Data Visualizations Using Matplotlib and Seaborn

Making informative visualizations (sometimes called plots) is one of the most important tasks in the field of data science. It may be a part of the exploratory process—for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. Python has many add-on libraries for making static or dynamic visualizations. Two popular libraries in Python for visualization are Matplotlib and Seaborn.

Matplotlib

Matplotlib is a desktop plotting package designed for creating plots and figures suitable for publication. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib.

Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

import matplotlib as mpl

import matplotlib.pyplot as plt

A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context. The best use of Matplotlib differs depending on how you are using it. The three main contexts for using Matplotlib are **in a script**, **an IPython terminal**, or **a Jupyter notebook**.

Plotting from a script

If you are using Matplotlib from within a script, the function *plt.show()* is your friend. plt.show() starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

One thing to be aware of: the plt.show() command should be used only once per Python session, and is most often seen at the very end of the script. Multiple show() commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Plotting from an IPython shell

To enable this mode, you can use the *%matplotlib* magic command after starting ipython. Some changes (such as modifying properties of lines that are already drawn) will not draw automatically; to force an update, use *plt.draw()*. Using plt.show() in Matplotlib mode is not required.

Plotting from an IPython notebook

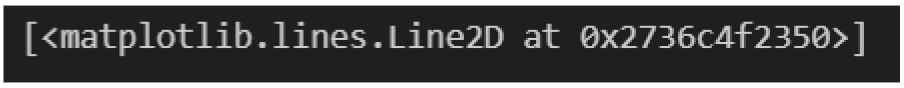
The IPython notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document.

• *Using %matplotlib* notebook will lead to interactive plots embedded within the notebook

• Using *%matplotlib inline* will lead to static images of your plot embedded in the notebook

Note:

You may notice output like figure shown below while running the Matplotlib library. Matplotlib returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output, or you can put a **semicolon** at the end of the line to suppress the output.



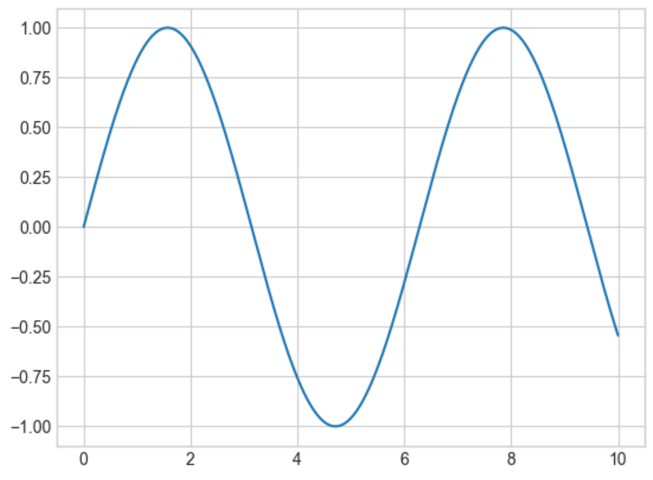
```
[<matplotlib.lines.Line2D at 0x2736c4f2350>]
```

Figures and Axes

For all Matplotlib plots, we start by creating a figure and an axes. In Matplotlib, the figure (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The axes (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization. Once we have created an axes, we can use the ax.plot function to plot some data.

Code and output:

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
plt.plot(x,np.sin(x));
```



Saving Figures to File

Matplotlib allows you to save figures in various formats like PNG, PDF, etc. You can save a figure using the *savefig()* command. For example, to save the previous figure as a PNG file, you can run this:

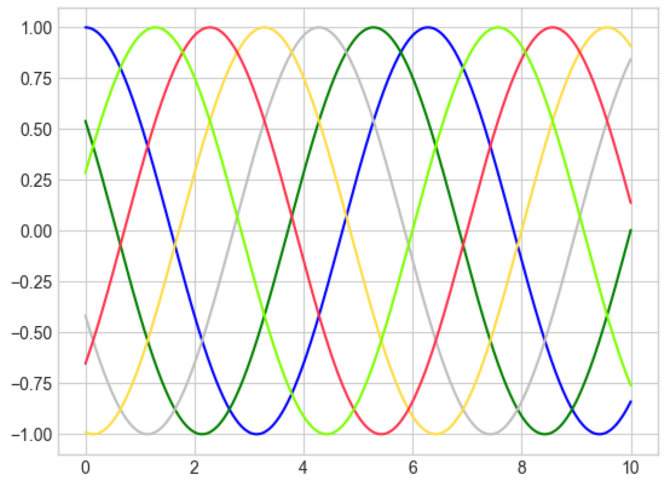
Code: *fig.savefig('my_figure.png')*

Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot() function takes additional arguments that can be used to specify these. To adjust the color, you can use the color keyword, which accepts a string argument representing virtually any imaginable color.

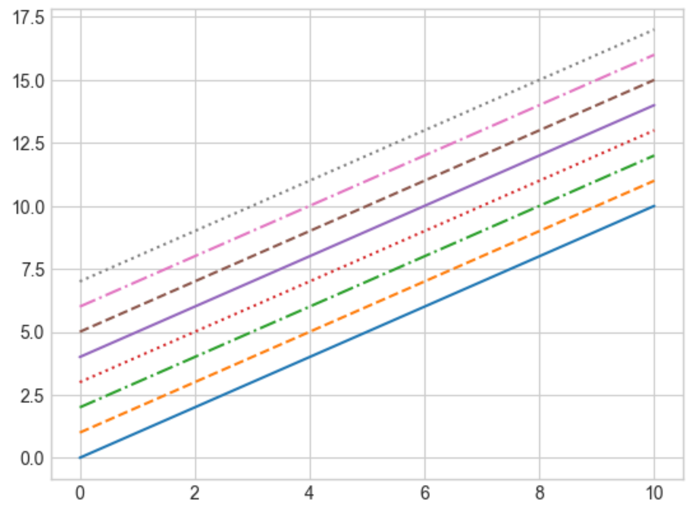
Code and output:

