Data Visualization with Python

-- A Pandas and Seaborn Approach

ICG Tips & Tricks session

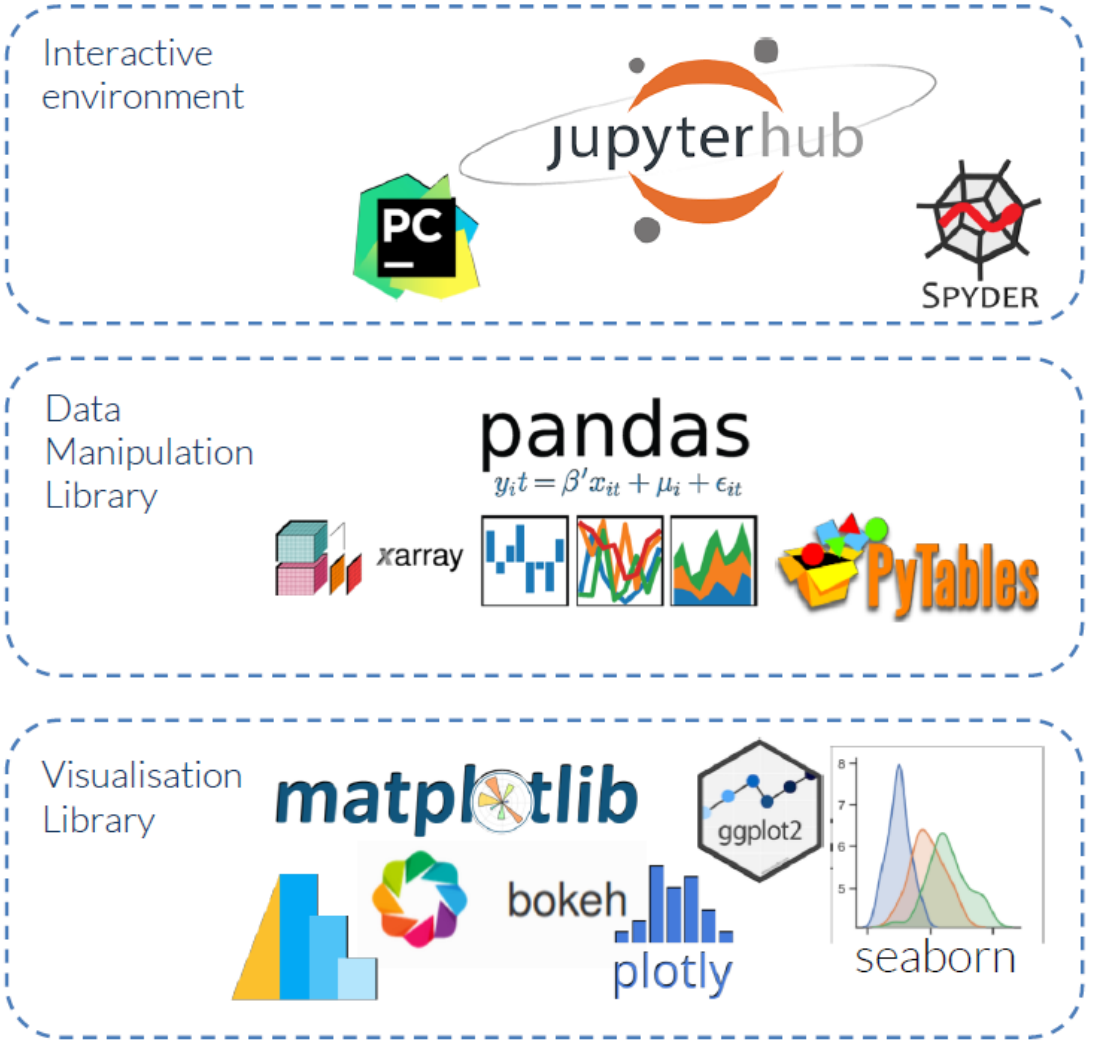
Dandan Zhao

Oct 31, 2024

Data Visualization in Python

Python Visualisation

- Lots of choice of libraries
- Many tools, with varied APIs & outputs
- Best to conquer and become familiar with one / two



Outline

Data Manipulation

1. Exploring Data

   open file, save file, get summary

2. Basic Data Selection

   Filtering columns/rows, Aggregation

Chart Construction

1. Different plots

   scatter plot, box plot, heatmap, …

2. Plot Customization

Introduction to Pandas

Why Pandas?

