



Data Visualization Basics in Python

Disclaimer: These notes and examples are an adaptation of the references listed at the end. They are compiled to fit the scope of this specific course.

Introduction

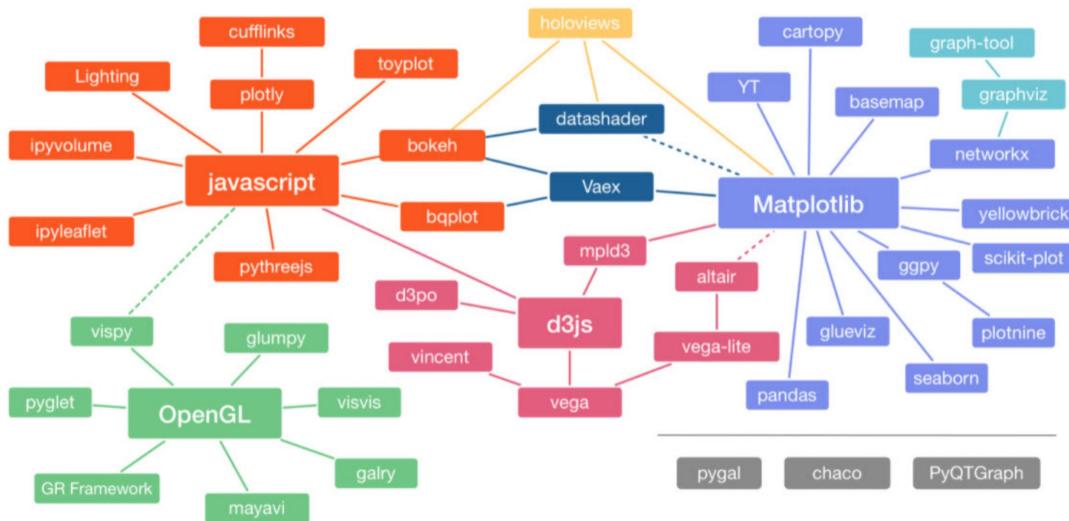
Python offers a variety of tools and packages for visualization. The following is a list of popular packages available in python for visualization

- `matplotlib`
- `seaborn`
- `pandas`
- `altair`
- `bokeh`
- `plotly`

Understanding which package to use, how to use it, and what graph to use for visualizing your data can be challenging. In this introductory handout, we will learn about `matplotlib`, `seaborn`, and `pandas`, three of the most commonly used visualization packages in python.

Python visualization landscape

The <https://pyviz.org/index.html> website provides an accurate image of python's visualization tools. This platform helps users decide on the best python data visualization tools and their purposes with links, overviews, comparisons, and examples. The most important part of this landscape is noticing the libraries' inner dependencies. The following graphics is taken from <https://pyviz.org/overviews/index.html>.



As you can see, many python visualization libraries heavily depend on `matplotlib`.

matplotlib

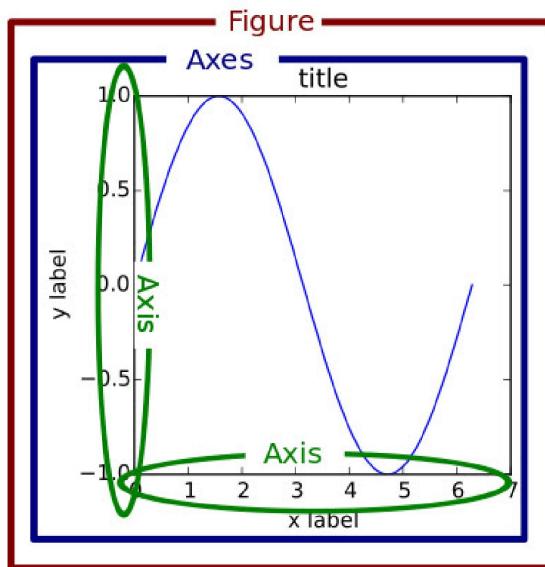
`matplotlib` is the grandfather of python visualization tools! As it is clear from the above landscape, many of python's visualization tools -- most notably `seaborn` -- are just a user-friendly skin (wrapper) over `matplotlib`. `matplotlib` is very powerful and complicated. Even though you can accomplish most of your visualization tasks with `matplotlib`, many users try to avoid using `matplotlib` due to its complexity. However, due to different packages' dependency on `matplotlib`, we will learn the basics before switching to `seaborn` and visualization using `pandas`.

plotting with matplotlib

`matplotlib` (<https://matplotlib.org/>) is python's most widely used visualization library. The **Example** section of this library is available at <https://matplotlib.org/stable/gallery/index.html>. It shows a variety of graphs you can plot using `matplotlib`. Jupyter notebook and python source code for each of these examples are available for download in each example web page.

For `anaconda` users, `matplotlib` is already available. However, try `pip install matplotlib` if you need to install it. Let's understand the anatomy of a plot. Key components of a plot are `Figure` and `Axes`.

- **Figure**: A `Figure` is the top-level container for everything plotted. You can consider `Figure` to be a piece of paper or a canvas you can draw and write on.
- **Axes**: Contained within a `Figure` is one or more `Axes`. An `Axes` is a plotting surface. An `Axes` is where we plot our data, and add all the associated labels/ticks. Each `Axes` has one `X-Axis` and one `Y-Axis` (and possibly one `Z-Axis`). The following shows the anatomy of a plot.



Note 1: `Axes` is NOT a plural of `Axis`; they are entirely separate concepts. An `Axes` is the plotting surface and may contain `X-Axis`, `Y-Axis`, or `Z-Axis`.

Note 2: Please do not confuse the concept of `axis` in `numpy`, `pandas`, and `Axes` in `matplotlib` as they are very different concepts. In `numpy` and `pandas`, the setting of `axis = 0` or `axis = 1` is used to dictate calculations for columns or rows; whereas, in `matplotlib` `Axes` is a plotting area and `Axis` is for X, Y, or Z axis.

Plotting in `matplotlib` is done using its `pyplot` module. `matplotlib` suggests importing `pyplot` using

```
import matplotlib.pyplot as plt
```

- Older versions of `Jupyter notebook` users: you need to include the line `%matplotlib inline` in your notebook. This will direct `matplotlib` to display plots in the notebook.
- If you are using other IDEs such as `pycharm` or `spyder`, you need to add `plt.show()` instead.

There are two approaches for creating plots in `matplotlib` as

- **functional approach**
- **Object-oriented approach**

Rule of thumb: Almost all `matplotlib` plot types (linear, bar, histogram, etc.) accept arrays (`python` arrays, `numpy` arrays, `pandas Series`, or `pandas DataFrame`) for values of `x` and `y`.