```
plt.plot(x,np.cos(x), color='blue')   # specify color by name
plt.plot(x,np.cos(x + 1), color='g')    # short color code (rgbcmyk)-
Red,Green,Blue,Cyan,Magenta,Yellow,black
plt.plot(x,np.cos(x + 2), color='0.75')    # Grayscale between 0 and 1
plt.plot(x,np.cos(x + 3), color='#FFDD44')   # Hex code (RRGGBB from 00 to FF)
plt.plot(x,np.cos(x + 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
plt.plot(x,np.cos(x + 5), color='chartreuse') # all HTML color names supported
```



If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines. Similarly, you can adjust the line style using the linestyle keyword

```
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```



If you would like to be extremely terse, these line styles and color codes can be combined into a single non-keyword argument to the plt.plot() function like -g for solid green,--c for dashed cyan.

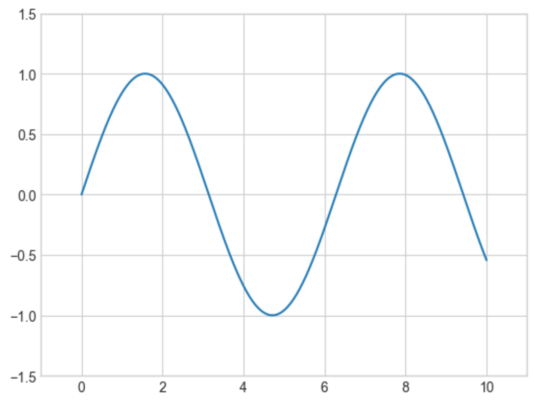
Adjusting the Plot:

Matplotlib does a good job of selecting default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the *plt.xlim()* and *plt.ylim()* methods.

A useful related method is *plt.axis()* (note here the potential confusion between axes with an e, and axis with an i). The plt.axis() method allows you to set the x and y limits with a single call, by passing a list that specifies *[xmin, xmax, ymin, ymax].*

Code and output:

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```



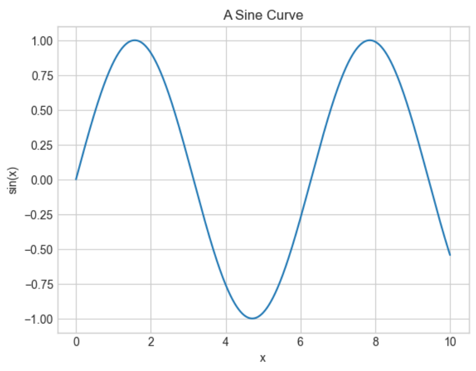
The plt.axis() method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot, it allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y by passing values like tight, equal.

Labelling Plots

Titles and axis labels are essential to understand the plot; Matplotlib provides methods to set them easily.

Code and output:

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)")
```



Customizing Plot Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. The simplest legend can be created with the plt.legend() command, which automatically creates a legend for any labeled plot elements.

There are many ways we might want to customize such a legend. Some of them are

- we can specify the location and turn off the frame by *loc='upper left', frameon=False*
- We can use the ncol command to specify the number of columns in the legend. That can be achieved by the parameter *ncol.*
- We can use a rounded box (fancybox) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text by *fancybox=True, framealpha=1, shadow=True, borderpad=1*

Simple Scatter Plots:

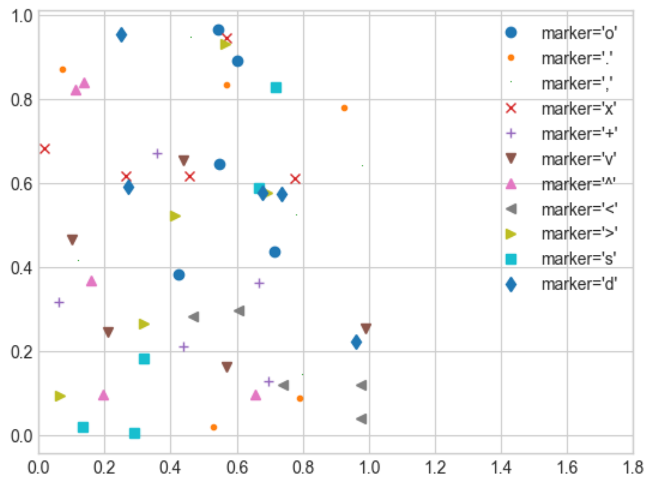
Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

Scatter Plots with plt.plot

The plt.plot() function can also create scatter plots by modifying the third argument. The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-' and '--' to control the line style, the marker style has its own set of short string codes.

Code and output:

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
    label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

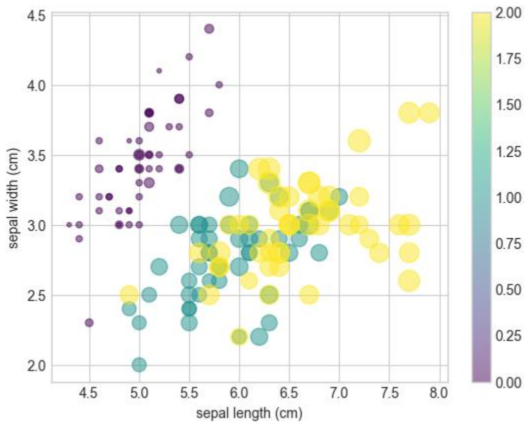


Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the *plt.scatter()*, which can be used very similarly to the plt.plot function. The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Code and output:

