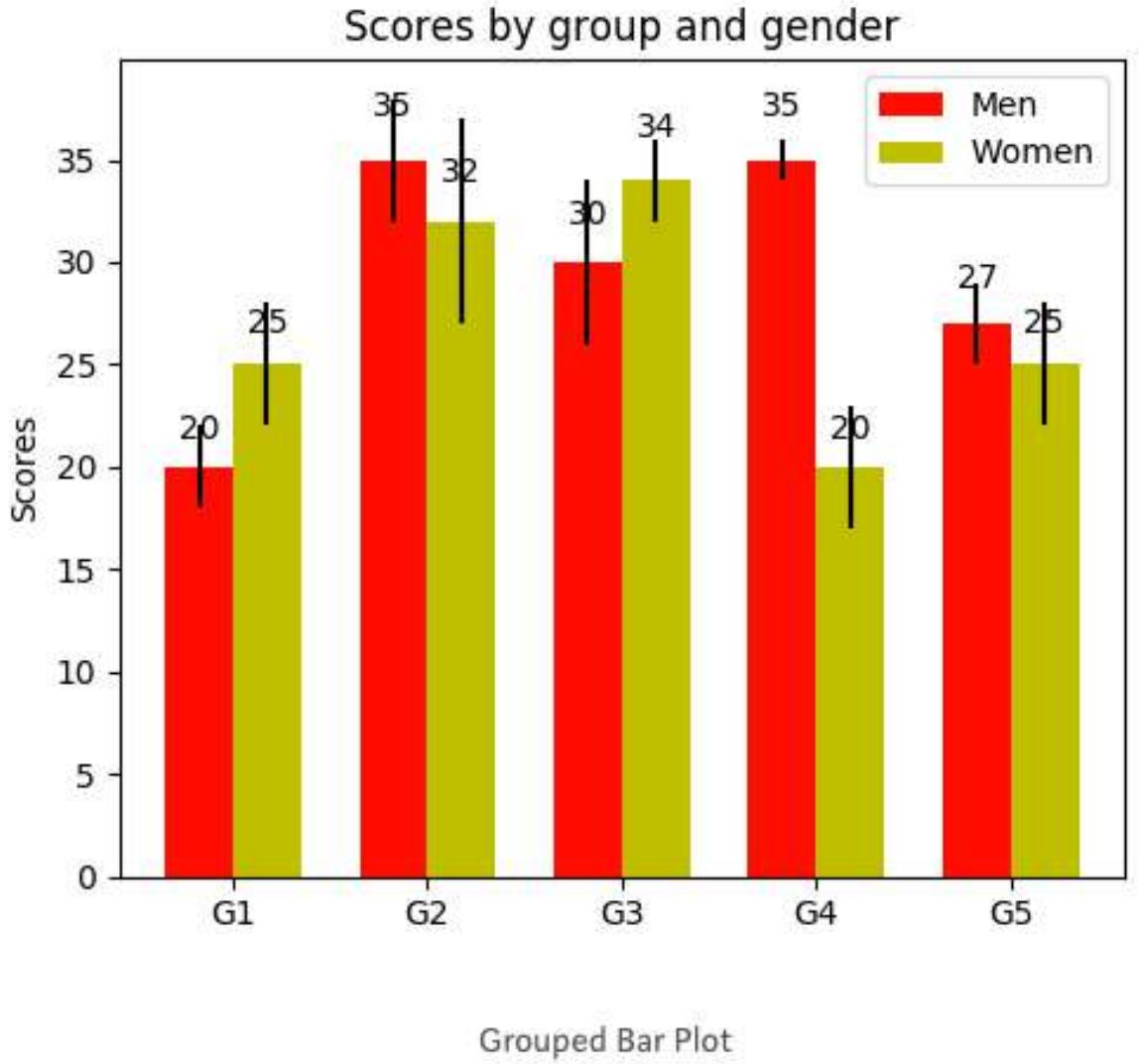
Bar Plots

- Stacked bar plots are great for visualizing the categorical make-up of different variables.
- In the stacked bar plot figure below we are comparing the server load from day-to-day.
- With the colour coded stacks, we can easily see and understand which servers are worked the most on each day and how the loads compare to the other servers on all days.
- The code for this follows the same style as the grouped bar plot.
- We loop through each group, except this time we draw the new bars on top of the old ones rather than beside them.