Use `matplotlib` functional approach for plotting

Here is an example:

Example

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

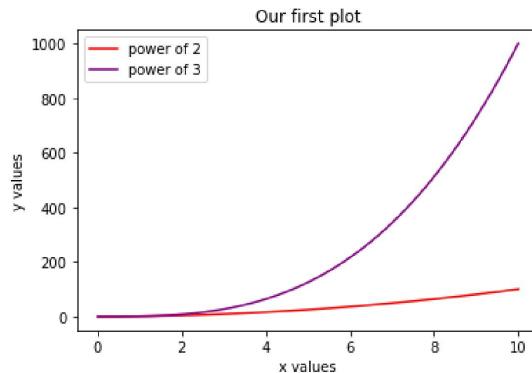
arr_x = np.linspace(0, 10)
arr_y_1 = arr_x ** 2
arr_y_2 = arr_x ** 3

plt_1 = plt.plot(arr_x, arr_y_1, "red", label="power of 2")
plt_2 = plt.plot(arr_x, arr_y_2, "purple", label="power of 3")

# optional but highly recommended features
plt.title("Our first plot")
plt.xlabel("x values")
plt.ylabel("y values")

# default labels
plt.legend()
```

yields a plot with two lines plotted. Legend will also be shown; the default plot legend is the provided plot labels.



`matplotlib` default is plotting just one plot on the canvas. We can create multi-plots on the same figure (canvas) using `.subplot` method. This method has three arguments:

- `nrows` : number of rows in the `Figure`
 - `ncols` : number of columns in the `Figure`
 - `plot_number` : refers to a specific plot in the `Figure`

The following snippet will create six subplots, plot different functions in each subplot.

Example

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

arr_x = np.linspace(2, 10)
arr_y_1_1 = np.sin(arr_x)
arr_y_1_2 = np.cos(arr_x)
arr_y_1_3 = np.log(arr_x)
arr_y_2_1 = np.tan(arr_x)
arr_y_2_2 = np.sin(arr_x)
arr_y_2_3 = np.sinc(arr_x)

plt.subplot(2, 3, 1)
plt.plot(arr_x, arr_y_1_1, "navy")
```

```
plt.subplot(2, 3, 2)
plt.plot(arr_x, arr_y_1_2, "red")

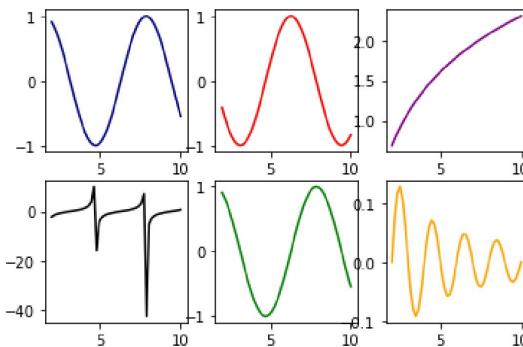
plt.subplot(2, 3, 3)
plt.plot(arr_x, arr_y_1_3, "purple")

plt.subplot(2, 3, 4)
plt.plot(arr_x, arr_y_2_1, "black")

plt.subplot(2, 3, 5)
plt.plot(arr_x, arr_y_2_2, "green")

plt.subplot(2, 3, 6)
plt.plot(arr_x, arr_y_2_3, "orange")
```

yields six plots in one `Figure`.



Note: If your plots are not showing, please add the magic function `%matplotlib inline` to your cell. So the top part of your cell will look like

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Use `matplotlib` object oriented approach for plotting

This is the most common way of creating plots. The main idea is to create a `Figure` object, then make one or more `Axes` in that `Figure` object, and call the necessary methods off the `Figure` and `Axes` to customize the contents of a plot.

Use `plt.figure()`

Here is an example

Example

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

# create data
arr_x = np.linspace(2, 10)
arr_y_1 = np.sinc(arr_x)

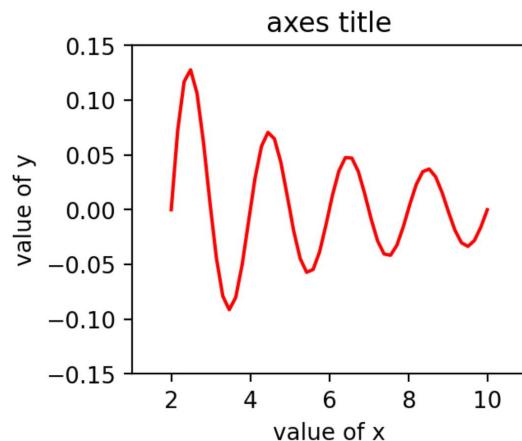
#create a figure. This is similar to a canvas surface
fig_1 = plt.figure()

# create an empty plot. We call it axes
axes_1 = fig_1.add_axes([0.1, 0.2, 0.4, 0.5])

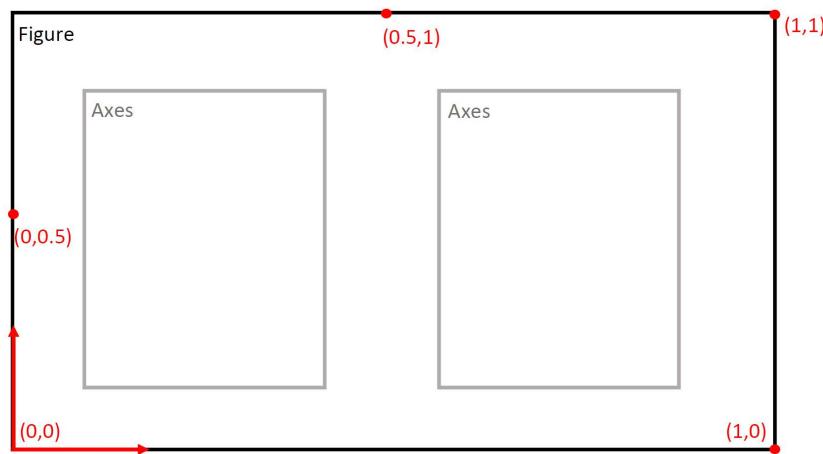
# plot in axes_1
axes_1.plot(arr_x, arr_y_1, "red")