```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T
plt.scatter(features[0],features[1],alpha=0.5,s=features[3]*100,c=iris.target,cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
plt.colorbar();
```



plot Versus scatter: A Note on Efficiency

Aside from the different features available in plt.plot and plt.scatter, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, plt.plot can be noticeably more efficient than plt.scatter. The reason is that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In plt.plot, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: *plt.contour()* for contour plots, *plt.contourf()* for filled contour plots, and *plt.imshow()* for showing images.

A contour plot A contour plot can be created with the plt.contour function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays.

plt.imshow() plt.imshow() function interprets a two-dimensional grid of data as an image. plt.imshow() doesn’t accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin, ymax] of the image on the plot.plt.imshow() by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data. plt.imshow() will automatically adjust the axis aspect ratio to match the input data; you can change this by setting, for example, plt.axis(aspect='image') to make x and y units match.

Multiple Subplots

Sometimes it is helpful to compare different views of data side by side. To this end, Matplotlib has the concept of subplots: groups of smaller axes that can exist together within a single figure. These subplots might be insets, grids of plots, or other more complicated layouts.

plt.axes: Subplots by Hand

plt.axes() takes an optional argument that is a list of four numbers in the figure coordinate system. These numbers represent [bottom, left, width, height] in the figure coordinate system, which ranges from 0 at the bottom left of the figure to 1 at the top right of the figure.

plt.subplot: Simple Grids of Subplots

Aligned columns or rows of subplots are a common enough need that Matplotlib has several convenience routines that make them easy to create. The lowest level of these is plt.subplot(), which creates a single subplot within a grid. As you can see, this command takes three integer arguments—the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right. The command plt.subplots_adjust can be used to adjust the spacing between plots.

Colorbars

In Matplotlib, a colorbar is a separate axis that can provide a key for the meaning of colors in a plot. By default, when a single color is used, negative values are represented by dashed lines, and positive values by solid lines. Alternatively, you can color-code the lines by specifying a colormap with the cmap argument. plt.colorbar() command, which automatically creates an additional axis with labeled color information for the plot.

Choosing the colormap

Sequential colormaps These consist of one continuous sequence of colors (e.g., binary or viridis).

Divergent colormaps These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., RdBu or PuOr).

Qualitative colormaps These mix colors with no particular sequence (e.g., rainbow or jet). Qualitative maps usually do not display any uniform progression in brightness as the scale increases.so qualitative maps are often a poor choice for representing quantitative data.

Customizing Ticks

Matplotlib’s default tick locators and formatters are designed to be generally sufficient in many common situations. But adjusting the tick locations and formatting for the particular plot type are also available in Matplotlib.

Major and Minor Ticks

Within each axis, there is the concept of a major tick mark and a minor tick mark. As the names would imply, major ticks are usually bigger or more pronounced, while minor ticks are usually smaller. By default, Matplotlib rarely makes use of minor ticks, but one place you can see them is within logarithmic plots.

Hiding Ticks or Labels

Perhaps the most common tick/label formatting operation is the act of hiding ticks or labels. We can do this using *plt.NullLocator()* and *plt.NullFormatter().*

Reducing or Increasing the Number of Ticks

One common problem with the default settings is that smaller subplots can end up with crowded labels. Particularly for the x ticks, the numbers nearly overlap, making them quite difficult to decipher. We can fix this with the *plt.MaxNLocator()*, which allows us to specify the maximum number of ticks that will be displayed

Setting Styles

We will use the *plt.style* directive to choose appropriate aesthetic styles for our figures.

Various built-in styles

FiveThirtyEight style: The FiveThirtyEight style mimics the graphics found on the popular FiveThirtyEight website. *plt.style.context('fivethirtyeight')*

Ggplot: The ggplot package in the R language is a very popular visualization tool. Matplot lib’s ggplot style mimics the default styles from that package *plt.style.context('ggplot')*

Dark background: For figures used within presentations, it is often useful to have a dark rather than light background. *plt.style.context('dark_background')*

Grayscale: Sometimes you might find yourself preparing figures for a print publication that does not accept color figures. For this, the grayscale style can be very useful. *plt.style.context('grayscale')*

Seaborn style: Matplotlib also has stylesheets inspired by the Seaborn library. These styles are loaded automatically when Seaborn is imported into a notebook.

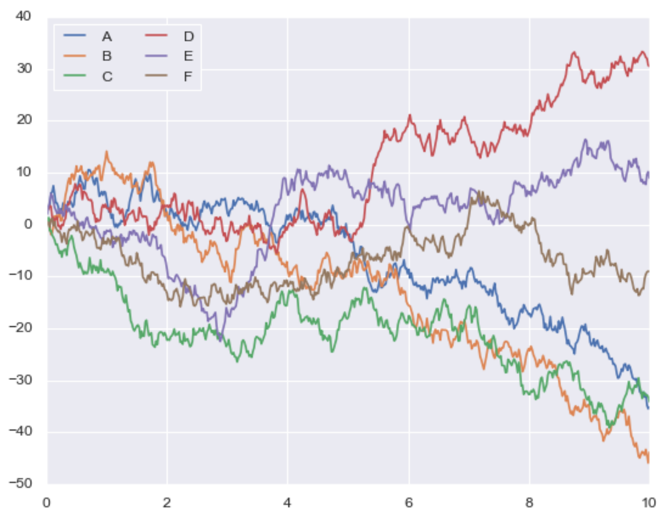
Seaborn

Seaborn provides an API on top of Matplot lib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames. Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib’s default parameters and in turn get even simple

Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's set_theme() method.

Code and output:

```
import seaborn as sns
sns.set_theme()
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```



Classification of seaborn

When working with Seaborn, it's important to understand the two main types of plot functions: figure-level and axes-level functions.