Bar Plots

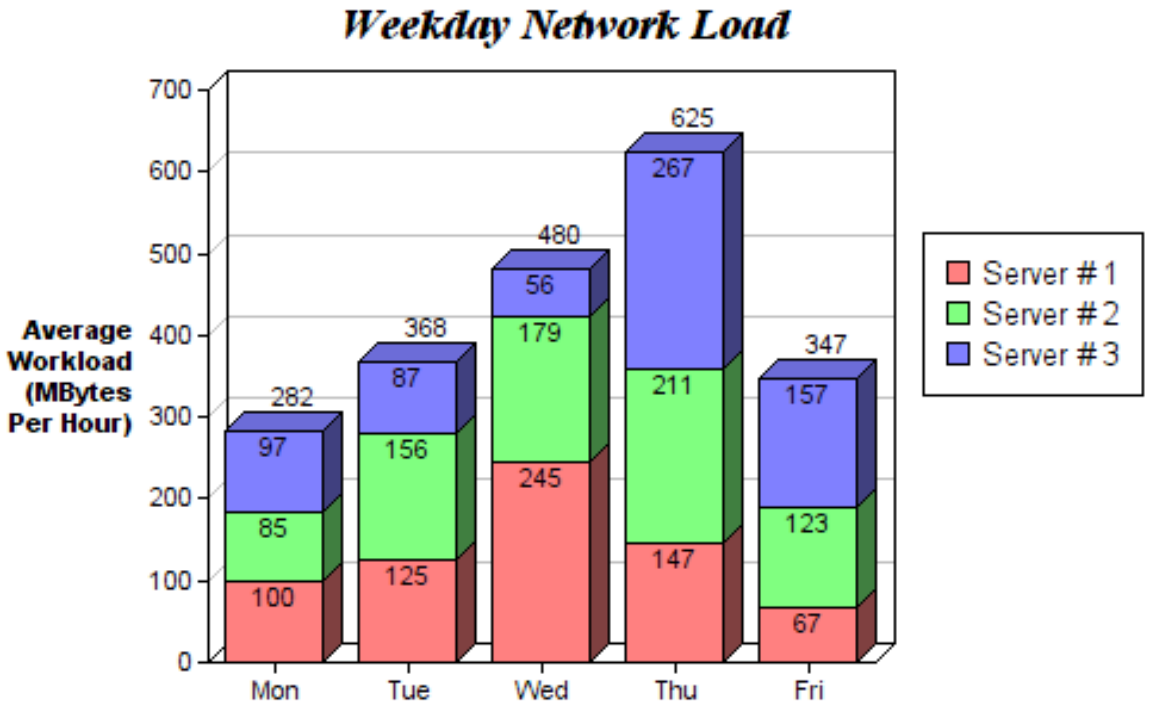
- Stacked bar plots are great for visualizing the categorical make-up of different variables.
- In the stacked bar plot figure below we are comparing the server load from day-to-day.
- With the colour coded stacks, we can easily see and understand which servers are worked the most on each day and how the loads compare to the other servers on all days.
- The code for this follows the same style as the grouped bar plot.
- We loop through each group, except this time we draw the new bars on top of the old ones rather than beside them.



Grouped Bar Plot

Bar Plots

- Stacked bar plots are great for visualizing the categorical make-up of different variables.
- In the stacked bar plot figure below we are comparing the server load from day-to-day.
- With the colour coded stacks, we can easily see and understand which servers are worked the most on each day and how the loads compare to the other servers on all days.
- The code for this follows the same style as the grouped bar plot.
- We loop through each group, except this time we draw the new bars on top of the old ones rather than beside them.



Stacked Bar Plot

Bar Plots

```
1  def barplot(x_data, y_data, error_data, x_label="", y_label="", title=""):
2      _, ax = plt.subplots()
3      # Draw bars, position them in the center of the tick mark on the x-axis
4      ax.bar(x_data, y_data, color = '#539caf', align = 'center')
5      # Draw error bars to show standard deviation, set ls to 'none'
6      # to remove line between points
7      ax.errorbar(x_data, y_data, yerr = error_data, color = '#297083', ls = 'none', lw = 2,
8      ax.set_ylabel(y_label)
9      ax.set_xlabel(x_label)
10     ax.set_title(title)
11
```

Bar Plots

```
14  def stackedbarplot(x_data, y_data_list, colors, y_data_names="", x_label="", y_label="", t
15      _, ax = plt.subplots()
16      # Draw bars, one category at a time
17      for i in range(0, len(y_data_list)):
18          if i == 0:
19              ax.bar(x_data, y_data_list[i], color = colors[i], align = 'center', label = y_
20          else:
21              # For each category after the first, the bottom of the
22              # bar will be the top of the last category
23              ax.bar(x_data, y_data_list[i], color = colors[i], bottom = y_data_list[i - 1],
24      ax.set_ylabel(y_label)
25      ax.set_xlabel(x_label)
26      ax.set_title(title)
27      ax.legend(loc = 'upper right')
```