# set axes_1 settings
axes_1.set(
    xlim=(1, 11),
    ylim=(-0.15, 0.15),
    xlabel="value of x",
    ylabel="value of y",
    title="axes title",
)
```

creates a `figure` called `fig_1`; adds an `Axes` called `axes_1` at `x_0 = 0.1`, `y_0 = 0.2`, `width = 0.4`, and `height = 0.5`; plots a red curve, and sets specific settings for the plot

**Explanation:**

- `fig_1 = plt.figure()` creates a `Figure` object and calls it `fig_1`. `Figure` coordinates go `0` to `1`, where `(0,0)` is the lower left corner and `(1,1)` is the upper right corner. A coordinate of `y=1.05` is hence slightly outside the figure.



- `axes_1 = fig_1.add_axes([0.1, 0.2, 0.4, 0.5])` creates an `Axes` object. The syntax of `add_axes([x_0, y_0, w, h])` creates an `Axes`; (x_0, y_0) is the lower left point of the new axes in figure coordinates, w is its width and h is its height. So the axes are positioned in absolute coordinates on the canvas.
- `axes_1.plot(arr_x, arr_y_1, "red")` plots a red curve based on the given data points.
- `axes_1.set()` method is used to set `axes_1` settings.

Adding one more Axes: In the following snippet, we will add one more `Axes` to the same canvas (`Figure` object) and plot two different functions. This time we will add more settings for the `Figure`, and finally, we will save it.

Example

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

# create data
arr_x = np.linspace(2, 10)
arr_y_1 = np.sinc(arr_x)
arr_y_2 = np.log(arr_x)

# create a figure. This is similar to a canvas surface
fig_1 = plt.figure()

# create empty plots
```

```
axes_1 = fig_1.add_axes([0.1, 0.0, 0.4, 0.25])
axes_2 = fig_1.add_axes([0.0, 0.5, 0.3, 0.4])

# plot in axes_1
axes_1.plot(arr_x, arr_y_1, "red")

# plot in axes_2
axes_2.plot(arr_x, arr_y_2, "purple")

# set axes_1 settings
axes_1.set(
    xlim=(1, 11),
    ylim=(-0.15, 0.15),
    xlabel="value of x_1",
    ylabel="value of y_1",
    title="axes_1 title",
)

# set axes_2 settings
axes_2.set(
    xlim=(1, 11),
    ylim=(0.5, 3),
    xlabel="value of x_2",
    ylabel="value of y_2",
    title="axes_2 title",
)

# figure settings
fig_1.suptitle("figure title", size=20, x=0.2, y=1.1)
```

in this snippet, we will add two `Axes` to the `Figure` object; customize their settings; and finally, add a figure title with a font size of 20 at coordinates of (0.2, 1.1) relative to the lower left corner of the canvas

The default size of a canvas (`Figure`) in `matplotlib` is `(6.4, 4.8)` inches (6.4 inches in width and 4.8 inches in height) which is not the actual size of it on the screen but would be the exact size if you saved the `Figure` to a file (with a dpi of 100 pixels per inch); we can confirm that by calling `fig_1.get_size_inches()`. We can further customize that by calling `fig_1.set_size_inches(12,5)`. Additionally, you can save an image using `savefig()` method, as well. The following will illustrate these points

```
import matplotlib.pyplot as plt

# create a figure (plotting area)
fig = plt.figure()
```

yields

```
| <Figure size 640x480 with 0 Axes>
```

indicating that, by default, `fig` has $6.4 \times 100 = 640$ pixels across the x-axis and $4.8 \times 100 = 480$ pixels across the y-axis.

trying

```
| ````python size_in_inch = fig.get_size_inches() size_in_inch # size is the figure dimensions in inches (6.4 by 4.8)
```

yields

```
| array([6.4, 4.8])
```

and trying

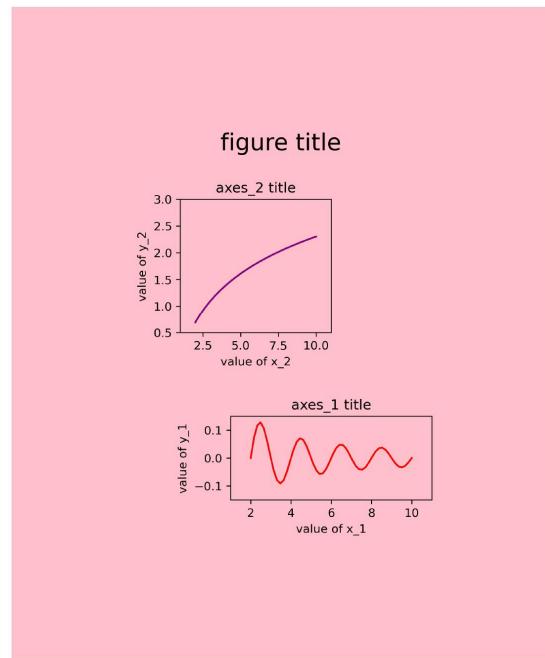
```
| dot_per_inch = fig.dpi # number of dots per inch
| dot_per_inch
```

yields

```
| 100.0
```

as mentioned earlier, the above attributes can be modified. Here is an example

```
# save the file
fig_1.savefig(
    "save_1.png",           # file name
    bbox_inches="tight",     # saves the file in a tight fit
    pad_inches=1.5,          # space around the figure
    transparent=True,        # transparency of axes
    facecolor="pink",         # face color is pink
    orientation="landscape", # image orientation
    dpi = 300                # dot per inch
)
```



- `dpi` stands for *dot per inch*. This value dictates the figure resolution in dots per inch; a higher number means higher resolution (and bigger image file size). For most applications, `dpi` in the range of 200 to 400 suffices.

Use `plt.subplots(nrows, ncols)`

We can create multiple plots in the object-oriented approach using the `.subplots()` method and NOT `.subplot()`. The `.subplots()` method takes two arguments, `nrows`, for the number of rows in the `Figure` and `ncols`, for the number of columns in the `Figure`.

Example

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

# generate data
```

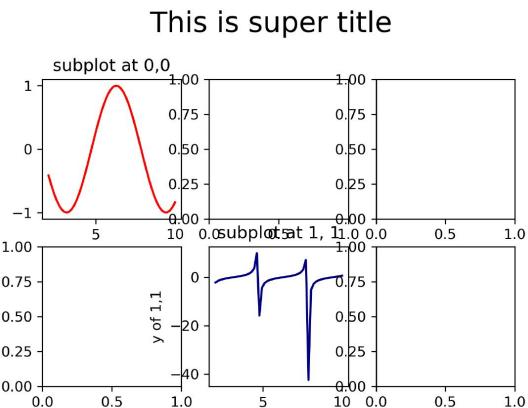
```
arr_x = np.linspace(2, 10)
arr_y_1 = np.cos(arr_x)
arr_y_2 = np.tan(arr_x)

fig, axes = plt.subplots(nrows=2, ncols=3)