- Fast and efficient for data manipulation.
- Easily filters, groups, and aggregates data.

Pandas vs. Excel:

- Pandas allows for automation and reproducibility of tasks.
- Works seamlessly with large datasets, unlike Excel, which struggles with size and complexity.
- Ideal for scientific workflows where precise, repeatable data analysis is required

Core structure: Series & DataFrames

- Series: one-dimensional labeled array capable of holding data of any type
- DataFrames: structures with rows and columns for tabular data, capable of handling large, complex datasets.

How to Install Python and Pandas

Install Python

- Go to Python's official website: https://www.python.org/downloads/
- Download the latest version suitable for your operating system.
- Run the installer and make sure to check the box **“Add Python to PATH”**.

Install Pandas

- Once Python is installed, open a command prompt or terminal.
- Type the following command and press Enter to install pandas via pip (Python's package manager):

```
pip install pandas
```

```
import pandas as pd
```

Perun:

- Default python3 doesn’t include pandas

```
[perun4][dzhao][~] python3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license"
>>> import pandas as pd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pandas'
```

- source activate python36-generic

Loading and Saving Data in Pandas

- **Loading Data**:

  # From a CSV file:

  > pd.read_csv('file.csv')

  # From a TSV file:

  > pd.read_csv('file.tsv', sep="\t")

  # From an Excel file

  > pd.read_excel('file.xlsx')

- **Saving Data:**

  # To a TSV file:

  > df.to_csv('file.tsv', sep='\t', index=False)

  # To an Excel file:

  > df.to_excel('file.xlsx', index=False)

```
import pandas as pd

euk = pd.read_csv("eukaryotes.tsv",
                  sep="\t")

euk.to_excel('eukaryotes.xlsx',
             index=False)
```

Example1_Exploring_Data_with_Pandas.py

Exploring Data with Pandas

df.shape()

- Print the shape of the DataFrame
- The number of rows and columns

```
>>> euk = pd.read_csv("eukaryotes.tsv", sep="\t")
>>> print(euk.shape)
(8302, 9)
```

Exploring Data with Pandas

df.shape()

- Print the shape of the DataFrame
- The number of rows and columns

df.info()

- Provides a summary of the DataFrame
- including data types, index, no of rows, data columns, memory storage, and missing values.

```
>>> euk = pd.read_csv("eukaryotes.tsv", sep="\t")
>>> print(euk.shape)
(8302, 9)
```

```
>>> print(euk.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8302 entries, 0 to 8301
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Species             8302 non-null   object
 1   Kingdom             8302 non-null   object
 2   Class               8302 non-null   object
 3   Size (Mb)           8302 non-null   float64
 4   GC%                 8302 non-null   object
 5   Number of genes     8302 non-null   object
 6   Number of proteins  8302 non-null   object
 7   Publication year    8302 non-null   int64
 8   Assembly status     8302 non-null   object
dtypes: float64(1), int64(1), object(7)
memory usage: 583.9+ KB
None
```

Exploring Data with Pandas

df.shape()

- Print the shape of the DataFrame
- The number of rows and columns

df.info()

- Provides a summary of the DataFrame
- including data types, index, no of rows, data columns, memory storage, and missing values.

```
>>> euk = pd.read_csv("eukaryotes.tsv", sep="\t")
>>> print(euk.shape)
(8302, 9)
```

```
>>> print(euk.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8302 entries, 0 to 8301
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Species             8302 non-null   object
 1   Kingdom             8302 non-null   object
 2   Class               8302 non-null   object
 3   Size (Mb)           8302 non-null   float64
 4   GC%                 8302 non-null   object
 5   Number of genes     8302 non-null   object
 6   Number of proteins  8302 non-null   object
 7   Publication year    8302 non-null   int64
 8   Assembly status     8302 non-null   object
dtypes: float64(1), int64(1), object(7)
memory usage: 583.9+ KB
None
```