Bar Plots

```
31  def groupedbarplot(x_data, y_data_list, colors, y_data_names="", x_label="", y_label="", t
32      _, ax = plt.subplots()
33      # Total width for all bars at one x location
34      total_width = 0.8
35      # Width of each individual bar
36      ind_width = total_width / len(y_data_list)
37      # This centers each cluster of bars about the x tick mark
38      alteration = np.arange(-(total_width/2), total_width/2, ind_width)
39
40      # Draw bars, one category at a time
41      for i in range(0, len(y_data_list)):
42          # Move the bar to the right on the x-axis so it doesn't
43          # overlap with previously drawn ones
44          ax.bar(x_data + alteration[i], y_data_list[i], color = colors[i], label = y_data_na
45      ax.set_ylabel(y_label)
46      ax.set_xlabel(x_label)
47      ax.set_title(title)
48      ax.legend(loc = 'upper right')
```

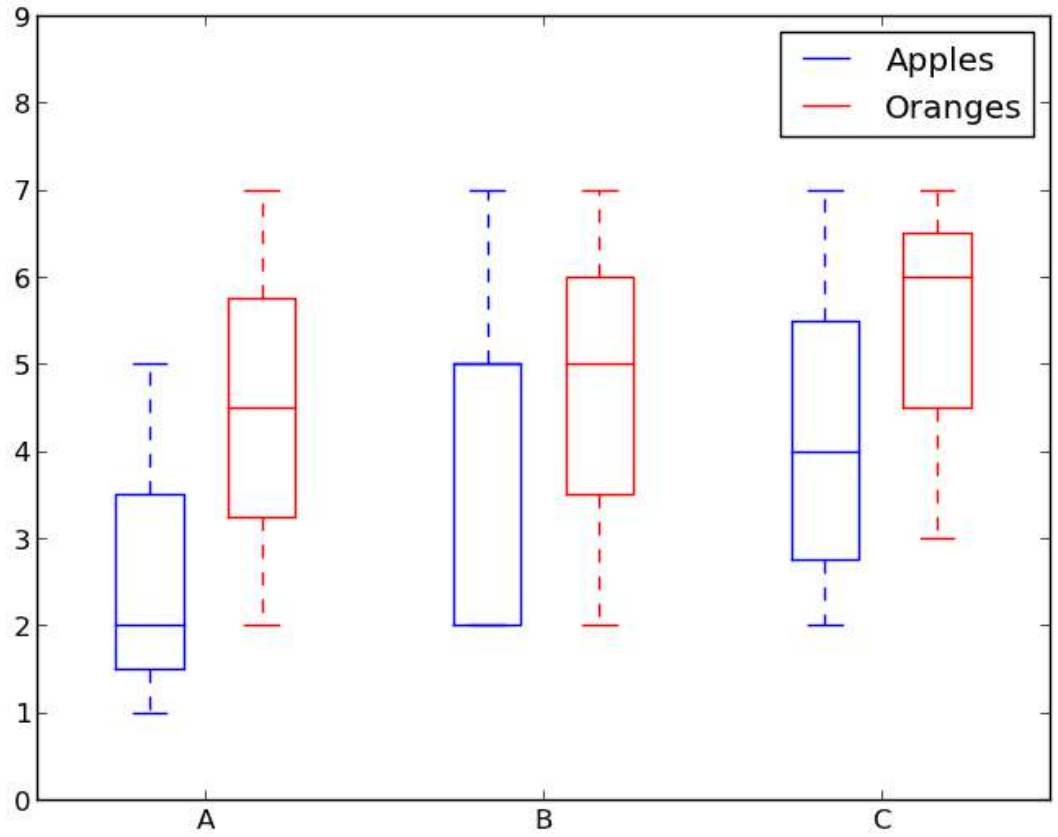
barplot.py hosted with ❤ by GitHub view raw

Box Plots

- We previously looked at histograms which were great for visualizing the distribution of variables.
- But what if we need more information than that? Perhaps we want a clearer view of the standard deviation? Perhaps the median is quite different from the mean and thus we have many outliers? What if there is so skew and many of the values are concentrated to one side?
- That's where boxplots come in. Box plots give us all of the information above.
- The bottom and top of the solid-lined box are always the first and third quartiles (i.e 25% and 75% of the data), and the band inside the box is always the second quartile (the median).
- The whiskers (i.e the dashed lines with the bars on the end) extend from the box to show the range of the data.