1. Axes-Level Functions:

Axes-level functions create individual plots and provide fine-grained control over the plot's elements, such as labels, titles, etc. These functions return Matplotlib Axes objects, which allow you to further customize your plot.

Characteristics of Axes-Level Functions:

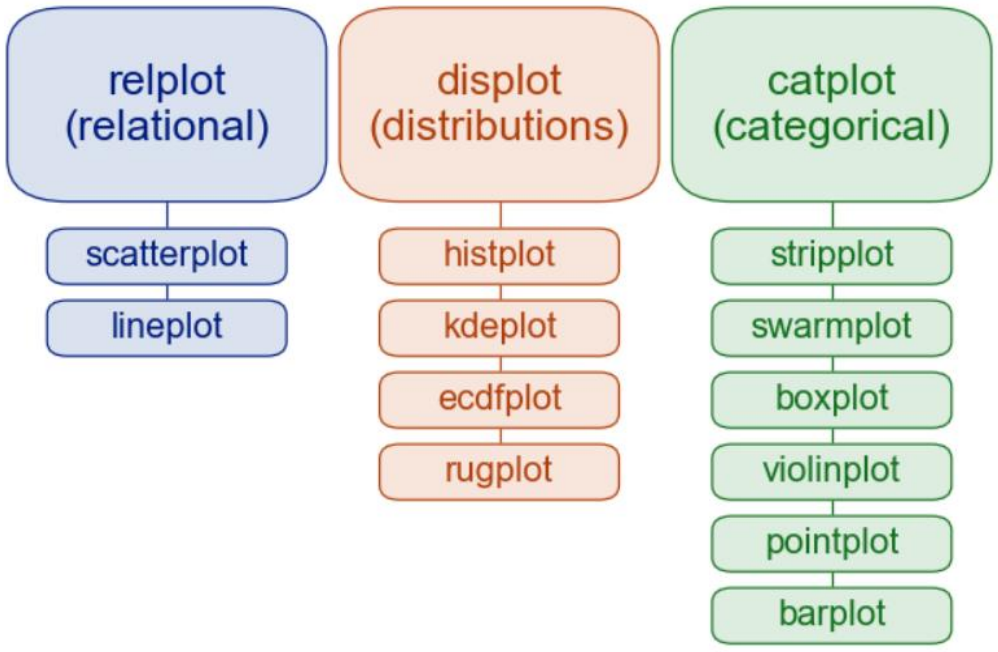
- They plot data on one Axes object.
- You can use them in subplots or customize them within larger figures.
- You have fine control over individual plot elements using Matplotlib commands.
- Examples include: sns.scatterplot(), sns.lineplot(), sns.histplot(), sns.boxplot(), sns.barplot(), etc.

2.Figure-Level Functions:

Figure-level functions are more comprehensive and manage the creation of entire plots or grids of plots. They handle higher-level tasks, such as the creation of the figure (canvas) and axes, arranging multiple plots, and providing additional options for facet plots (splitting data into subsets).

- Automatically create a figure and multiple subplots if needed.

- Use a higher-level interface for complex plots (e.g., faceting, multi-plot layouts).
- Return a FacetGrid or PairGrid object, which allows for multiple subplots or conditional data plotting.
- Handle the overall layout and legends by default, so you don't need to manage these aspects.
- Examples include: sns.relplot(), sns.catplot(), sns.lmplot(), sns.pairplot(), and sns.jointplot().



Key Differences Between Figure-Level and Axes-Level Functions:

Feature	**Axes-Level Functions**	**Figure-Level Functions**
Scope	Single plot on one Axes	Can create multiple subplots, manage layout
Returned Object	Matplotlib Axes object	Seaborn FacetGrid or PairGrid object
Control	Fine-grained control over plot elements	Higher-level control over the overall figure layout
Example Functions	sns.scatterplot(), sns.histplot()	sns.relplot(), sns.catplot(), sns.lmplot()
Faceting/Conditioning	No automatic faceting	Automatically handles faceting, subplots, and legends
Customization	Manual with Matplotlib functions	High-level options with Seaborn interface

Explanation of concepts along with project:

Features in dataset:

- **age:** age of the participants.
- **gender:** men or women.
- **final:** finishing time for the full marathon in hours
- **split:** the first half-marathon time in hours
- **final_sec**: the **total time taken** to complete the marathon (in seconds).
- **split_sec**: the first half-marathon (in seconds)
- **split_frac**: the **split fraction** of every participant, how much faster/slower the second half of the race was compared to the first half. A negative value means a runner had a faster second half (negative split), and a positive value means a slower second half.
- **age_dec**: **age decade** groups of every participant.

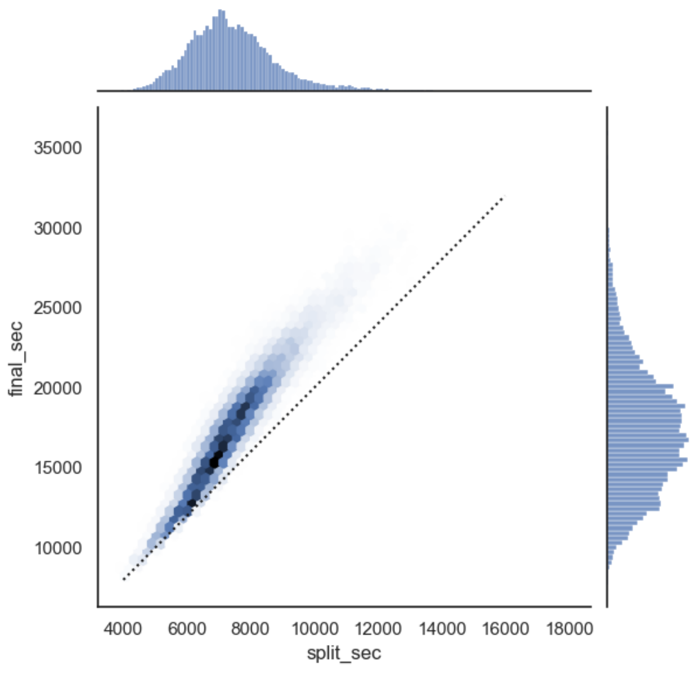
Note: Refer the marathon.ipynb before proceed.