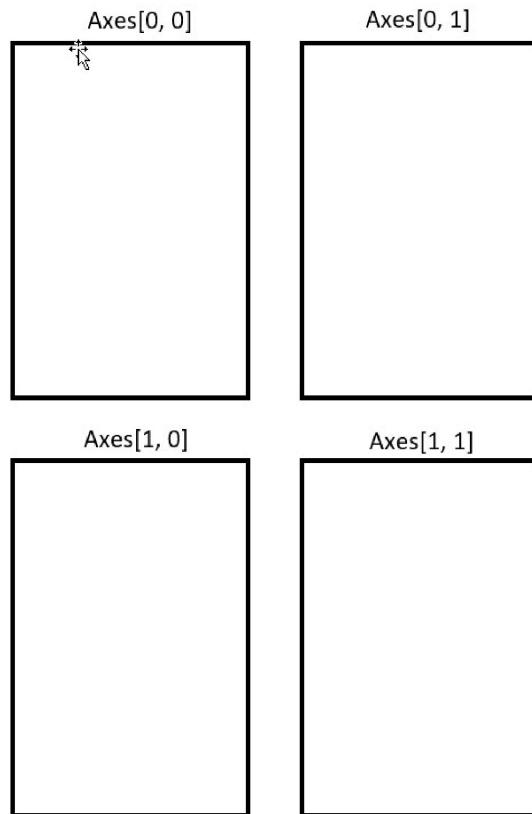
axes[0, 0].plot(arr_x, arr_y_1, "red")
axes[1, 1].plot(arr_x, arr_y_2, "navy")

axes[0, 0].set(xlabel="x of 0,0", title="subplot at 0,0")
axes[1, 1].set(ylabel="y of 1,1", title="subplot at 1, 1")

fig.suptitle("This is super title", size=20, x=0.5, y=1.05)
```



Explanation: The `.subplots(nrows, ncols)` function returns a two-item tuple object containing a `Figure` and one or more `Axes` objects (here 6 `Axes`). The second item in the tuple of `.subplots(nrows, ncols)` is a `numpy` array containing all the `Axes`. You can check that by calling `axes`. We can use 0-based indexing to access each `Axes` and customize the plot and its features. The following shows the location of each `Axes` when using the `.subplots()` method:



Setting up the spacing between subplots

Subplots in `Matplotlib` are multiple plots within the same `figure`. As you can see, there is an overlapping issue in the subplots we have created. The `spacing` attribute determines how far apart the subplots are in the figure. We can handle that issue using the `.tight_layout()` method. Subplots will be spaced out further away by adding `plt.tight_layout()`. The following snippet will generate the same `figure` and `subplots`; this time with better spacing between `subplots`.

Example

```

import numpy as np
import matplotlib.pyplot as plt

arr_x = np.linspace(2, 10)
arr_y_1 = np.cos(arr_x)
arr_y_2 = np.tan(arr_x)

fig, axes = plt.subplots(nrows=2, ncols=3)

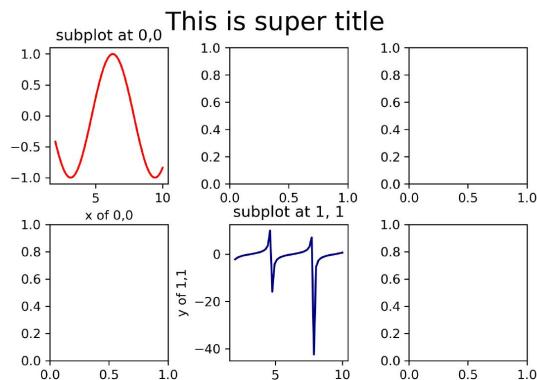
axes[0, 0].plot(arr_x, arr_y_1, "red")
axes[1, 1].plot(arr_x, arr_y_2, "navy")

plt.tight_layout()           # control spacing

axes[0, 0].set(xlabel="x of 0,0", title="subplot at 0,0")
axes[1, 1].set(ylabel="y of 1,1", title="subplot at 1, 1")

fig.suptitle("This is super title", size=20, x=0.5, y=1.05)

```

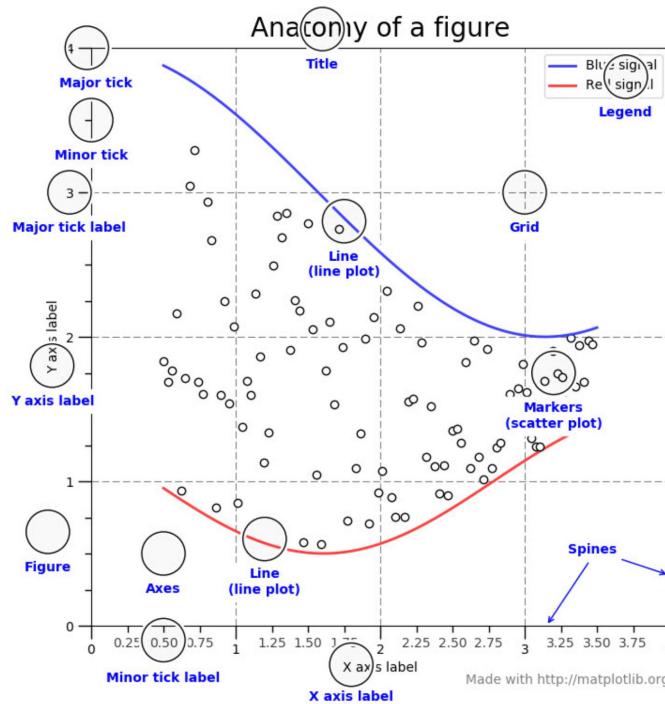


Note: `plt.tight_layout()` has an optional argument of `pad` with a default value of `1.08`. By increasing that number, the spacing between subplots will increase.

Side note: `plt.tight_layout()` has optional arguments of `h_pad` and `w_pad` as well. I will leave exploring these options to you!

Fine-tuning matplotlib figure

One of the best features of `matplotlib` is its capability to customize every possible component of a plot. The following image is taken from the `matplotlib` website and shows a deeper look at the components of a `matplotlib` `figure`.



In this section, we will learn about most important customization we might need for a `figure`. While this is an attractive feature, with this level of flexibility comes complexity, frustration, and confusion. For further `figure` customization, please consult with `matplotlib` 3489-page-long documentation at <https://matplotlib.org/stable/Matplotlib.pdf>.

Let's look at one self-explanatory example:

Example

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

arr_x = np.linspace(2, 10, 100)

# set figure settings at initiation
fig, axes = plt.subplots(nrows=2, ncols=2, figsize = (5, 3), dpi = 200)
plt.tight_layout(pad = 2)

#plotting
axes[0, 0].plot(arr_x, np.sin(arr_x), "red", linewidth = 0.3, marker = 'o', markersize = 1,
label = 'r')

axes[0, 1].plot(arr_x, np.cos(arr_x), "black", label = 'b', linestyle = '--')

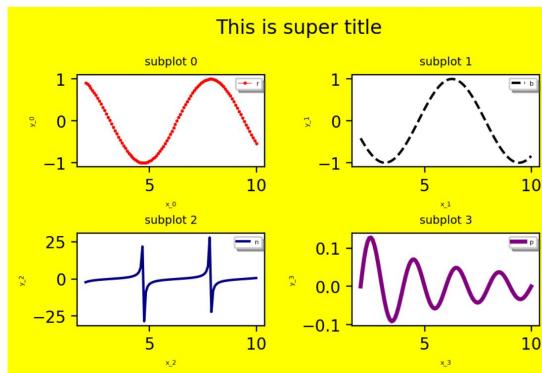
axes[1, 0].plot(arr_x, np.tan(arr_x), "navy", label = 'n')

axes[1, 1].plot(arr_x, np.sinc(arr_x), "purple", linewidth = 2.5, label = 'p')

# common notation for changing settings of all subplots all at once
for i, sub_p in enumerate(fig.axes):
    sub_p.set_xlabel("x_" + str(i), fontsize = 4)
    sub_p.set_ylabel("y_" + str(i), fontsize = 4)
    sub_p.set_title('subplot ' + str(i), fontsize = 7)
    sub_p.legend(loc='upper right', shadow = True, fontsize = 4)

fig.suptitle("This is super title", size=12, x=0.5, y=1.05)

fig.savefig("save_me.png", bbox_inches="tight", facecolor = 'yellow')
```



Note 1: `linestyle` and `marker` arguments are used to control plot appearance. A whole list of what is possible can be found at https://matplotlib.org/stable/api/markers_api.html and https://matplotlib.org/2.0.1/api/lines_api.html.