Box Plots

- Since the box plot is drawn for each group/variable it's quite easy to set up. The x_data is a list of the groups/variables.
- The Matplotlib function boxplot() makes a box plot for each column of the y_data or each vector in sequence y_data; thus each value in x_data corresponds to a column/vector in y_data.
- All we have to set then are the aesthetics of the plot.



Box Plots

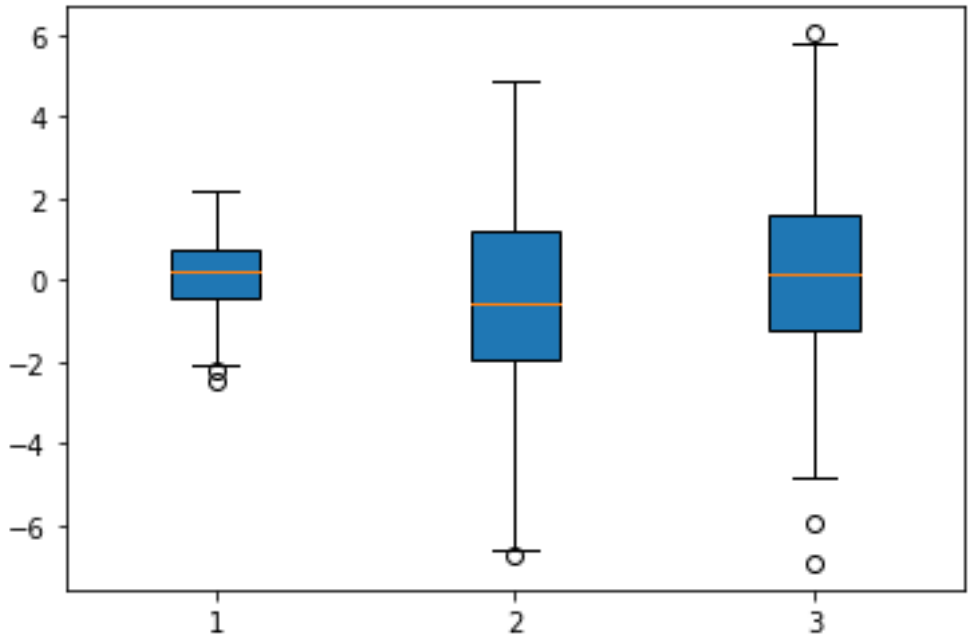
- Since the box plot is drawn for each group/variable it's quite easy to set up. The x_data is a list of the groups/variables.
- The Matplotlib function boxplot() makes a box plot for each column of the y_data or each vector in sequence y_data; thus each value in x_data corresponds to a column/vector in y_data.
- All we have to set then are the aesthetics of the plot.

```
def boxplot(x_data, y_data, base_color="#539caf", median_color="#297083", x_label="", y_lab
    _, ax = plt.subplots()

    # Draw boxplots, specifying desired style
    ax.boxplot(y_data
               # patch_artist must be True to control box fill
               , patch_artist = True
               # Properties of median line
               , medianprops = {'color': median_color}
               # Properties of box
               , boxprops = {'color': base_color, 'facecolor': base_color}
               # Properties of whiskers
               , whiskerprops = {'color': base_color}
               # Properties of whisker caps
               , capprops = {'color': base_color})

    # By default, the tick label starts at 1 and increments by 1 for
    # each box drawn. This sets the labels to the ones we want
    ax.set_xticklabels(x_data)
    ax.set_ylabel(y_label)
    ax.set_xlabel(x_label)
    ax.set_title(title)
```

boxplot.py hosted with ❤ by GitHub view raw

Box Plots

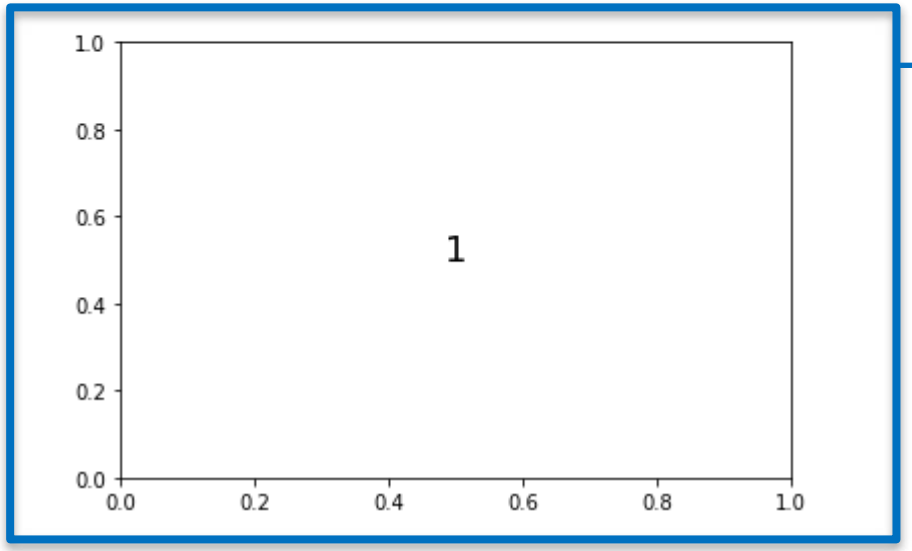
```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data,vert=True,patch_artist=True);
```