Jointplot

Seaborn's jointplot function is a powerful tool for visualizing the relationship between two variables along with their individual distributions. It provides a convenient interface to the JointGrid class, allowing you to create various types of plots such as scatter plots, KDE plots, histograms, hex plots, regression plots, and residual plots.

Code and output:

```
with sns.axes_style('white'):
        g = sns.jointplot(marathon,x="split_sec", y="final_sec", kind='hex')
        g.ax_joint.plot(np.linspace(4000, 16000),np.linspace(8000, 32000), ':k')
```



g.ax_joint.plot(np.linspace(4000, 16000),np.linspace(8000, 32000), ':k') adds a reference line to the joint plot in Seaborn.

Explanation of Each Component:

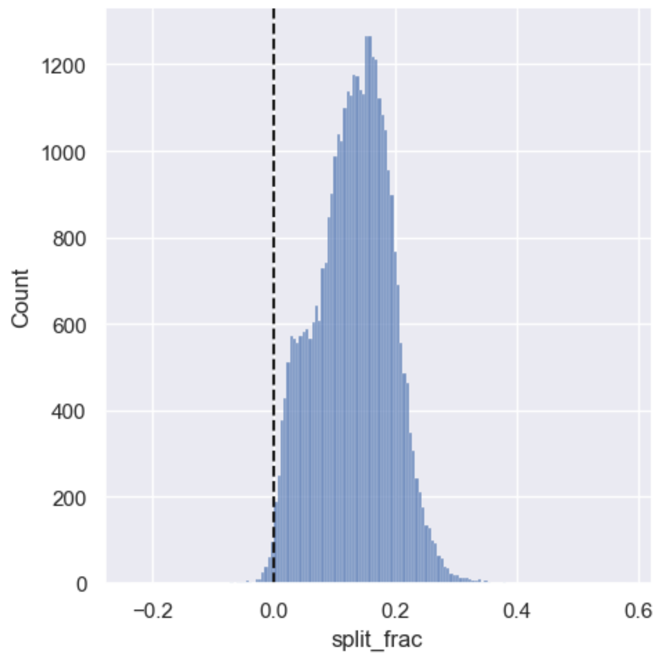
1. **g.ax_joint.plot()**:
   - g is a joint plot object created by Seaborn's sns.jointplot()
   - ax_joint is an attribute of the joint plot, referring to the central (main) Axes where the joint distribution is drawn.
   - plot() is a Matplotlib function to draw lines or points on this Axes object.
2. **np.linspace(4000, 16000)**:
   - np.linspace() generates evenly spaced numbers over a specified interval.
   - np.linspace(4000, 16000) creates a sequence of numbers starting from 4000 to 16000. This is likely the x-coordinates for the line.
3. **np.linspace(8000, 32000)**: Similarly, this creates a sequence of numbers from 8000 to 32000. This is likely the y-coordinates for the line.
4. **':k'**: ':k' means the line will be **dotted** and **black**.

Displot

The displot function in Seaborn is a versatile tool for creating distribution plots. It provides a figure-level interface for drawing distribution plots onto a FacetGrid, allowing for the visualization of univariate or bivariate distributions of data. The function supports several types of plots, including histograms, kernel density estimates (KDE)

Code and output:

```
sns.displot(marathon['split_frac'])
plt.axvline(0, color="k", linestyle="--");
```



The *axvline()* in Matplotlib is used to add vertical lines across the axes of a plot. This can be useful for highlighting specific x-values or for visualizing thresholds.

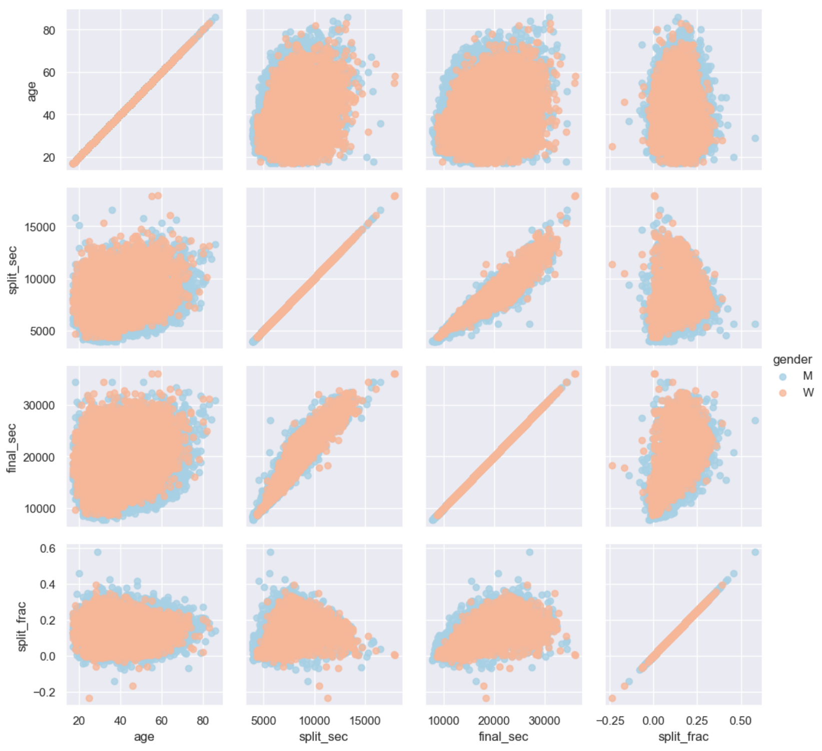
PairGrid

PairGrid is a powerful and flexible class provided by Seaborn, a statistical data visualization library in Python. It's designed to create a grid of Axes such that each variable in the data will be shared across the y-axes across a single row and the x-axes across a single column. The primary use of **PairGrid** is to visualize pairwise relationships in a dataset, allowing for the inspection of interactions between multiple variables. To initialize a **PairGrid**, you need to pass your data as a DataFrame to the constructor, along with any other aesthetic parameters you wish to define, such as hue, palette, or vars.