Type

Index/Rows

Summary of data columns

Types of columns

Memory usage

Missing values

Exploring Data with Pandas

df.describe()

- Gives various summary statistics for the DataFrame
- including count, mean, SD, min, 25%, 50%, 75% percentile, and max value.

```
>>> print(euk.describe())
          Size (Mb)  Publication year
count   8302.000000       8302.000000
mean     401.918437       2015.849313
std     1111.538289          2.924305
min        0.011236       1992.000000
25%       19.249775       2015.000000
50%       39.559700       2017.000000
75%      258.848000       2018.000000
max    32396.400000       2019.000000
```

Exploring Data with Pandas

df.head()

- display the first n rows of the DataFrame, default n=5

df.tail()

- display the last n rows of the DataFrame, default n=5

```
>>> print(euk.head())
                    Species   Kingdom           Class  ...  Number of proteins Publication year Assembly status
0  Emiliania huxleyi CCMP1516  Protists  Other Protists  ...               38554             2013       Scaffold
1        Arabidopsis thaliana    Plants     Land Plants  ...               48265             2001     Chromosome
2                 Glycine max    Plants     Land Plants  ...               71219             2010     Chromosome
3         Medicago truncatula    Plants     Land Plants  ...               41939             2011     Chromosome
4        Solanum lycopersicum    Plants     Land Plants  ...               37660             2010     Chromosome

[5 rows x 9 columns]
>>>
>>> print(euk.tail(2))
                     Species Kingdom       Class  ...  Number of proteins Publication year Assembly status
8300  Saccharomyces cerevisiae   Fungi  Ascomycetes  ...                  -             2018     Chromosome
8301  Saccharomyces cerevisiae   Fungi  Ascomycetes  ...                  -             2018     Chromosome

[2 rows x 9 columns]
```

Basic Data Selection

Selecting Columns/Rows

- Isolate a single column using a square bracket [ ] with a column name in it
- Isolating two or more columns using [[ ]]
- filtering a row using .loc[[ ]]
- filtering rows using .iloc[[i:j]]

inclusive of row I, but not row j

```
# Isolate a Single Column
species_column = euk['Species']

# Isolate Two or More Columns
species_kingdom_columns = euk[['Species', 'Kingdom']]

# Filtering based on the index value
df_index_1 = euk.loc[[1]]


# Isolate a Range of Rows
# Using iloc for integer-location indexing
# Isolate rows with index between 2 and 9 (inclusive of 2 but not 10)
rows_range = euk.iloc[2:10]
```

Example2_Basic_Data_selection.py

Basic Data Selection

- **Conditional slicing**

Filter rows based on conditions

- **Value based, greater or smaller than a threshold**
- **>, <, >=, <=, ==, !=**

- Multiple Conditions: Combine conditions using & (AND), or | (OR), ~(NOT)

```
# Example of filtering based on a condition
# Find all genomes where 'Size (Mb)' is greater than 500
large_genomes = euk[euk["Size (Mb)"] > 500]
```

```
# Multiple Conditions Using logical conditions &, |, ~
# Find genomes between a hundred and a thousand megabases using two approaches
filtered = euk[(euk["Size (Mb)"] > 100) & (euk["Size (Mb)"] < 1000)]
```

```
# Find genomes that are not birds or fishes, or have a GC% below 40
filtered3 = euk[~euk["Class"].isin(["Birds", "Fishes"]) | (euk["GC%"] < 40)]
```

Basic Data Selection

- **Aggregating data with .groupby()**

aggregate values by grouping them by specific column values.

Any summary method can be used alongside .groupby(), including .min(), .max(), .mean(), .median(), .sum(), .mode(), and more.

```
# Aggregating data with .groupby()
# Group genomes by 'Kingdom' and calculate the mean size for each kingdom
grouped_kingdom = euk.groupby("Kingdom")[["Size (Mb)"]].mean()
print("\nMean size of genomes by Kingdom:")
print(grouped_kingdom)
```

```
Mean size of genomes by Kingdom:
             Size (Mb)
Kingdom
Animals   1128.217706
Fungi       31.564293
Other       93.289994
Plants     800.261036
Protists    48.429596
```

Using Seaborn to Create beautiful statistical graphics

Introduction to Seaborn

Why Seaborn?