Matplotlib Structure

```
plt.text(
        0.5,
        0.5,
        1,
        fontsize=18,
        ha='center'
    )
```

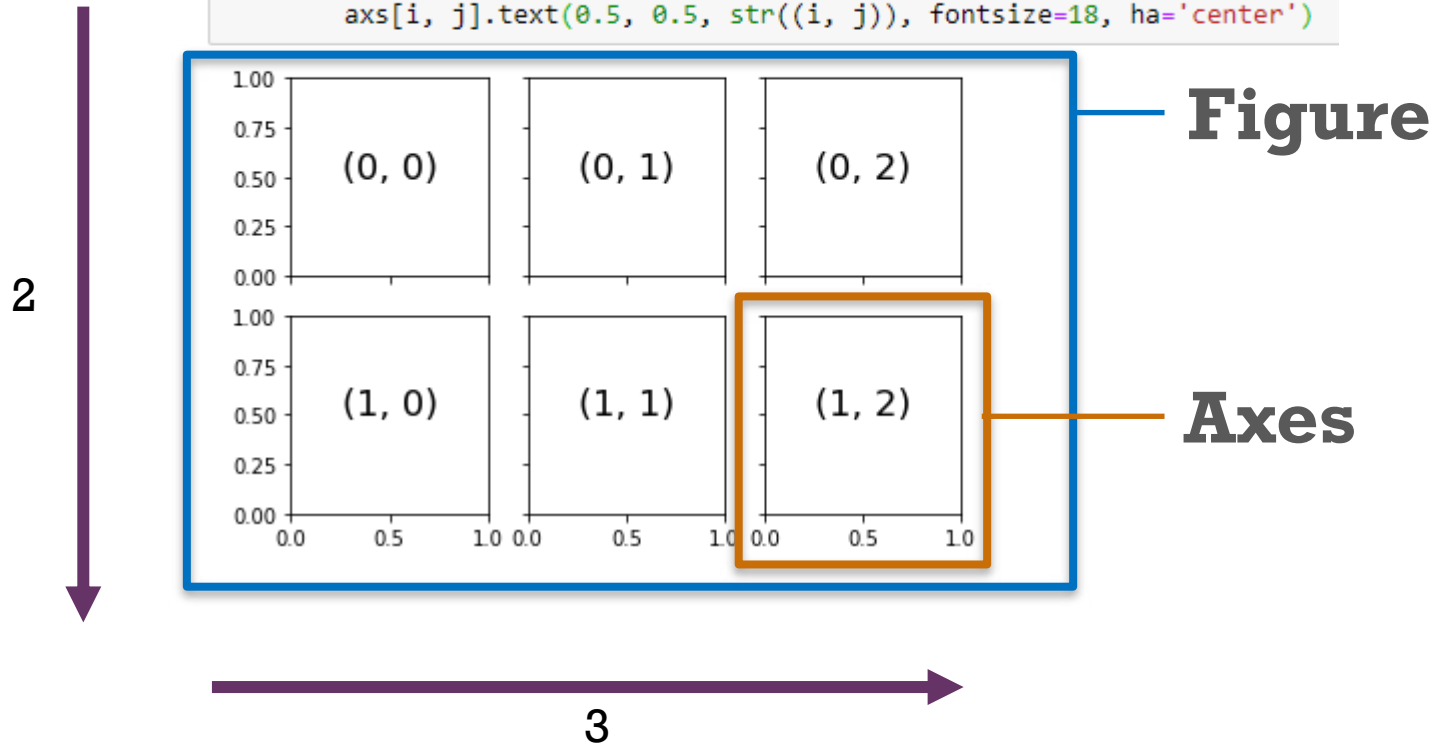
```
Text(0.5, 0.5, '1')
```



Figure

Matplotlib Structure

```
# Add axe to figure
fig, axs = plt.subplots(2, 3, sharex='col', sharey='row')
for i in range(2):
    for j in range(3):
        axs[i, j].text(0.5, 0.5, str((i, j)), fontsize=18, ha='center')
```



Any Questions?