Code and output:

```
pg = sns.PairGrid(marathon, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
 hue='gender', palette='RdBu_r')
pg.map(plt.scatter, alpha=0.8)    # Map the plots to the grid and add a legend
pg.add_legend();
```

- *hue* - allows you to add an additional dimension to your plots by coloring points according to a categorical variable.
- *vars* - specifying different variables for the rows and columns.
- *palette*-Set of colors for mapping the “hue“ variable.



KDE Plot

A **Kernel Density Estimate (KDE) plot** is a method for visualizing the distribution of observations in a dataset, similar to a histogram. It represents the data using a continuous probability density curve in one or more dimensions. KDE plots are particularly useful for visualizing the probability density function of continuous or non-parametric data.

Customizing the KDE Plot

Seaborn's kdeplot function offers various parameters to customize the plot:

- shade: Fill the area under the KDE curve.
- color: Change the color of the KDE curve.
- bw_adjust: Adjust the bandwidth of the KDE. Increasing this value will make the curve smoother.
- hue: Add a semantic variable to map the color of plot elements.

KDE plots are useful for:

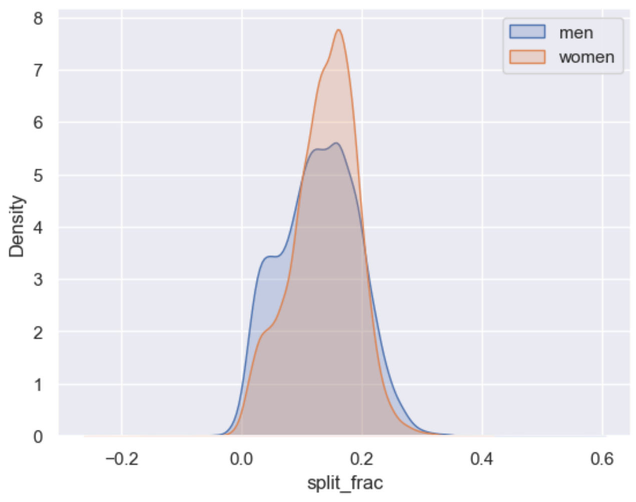
- **Visualizing the distribution** of a single variable or the joint distribution of two variables.
- **Comparing distributions** across different categories using the hue parameter.
- **Smoothing data** to reveal underlying patterns without assuming a specific parametric form.

Important Considerations

- **Bandwidth Selection**: The bandwidth, or standard deviation of the smoothing kernel, is crucial. An over-smoothed curve can erase true features, while an under-smoothed curve can create false features.
- **Boundary Issues**: The KDE curve can extend to values that do not make sense for the dataset. Use the cut and clip parameters to control the extent of the curve.

Code and output:

```
sns.kdeplot(marathon.split_frac[marathon.gender=='M'], label='men', fill=True)
sns.kdeplot(marathon.split_frac[marathon.gender=='W'], label='women', fill=True)
plt.xlabel('split_frac')
plt.legend();
```



Violin Plot

A **violin plot** is a combination of a **box plot** and a **KDE plot (Kernel Density Estimate plot)**. It provides a visual representation of the distribution of a dataset across different categories, showing the probability density of the data at different values.

Here's a breakdown of how a violin plot works:

1. Kernel Density Estimate (KDE) - The "Violin" Shape:

- The main part of the violin plot (the "violin" shape) is essentially a **smoothed histogram** of the data.

- **KDE** estimates the probability density of the data (how likely different values are to occur). It smooths out the data into a continuous curve, rather than showing the count of values in discrete bins (as in a histogram).
- The **width of the violin** at a particular value indicates the **density of data points** at that value. A wider section means more data points are concentrated at that value, while a narrower section means fewer points.
- **Symmetry**: The KDE is usually plotted symmetrically, meaning the same distribution is shown on both sides of the central line, which is why it looks like a violin.

2. Middle Line and Box Plot Features (Optional):

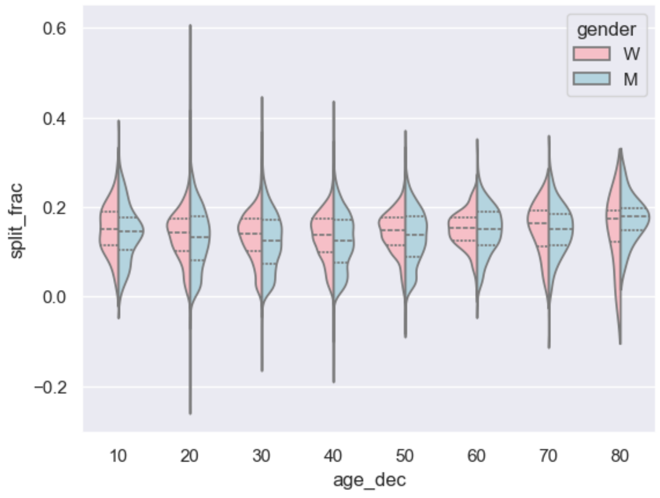
- Violin plots can also include features of a **box plot** in the middle:
    - **White dot or line**: Represents the **median** (middle value of the dataset).
    - **Thick bar** (dashed or solid line): Shows the **interquartile range (IQR)**, which represents the middle 50% of the data.
    - **Thin line** (whiskers): Represents the **range of the data** excluding outliers.