- A Python data visualization library built on top of Matplotlib.
- Provides a high-level interface for creating visually attractive and complex statistical graphics.

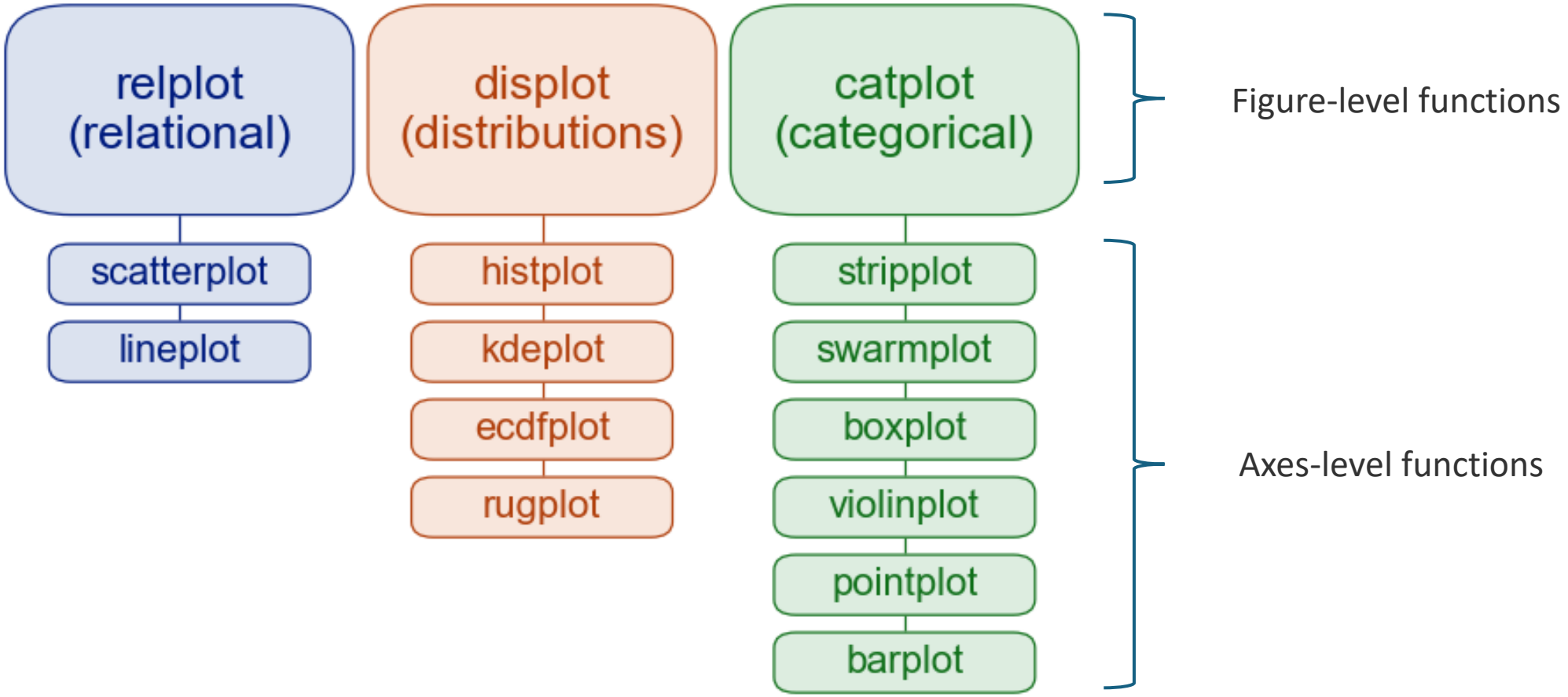
Seaborn vs. Matplotlib:

- Ease of Use: Seaborn simplifies the creation of common plot types, while Matplotlib provides more detailed control.
- Integrated Styling: Seaborn comes with built-in themes, color palettes, and features like sns.set_theme(), which makes visualizations more aesthetically pleasing by default.
- Data Handling: Seaborn works well directly with pandas dataframes, reducing the steps needed to prepare data for plotting compared to Matplotlib.