Note 2: You can generate high quality images from `matplotlib figure`. You can use `.savefig()` method to save a `figure` in a variety of formats such as `jpg`, `jpeg`, `png`, `svg`, `pdf`, etc.

Optional: One can confirm existence of the above generated image file by running the following snippet

Optional

```
import matplotlib.image as mpimg  
plt.imshow(mpimg.imread('save_me.png'))
```

This will read the image file `save_me.png` and will display it.

Commonly used plots in `matplotlib`

A quick look at the `matplotlib` gallery available at <https://matplotlib.org/stable/gallery/index.html#> reveals a wide variety of plots that can be generated using `matplotlib`. However, a small set of relatively limited plot types is used in most practical situations. The rest of this handout will provide quick examples of each of these plot types

Histogram

- **Application:** Displays the `shape` of the data. A histogram is the standard way to show a (statistical) distribution of a numerical value.
- **Common syntax:** Use `.hist()` method with most common syntax of `ax.hist(x, bins = , range = , cumulative =)`

Parameter	Optional?	Description
<code>x</code>	No	array or sequence of arrays
<code>bins</code>	Yes	Most common format: integer number. It defines the number of equal-width bins in the range
<code>range</code>	Yes	A tuple. The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is <code>(x.min(), x.max())</code>
<code>cumulative</code>	Yes	Default: False If set True, it returns cumulative histogram of the data

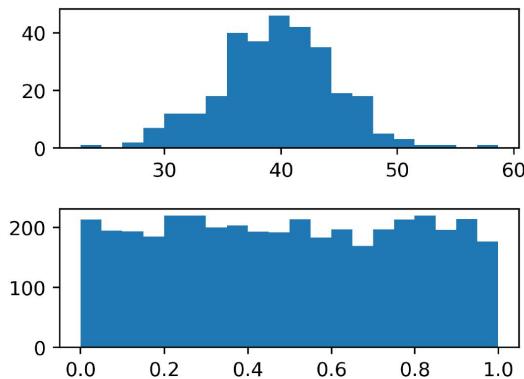
Histogram Example

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

# Generate data
data_1 = np.random.normal(40, 5, size = 300)
data_2 = np.random.random(size = 4000)

fig, axs = plt.subplots(nrows = 2, ncols = 1, figsize = (4, 3), dpi = 150)
plt.tight_layout()

axs[0].hist(data_1, bins=20)
axs[1].hist(data_2, bins=20)
```



Scatter plots

- **Application:** Visualizes how two numerical values are related or affected by each other; this plot helps us understand the relationships between variables.
- **Common syntax:** Use `.scatter()` method with most commonly used syntax of `ax.scatter(x, y, color =)`.

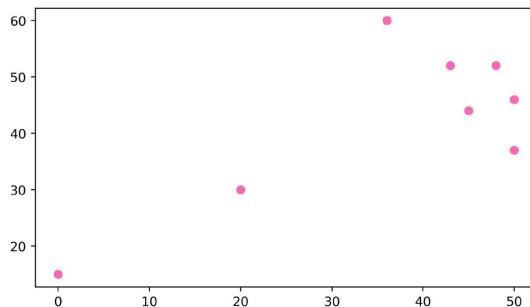
Parameter	Optional?	Description
x	No	sequence of scalars. The data x coordinates
y	No	sequence of scalars. The data y coordinates
color	Yes	Optional array or list of colors or color

Scatter plot Example 1

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

midterm_grades = [43, 20, 50, 45, 36, 0, 50, 48]
final_grades = [52, 30, 37, 44, 60, 15, 46, 52]

fig, ax = plt.subplots(figsize=(7, 4), dpi=100)
ax.scatter(midterm_grades, final_grades, color="hotpink")
```



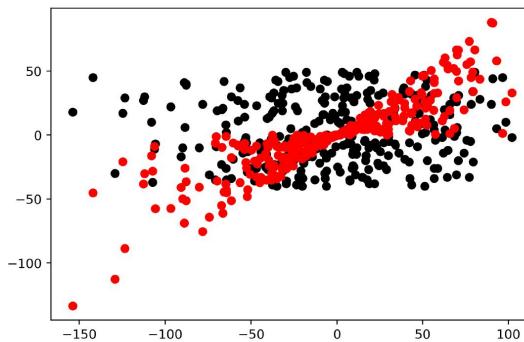
Scatter plot Example 2

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

# Generate data
data_1 = np.random.normal(1, 50, size = 300)
temp_1 = np.random.random(size = 300)
temp_2 = np.random.randint(-40,50, size = 300)
data_2 = data_1 * temp_1

fig, ax = plt.subplots()
plt.tight_layout()

ax.scatter(data_1, temp_2, color = 'black')
ax.scatter(data_1, data_2, color = 'red')
```



Bar chart

- **Application:** Presents categorical data with vertical or horizontal bars with heights or lengths proportional to the values they represent.
- **Common syntax:** Use `.bar()` method with most common syntax of `ax.bar(x, height)`

Parameter	Optional?	Description
<code>x</code>	No	sequence of scalars representing the x coordinates of the bars.
<code>bins</code>	No	scalar or sequence of scalars representing the height(s) of the bars.
<code>width</code>	Yes	The width of the bars. Default = 0.8 scalar or array-like, optional.
<code>bottom</code>	Yes	The y coordinate(s) of the bars bases. default is 0 Used mostly for building stacked bar chart

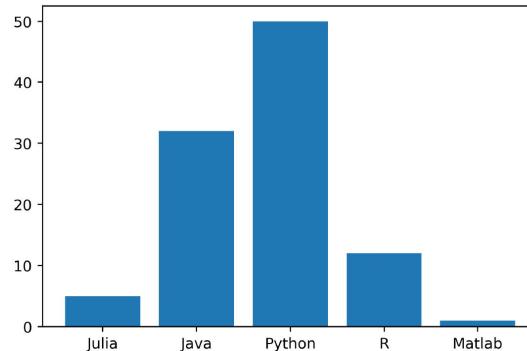
Bar chart Example 1

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

enrollment = [5,32,50,12,1]
```

```
languages = ['Julia', 'Java', 'Python', 'R', 'Matlab']
ax.bar(languages,enrollment)
```