3. Comparing Multiple Categories:

- **Multiple violins** are often shown side by side to compare distributions across different categories (like gender, age, etc.).
- This allows you to easily compare where data points are concentrated for different groups.

Code and output:

```
with sns.axes_style(style=None):
        sns.violinplot(x="age_dec", y="split_frac", hue="gender", data=marathon,
        split=True, inner="quartile", palette=["lightpink","lightblue"]);
```



The **split=True** parameter has been used, which allows you to compare men and women directly in the same violin for each age group, with their distributions being shown side by side in each violin shape.

Interpreting the Shapes:

Each "violin" shape gives you a sense of the distribution of split_frac for men and women in a particular age group. Here's what to look at:

1. **Width of the Violin**:

   - **Wider sections** indicate where more data points are concentrated (a higher density of values).
   - **Narrower sections** show where fewer data points exist (lower density).

For instance, in the age group 20s, the distribution for both men and women are wide around a split fraction of 0.2, indicating that many runners in their 20s had similar split times (relative to the total race time).

2. **Comparing Men (Blue) vs. Women (Pink)**:
   - In most age groups, **men's split_frac distribution (blue)** is slightly shifted downwards compared to women (pink). This suggests that men, on average, had a lower split fraction, which could indicate they ran the first half of the race faster relative to the second half, compared to women.
   - For the 80-year-olds, the **women's distribution** is concentrated much lower than for men. This means that women in this age group have significantly lower split fractions (better pacing or faster initial splits) than men.
3. **Symmetry**:
   - The violin is symmetric, meaning the same density is shown on both sides for each gender (this is how violin plots are usually presented).

Conclusion from the Plot:

- **Men vs. Women (Split Fraction)**: Men tend to have lower split fractions across most age groups, suggesting that they run the first half of the race faster compared to women.
- **Age-Related Trends**:
  - The **split fraction for men** remains fairly consistent across most age decades (20s to 60s).
  - Women's split fractions also remain fairly consistent, but the difference between men and women is more noticeable in certain age groups.
  - Interestingly, **women in the 80s age group** seem to have a better (lower) split fraction compared to men in the same age group, though this may be due to the smaller number of data points in that group.

Box Plot

In a violin plot in the above figure, there are **dashed lines** and **small box shapes** inside the violin. These represent the components of a **box plot** that are integrated into the violin plot. Here's a breakdown of the elements of a box plot:

Key Elements of a Box Plot:

1. **Median (Middle Dashed Line)**:
   - The median is the value that separates the lower 50% of the data from the upper 50%.
   - In the violin plot, this is shown as a **horizontal dashed line** in the middle of each violin, representing the **median split fraction** for each gender and age group.
2. **Interquartile Range (IQR) - The Box**:
   - The IQR is the range where the middle 50% of the data lies. It is the difference between the **75th percentile (Q3)** and the **25th percentile (Q1)**.
   - In the violin plot, the IQR is represented by the **box-like region** (dashed or solid lines) inside the violin.
     - The top of the box represents the **75th percentile** (Q3) – 75% of the data lies below this value.
     - The bottom of the box represents the **25th percentile** (Q1) – 25% of the data lies below this value.
   - The range of this box gives you an idea of how spread out the middle 50% of the values are.

3. **Whiskers (Upper and Lower Ends)**:
    - Whiskers extend from the box to show the range of the data. Typically, whiskers go up to **1.5 times the IQR** from the quartiles, but they can also show the minimum and maximum values that aren't considered outliers.
    - These whiskers might not be clearly visible in every violin plot, but they help indicate the spread of the data beyond the middle 50%.
4. **Outliers**:
    - Data points that fall outside the whiskers are considered outliers. Sometimes, these are represented as individual points outside the main distribution.
    - In your violin plot, outliers might not be explicitly shown, but if the violin extends thinly far from the center, it suggests the presence of extreme values.

How the Box Plot Elements Are Used in Your Violin Plot:
In the **violin plot** in the above figure, the box plot is overlaid inside the violin:

- The **middle dashed line** in each color segment represents the **median split fraction** for men and women.
- The **shaded area around the median** represents the middle 50% of the data, or the **IQR**. This shows you where most of the values for split fraction are concentrated.
- The **ends of the violins** stretch out to show the **range of the data**, including the less common split fractions at the extremes of the distribution.

By combining the box plot with the violin plot, you get both the **summary statistics** (median, quartiles, and range) and a **detailed view of the distribution shape** (density) for each gender and age group.
Example from the Plot:

- Take the age group of **20-30**:
    - The **dashed line in the pink violin** (for women) shows the median split fraction for women in that age group.
    - The **box-like area** around the dashed line shows where the middle 50% of the women's data lies (between the 25th and 75th percentiles).
    - Similarly, the **blue violin** (for men) shows the median split fraction for men in that age group, along with the middle 50% range.

The **box plot helps you quickly understand where the majority of the data is concentrated** (middle 50% and the median) while the violin plot shows you the overall distribution.

Summary:

- The **wider the plot** at a given point on the y-axis, the more runners had that split fraction.
- You can see how **men and women's distributions differ** in terms of their split fractions for each age group, with men generally having lower split fractions.
- **Women in the 80s age group** have a lower split fraction than men, suggesting a better split time, though this could be due to smaller sample sizes.