What Can Seaborn Do?

- Create plots like scatter plots, bar plots, violin plots, box plots, and heatmaps with minimal code.
- Easily customize plots, including themes, color palettes, axes, and more.
- Combine data visualization with statistical aggregation for data analysis.

Introduction to Seaborn



Scatter Plot in Seaborn

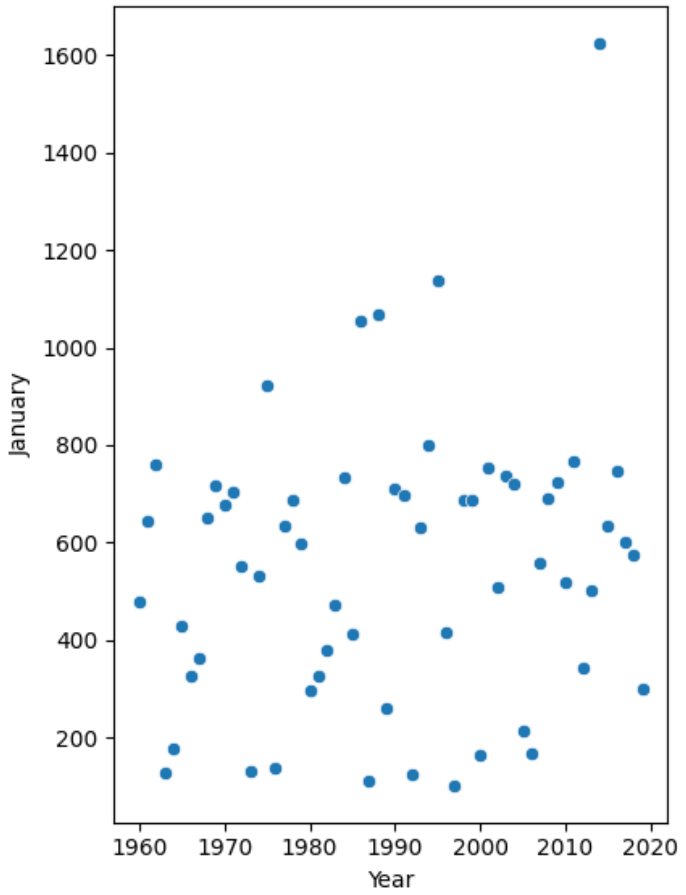
- **Purpose?**

Visualize the relationship between two continuous variables.

- **scatterplot()**

Easily create scatter plots to explore trends and patterns in your data.

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

# Read and prepare the data
df = pd.read_csv('london_rainfall.csv')

# Draw a scatterplot
# to show January Rainfall over years
sns.scatterplot(data=df,
                x='Year', y='January')

plt.show()
```



To display the figure, use **Show()** method.

Example1_scatter_plot_four different plots.py

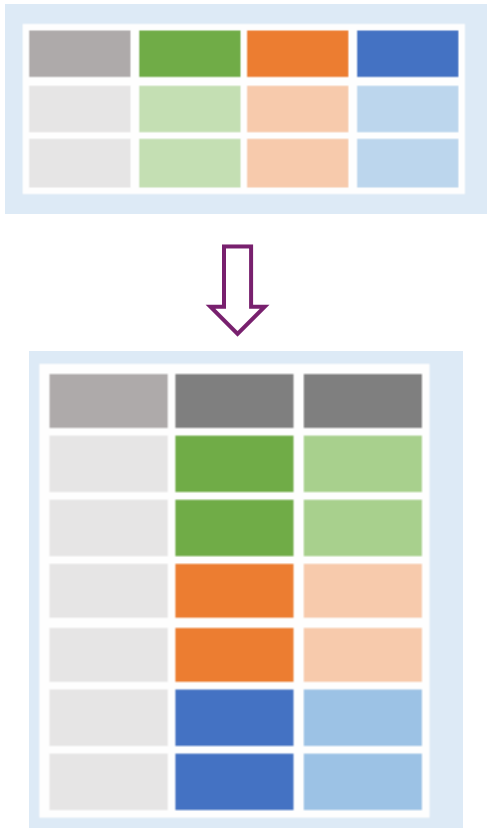
Wide/Summary Format

Year	January	February	March	April	May	June	July	August	September	October	November	December
1960	479	480	339	124	456	428	672	608	753	1555	895	566
1961	644	551	57	508	172	292	262	476	648	568	525	886
1962	761	126	358	418	296	61	828	529	884	376	419	528
1963	129	66	682	521	337	483	281	534	490	347	1190	165
1964	178	168	951	735	461	1101	202	212	111	292	296	299
1965	430	83	580	448	395	546	859	559	1094	189	654	873
1966	326	734	122	986	505	667	780	862	367	1221	417	625
1967	364	486	357	441	1021	510	735	474	595	1038	423	540
1968	651	253	226	442	808	549	794	682	1314	642	443	749
1969	717	450	560	203	595	266	687	930	31	51	687	461
1970	677	399	376	595	251	701	530	536	498	104	1512	385
1971	705	166	424	473	767	1277	193	644	111	578	639	210
1972	553	434	540	383	259	154	272	154	276	173	577	668
1973	133	126	105	581	542	688	491	409	607	386	266	523
1974	531	578	272	152	224	672	298	678	1400	719	1429	344
1975	923	230	697	436	629	232	363	133	1112	232	583	291

Reshaping data in pandas

- **`pd.melt(df)`**

Gather columns into rows