Bar chart Example 2 (taken from [matplotlib](#) website)

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

labels = ['Group 1', 'Group 2', 'Group 3', 'Group 4', 'Group 5']
men_scores = [20, 34, 30, 35, 27]
women_scores = [25, 32, 34, 20, 25]

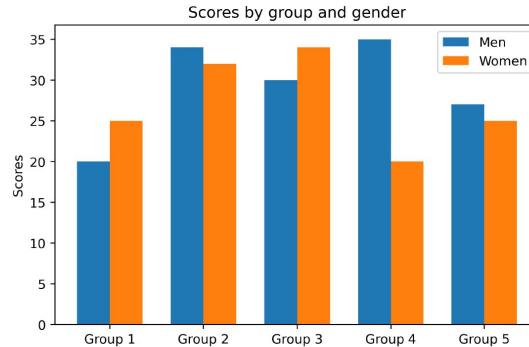
x = np.arange(len(labels)) # the label locations
width = 0.35 # the width of the bars

fig, ax = plt.subplots()

# avoiding overlaps by deducting and adding width/2
bar_1 = ax.bar(x - width/2, men_scores, width, label='Men')
bar_2 = ax.bar(x + width/2, women_scores, width, label='Women')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()
```

```
fig.tight_layout()
```

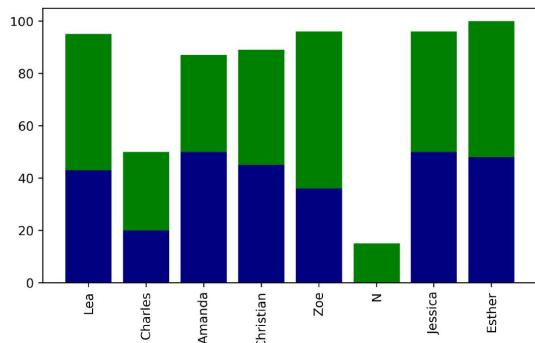


Bar chart Example 3 (stacked bar chart)

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

students = ["Lea", "Charles", "Amanda", "Christian", "Zoe", "N", "Jessica", "Esther"]
midterm_grades = [43, 20, 50, 45, 36, 0, 50, 48]
final_grades = [52, 30, 37, 44, 60, 15, 46, 52]

fig, ax = plt.subplots(figsize=(7, 4), dpi=100)
ax.bar(
    students,
    midterm_grades,
    color="navy",
)
ax.bar(
    students,
    final_grades,
    bottom=midterm_grades,
    color="green",
)
ax.set_xticklabels(students, rotation=90)
```



Note: Use `.barh()` for horizontal bar chart.

seaborn

`Matplotlib` is python's most important visualization library because many other visualization libraries in python depend on `matplotlib`. Though `matplotlib` offers high customization, it involves a lot of coding, so working with `matplotlib` could be very time-consuming. `seaborn` library provides simple, quick (low code), and attractive visualizations, especially for data analysis. One of the most critical aspects of `seaborn` is that it focuses just on the key elements of the charts. `seaborn` integrates well with `pandas`.

`seaborn` is built over `matplotlib` and simplifies many operations. We will use aspects of `matplotlib` in visualizing with `seaborn` because some of the `seaborn` functions return `matplotlib` objects.

`seaborn` documentation suggests importing it as `sns`

```
import seaborn as sns
```

Because of inner dependency of `seaborn` to `matplotlib.pyplot`; it is highly recommended that whenever you use `seaborn`, you should have

```
import matplotlib.pyplot as plt
```

as well.

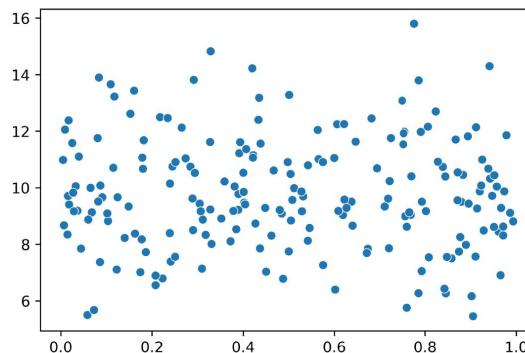
Let's start with a simple scatter plot.

Example

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

arr_1 = np.random.random(size=200)
arr_2 = np.random.normal(loc=10, scale=2, size=200)

axes = sns.scatterplot(x = arr_1, y = arr_2)
```



Note about passing data to `seaborn`: One of the many differences between `seaborn` and `matplotlib` is that in using `seaborn` it is highly recommended to explicitly determine values of `x` and `y`.

Passing `pandas` objects to `seaborn`: Often, we pass columns of a `pandas DataFrame` as values of `x` and `y`. This could be done in at least two different ways:

- `(x = df["col_1"], y = df["col_2"])` where `col_1` and `col_2` are valid columns in `DataFrame` `df`.
- The syntax of `(data = df, x = 'col_1', y = 'col_2')` can work equivalently well.

In the remainder of this handout, we will use the `clean_search_history.csv` data set.

`sns.barplot`

A bar plot is used for visualizing categorical columns vs. numerical columns. Lets create a barplot of `data_science` popularity for each category value (here `0` and `1`) of `categorical` column.

Example

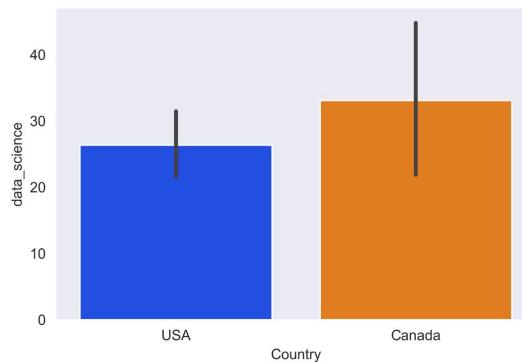
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.barplot(data=df, x="categorical", y="data_science", palette="bright")

axes.set(xlabel="Country", xticklabels=["USA", "Canada"])
```



This will set the height of every bar equal to the average values in the `data_science` column for each value of the `categorical` variable (0 and 1 here). `barplot` has an optional argument `estimator` that can control the height of each bar. Try `estimator = sum`, `estimator = len`, or `estimator = median`.

`sns.boxplot`

A `boxplot` is a visual representation of five-point summary statistics of a given data set. A five-point summary includes `minimum`, `first quartile (25th percentile)`, `median (second quartile or 50th percentile)`, `third quartile (75th percentile)`, and `maximum`. A `boxplot` shows distributions for categories. According to `seaborn` documentation:

A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers" using a method that is a function of the interquartile (IQ) range.

Technical note: A dataset's Inter-quartile (IQ) is defined as $IQ = Q3 - Q1$ where `Q3` is the 3rd quartile, and `Q1` is the 1st quartile. According to the Inter-Quartile method, any observation outside $[Q1 - 3 * IQ, Q3 + 3 * IQ]$ is considered an `outlier`.