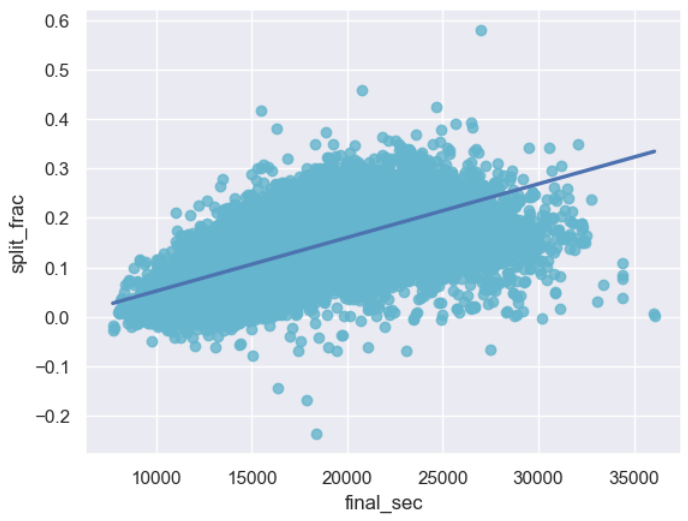
This plot gives a clear visual comparison of the pacing between men and women across different age groups, highlighting both similarities and differences in race performance based on gender and age.

Regplot

Seaborn's regplot function is a powerful tool for visualizing the relationship between two variables along with a linear regression model fit. It is part of the Seaborn library, which is built on top of Matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics**.**

Code and output:

```
g = sns.regplot(x='final_sec',y='split_frac', data=marathon,scatter_kws=dict(color='c'))
```

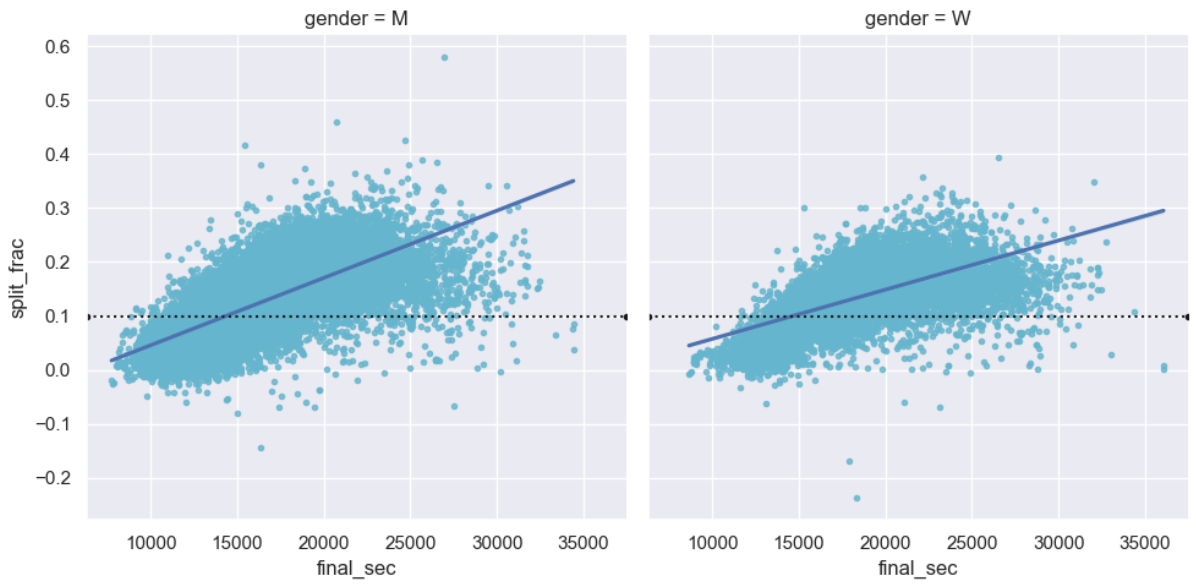


lmplot

Seaborn's lmplot is a powerful tool for visualizing data and fitting regression models across conditional subsets of a dataset. It combines the functionalities of regplot and FacetGrid, making it convenient to fit regression models and visualize them across different facets.

Code and output:

g = sns.lmplot(x='final_sec',y='split_frac', col='gender', data=marathon,markers=".", scatter_kws=dict(color='c'))
g.map(plt.axhline, y=0.1, color="k", ls=":");



The *axhline()* in Matplotlib is used to add horizontal lines across the axes of a plot. This can be useful for highlighting specific y-values or for visualizing thresholds.

Linear Regression Line:

The **blue line** in each plot is the linear regression line showing the trend between split_frac (y-axis) and final_sec (x-axis) for both men and women.

- The **positive slope** indicates that runners with longer finishing times (higher final_sec values) tend to have a higher split fraction, meaning they slow down more in the second half.
- Runners with **negative split fractions** (below the horizontal dotted line) performed better in the second half of the race.

Conclusion:

The key takeaway here is about the correlation between runners who finished quickly and their split fractions:

1. **Fast Finishers (Elite Runners)**:
    - The runners who finished in **less than 15,000 seconds** (~4 hours) are likely to have **negative split fractions** (they run the second half faster than the first).
    - These are likely **elite runners**, as they are capable of maintaining or increasing their pace in the second half of the race.
    - You can observe this in the plot because many of the points around or below **15,000 seconds** have **negative or low split fractions**.
2. **Slower Finishers**:
    - As finishing time increases beyond 15,000 seconds, the **split fraction tends to increase** (more positive values), which means slower runners are more likely to slow down significantly in the second half.
    - The **scatter of points above the dotted line** (positive split fraction) becomes denser for slower runners, indicating that they are more likely to have a slow second half compared to their first half.

Why This Happens:

- **Elite runners** tend to pace themselves better, allowing them to finish strong in the second half of the race, which is why they often have **negative splits**.
- **Non-elite or slower runners** may not be able to maintain their initial pace and tire out in the second half, leading to **positive split fractions**.