```
# Melt the DataFrame
# convert month columns (January - December)
# into a single column named 'Month'
df_melted = pd.melt(df, id_vars=['Year'],
                    var_name='Month',
                    value_name='Rainfall')
```

- **id_vars=['Year']:**

  specify the columns that should remain fixed

- **value_vars='Month':**

  the columns that are reshaped

	Year	Month	Rainfall
0	1960	January	479.0
1	1961	January	644.0
2	1962	January	761.0
3	1963	January	129.0
4	1964	January	178.0
..	...	...	...
115	1965	December	873.0
116	1966	December	625.0
117	1967	December	540.0
118	1968	December	749.0
119	1969	December	461.0

Long form:

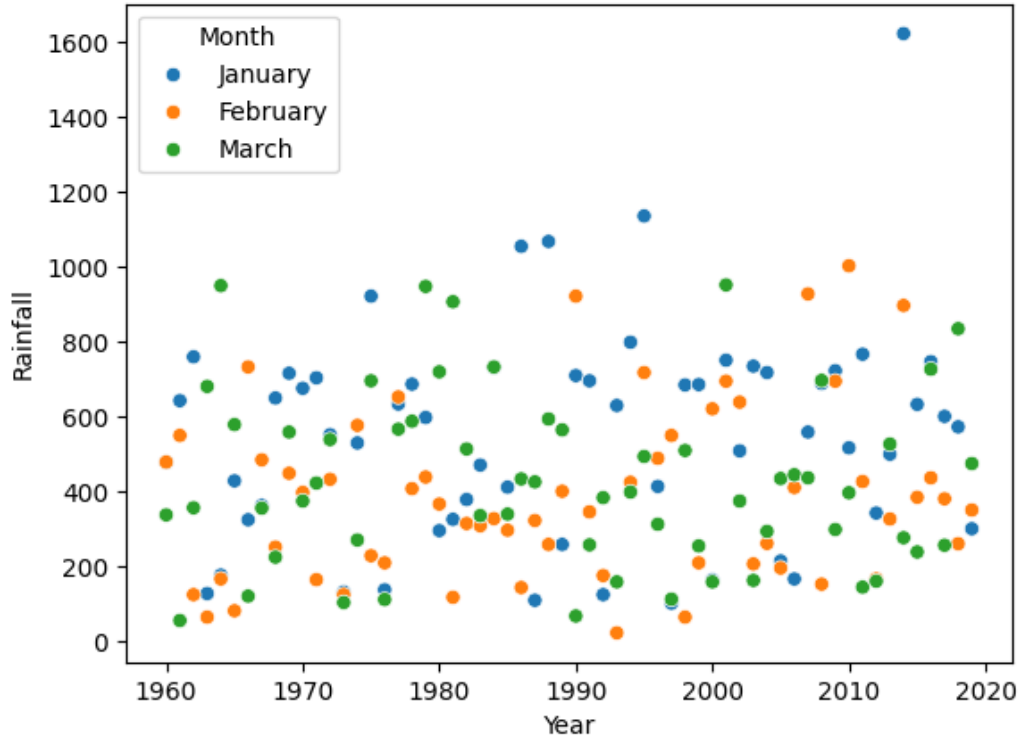
easier to work with for analysis or plotting

Read More Here: https://seaborn.pydata.org/tutorial/data_structure.html

Scatter Plot with two variables