Here is an example:

Example

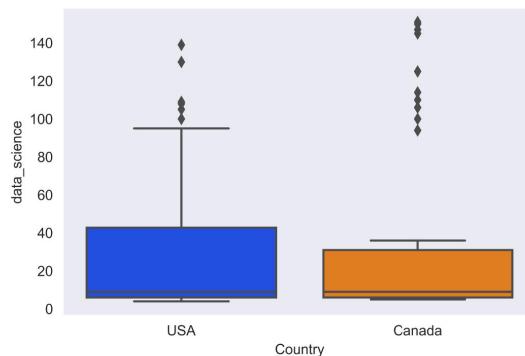
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.boxplot(data=df, x="categorical", y="data_science", palette="bright")

axes.set(xlabel="Country", xticklabels=["USA", "Canada"])
```



- `boxplot` has an optional argument of `orient`. Try `orient = 'h'`

`sns.violinplot`

A `violinplot` combines a `boxplot` and a `distplot`. A `boxplot` or a `violinplot` are created for `categorical-continuous` variables (if the `x-axis` is `categorical` and the `y-axis` is `continuous`). Here is an example

Example

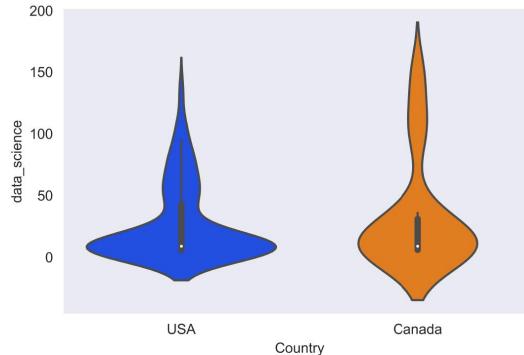
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.violinplot(
    data=df,
    x="categorical",
    y="data_science",
    palette="bright",
)
```

```
    axes.set(xlabel="Country", xticklabels=["USA", "Canada"])
```



sns.histplot

Use `sns.histplot` to visualize the distribution of one or more variables. Here are a few examples

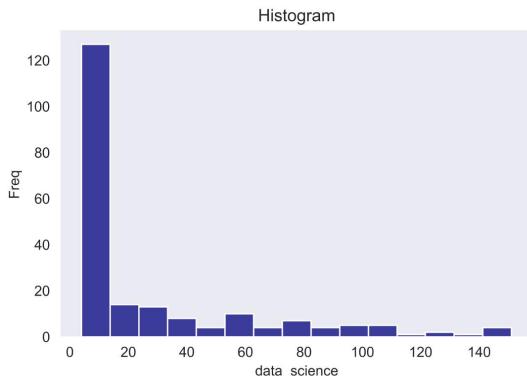
Example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.histplot(data=df, x="data_science", color="navy", bins=15, palette="bright")
axes.set(ylabel="Freq", title="Histogram")
```



- One acceptable rule of thumb for the number of bins is $\sqrt{\text{number of rows}}$
- Instead of `bins`, one can pass the `binwidth` value.
- A histogram with horizontal bars can be achieved simply by determining the value of `y` instead of `x`.
- a kernel density estimate can be added by passing `kde = True` to the `sns.histplot()` method. This provides (smoothed) information about the shape of the distribution. More about kernel density estimate at https://en.wikipedia.org/wiki/Kernel_density_estimation
- `hue` parameter can be used for mapping values to a categorical variable. Try `hue = 'categorical'`
- If a stacked histogram is desired, use `multiple = 'stack'`
- If neither `x` nor `y` is assigned, a histogram is drawn for each numeric column in `data`.
- When both `x` and `y` are assigned, a bi-variate histogram is computed and shown as a `heatmap`.

`sns.distplot`

A closely related plot is `sns.distplot` to visualize the distribution of one or more variables.

`sns.jointplot`

This method draws a plot of two variables with bi-variate (scatter plot) and uni-variate (histogram) graphs. A Joint-Plot takes two variables and creates Histogram and Scatter plot together.

A scatter plot is used for the visual inspection of correlation, and the histogram is used for inspecting the approximate shape of data distribution.

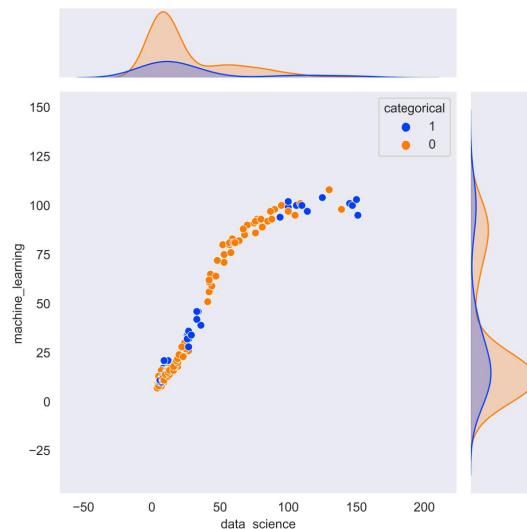
Example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.jointplot(
    data=df,
    x="data_science",
    y="machine_learning",
    hue="categorical",      # categorical variable for mapping color of plot elements
    hue_order=[1, 0],        # order of plotting for categorical levels of the hue
    color="navy",
    palette="bright",
)
```



Note: `jointplot()` has an optional argument of `kind` with a default value of `scatter`. Try: `kind = reg`, `kind = resid`, `kind = kde` and `kind = hist`. How do you think you can use `kind=resid`?

`sns.pairplot`

A `pairplot` visualizes pairwise relationships among numerical values in a data set.

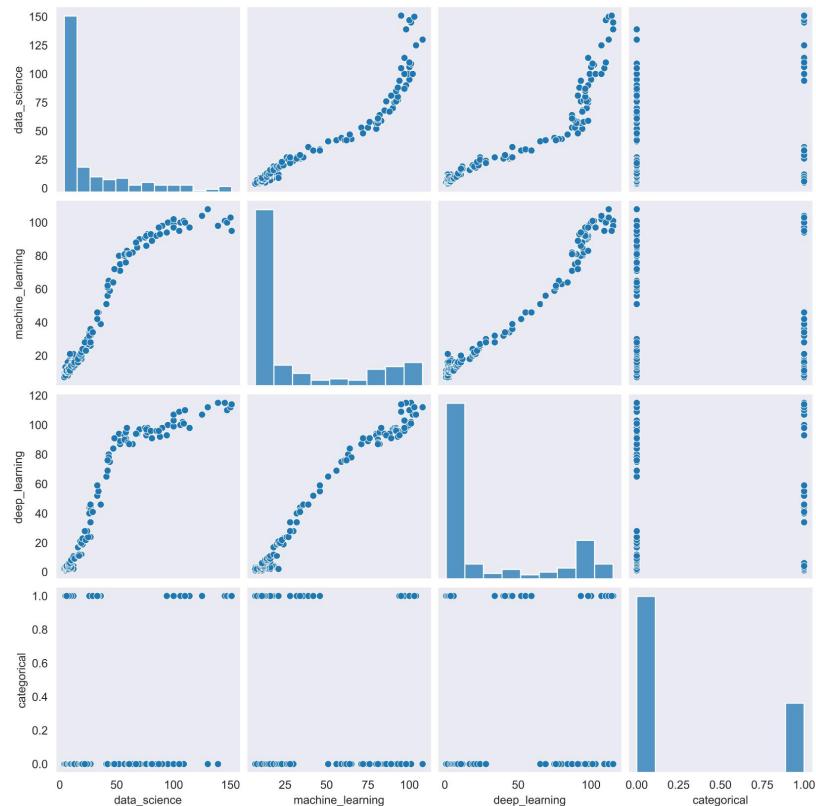
Example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.pairplot(data = df)
```



- The diagonal part shows the `distplot` or `histogram` with kernel density estimation. The upper and lower part of the `Pairplot` shows the `scatterplot`.
- The `hue` parameter can be used to color the plot using categorical columns.
- The `kind` parameter is similar to `jointplot`.

Try `axes = sns.pairplot(data=df, kind="reg", hue="categorical")`

sns.lmplot

You are familiar with `lmplot` from the linear regression part of the course. `lmplot` is a plot that fits the ordinary least square regression line to the data set showing as `scatterplot(s)`.

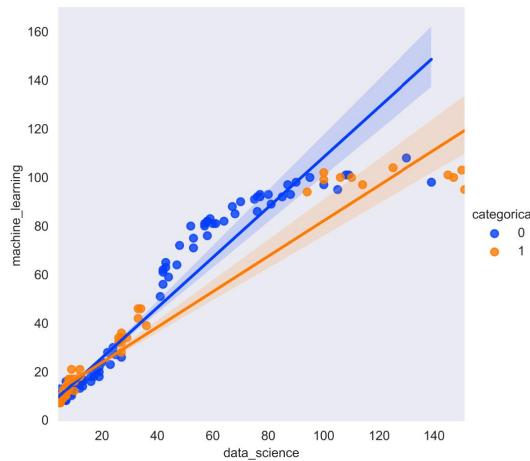
Example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.lmplot(data = df, x = 'data_science', y = 'machine_learning', hue = 'categorical',
palette = 'bright')
```

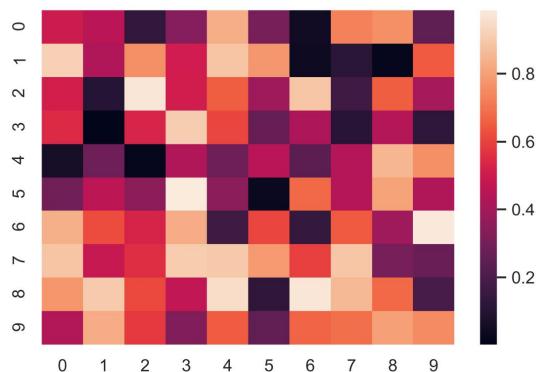


sns.heatmap

heatmap One of the most popular plots provided by `seaborn`. Let's look at several cool `heatmaps` taken from the `seaborn` website.

Uniform heatmap example

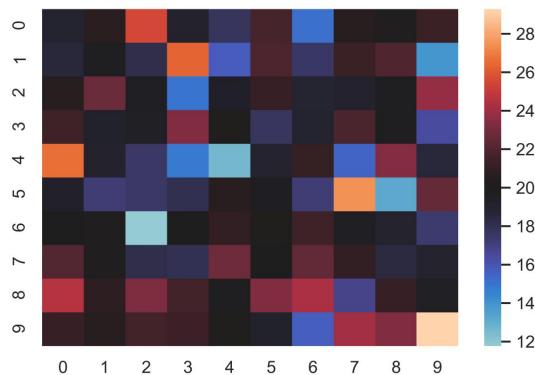
```
import numpy as np
import seaborn as sns
sns.set_theme()
uniform_data = np.random.random(size = (10, 10))
ax = sns.heatmap(uniform_data)
```



Normal heatmap example

```
import numpy as np
import seaborn as sns

sns.set_theme()
normal_data = np.random.normal(loc=20, scale=3, size=(10, 10))
ax = sns.heatmap(normal_data, center=20)
```



Besides making cool-looking plots(!), in data analytics, it is widespread to use `heatmap` to show all the correlations between variables in a dataset. Here is an example

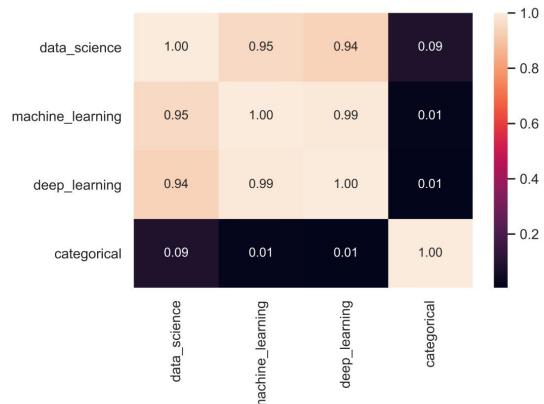
heatmap for visualizing correlation example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("clean_search_history.csv")

sns.set_style("dark")

axes = sns.heatmap(df.corr(), annot=True, fmt=".2f")
```



saving a sns plot

Similar to `matplotlib`, `savefig()` method could be used to save a `seaborn` plot. Here is the most common syntax used for `savefig`:

```
plt.savefig("file_name.extension", dpi = , facecolor = , edgecolor = , bbox_inches='tight', pad_inches = 0.5)
```

References

This document is an adaption from the following references:

- 1- *Python online documentation*; available at <https://docs.python.org/3/>
- 2- *Exploratory Data Analysis with Python*; Suresh Kumar Mukhiya and Usman Ahmed, Packt publications, 2020
- 3- *Introduction to programming in Python*; available at <https://introcs.cs.princeton.edu/Python/home>
- 4- *Introduction to Computation and Programming Using Python: With Application to Understanding Data*, John Guttag, The MIT Press, 2nd edition, 2016
- 5- *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019

6- *Data Mining for Business Analytics*, Galit Shmueli, Peter C. Bruce, Peter Gedeck, Nitin R. Patel, Wiley, 2020

7- *Interactive Data Visualization with Python*; Abha Belorkar, Sharath Chandra Guntuka, Shubhangi Hora, and Anshu Kumar. Packt publications, 2020

8- *Simulation Modeling with Python*; Giuseppe Ciaburro, Packt publication, 2020

9- <https://betterprogramming.pub/>

10- <https://www.learnpython.org/>

11- <https://seaborn.pydata.org/>

12- <https://pyviz.org/index.html>

13- <https://numpy.org/>

14- <https://pandas.pydata.org/>

15- *Pandas Cookbook*, Matt Harrison and Theodore Petrou, Packt publications, 2020