```
df_melted = df.melt(id_vars=['Year'], var_name='Month', value_name='Rainfall')

# Filter the melted DataFrame to include only rows where 'Month' is January, February, or March
filtered = df_melted[df_melted['Month'].isin(['January', 'February', 'March'])]
```

```
# Scatter plot - rainfall for Jan - Mar
sns.scatterplot(data=filtered,
                x='Year', y='Rainfall',
                hue='Month')
```



Example1_scatter_plot_two by two.py

Histogram in Seaborn

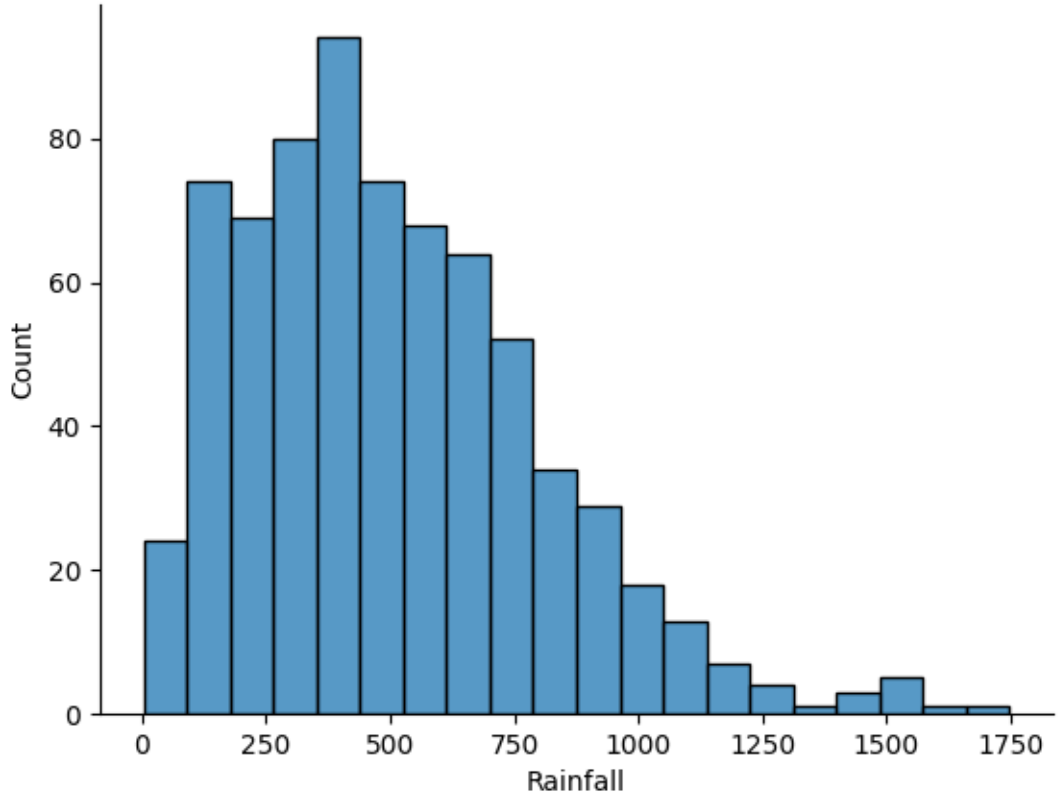
sns.histplot()

Aggregate statistic to compute in each bin:

- count: the number of observations
- density: normalize so that the total area of the histogram equals 1
- percent: normalize so that bar heights sum to 100
- probability or proportion: normalize so that bar heights sum to 1
- frequency: divide the number of observations by the bin width

```
# Create a histogram - rainfall
sns.histplot(data=df_melted, x='Rainfall')

# Remove the top and right spines for better aesthetics
sns.despine()
plt.show()
```

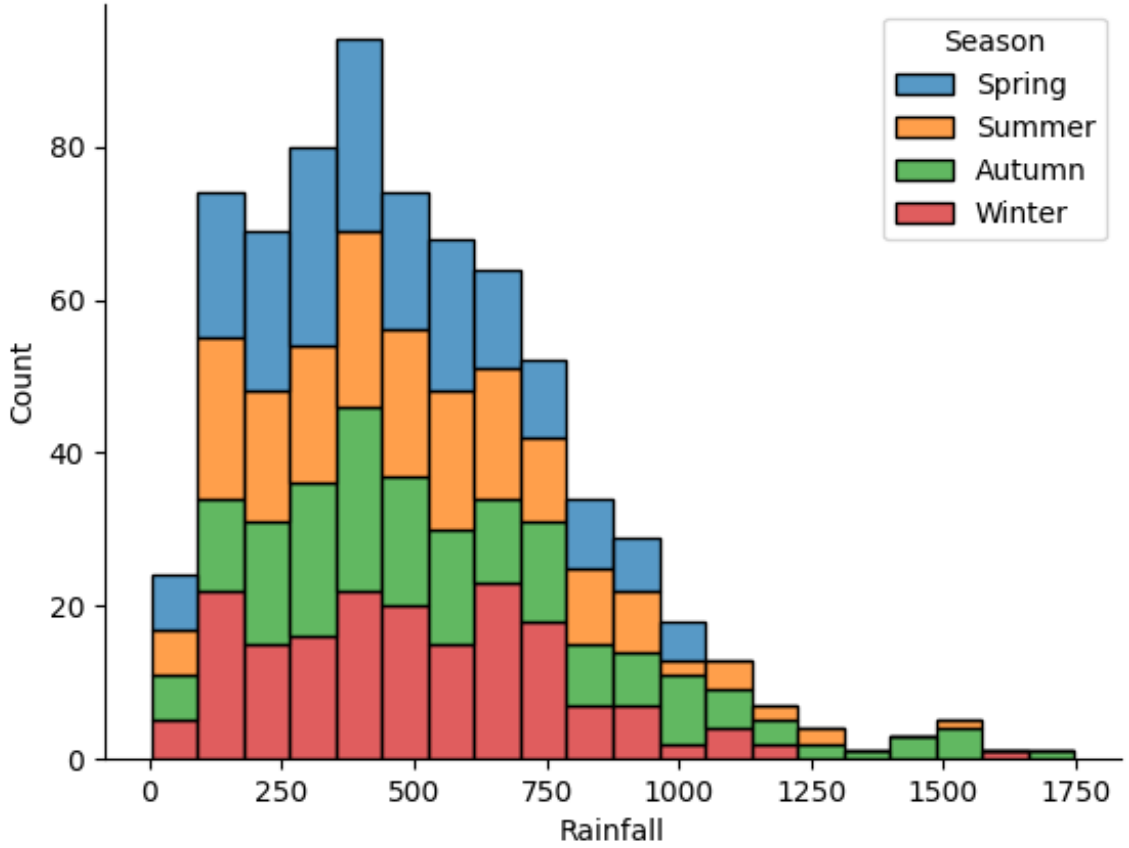
Example2-1_Histplot.py



Histogram in Seaborn

Create a new column ‘season’ using a dictionary

```
# Define the dictionary to map each month to its corresponding season
seasons = {
    'December': 'Winter', 'January': 'Winter', 'February': 'Winter',
    'March': 'Spring', 'April': 'Spring', 'May': 'Spring',
    'June': 'Summer', 'July': 'Summer', 'August': 'Summer',
    'September': 'Autumn', 'October': 'Autumn', 'November': 'Autumn'
}

# Map the 'Month' column to the corresponding season
# using the seasons dictionary
df_melted['Season'] = df_melted['Month'].map(seasons)

# Create a histogram - rainfall
sns.histplot(data=df_melted, x='Rainfall',
             hue='Season',
             multiple="stack",
             hue_order=['Spring', 'Summer', 'Autumn','Winter'])
```



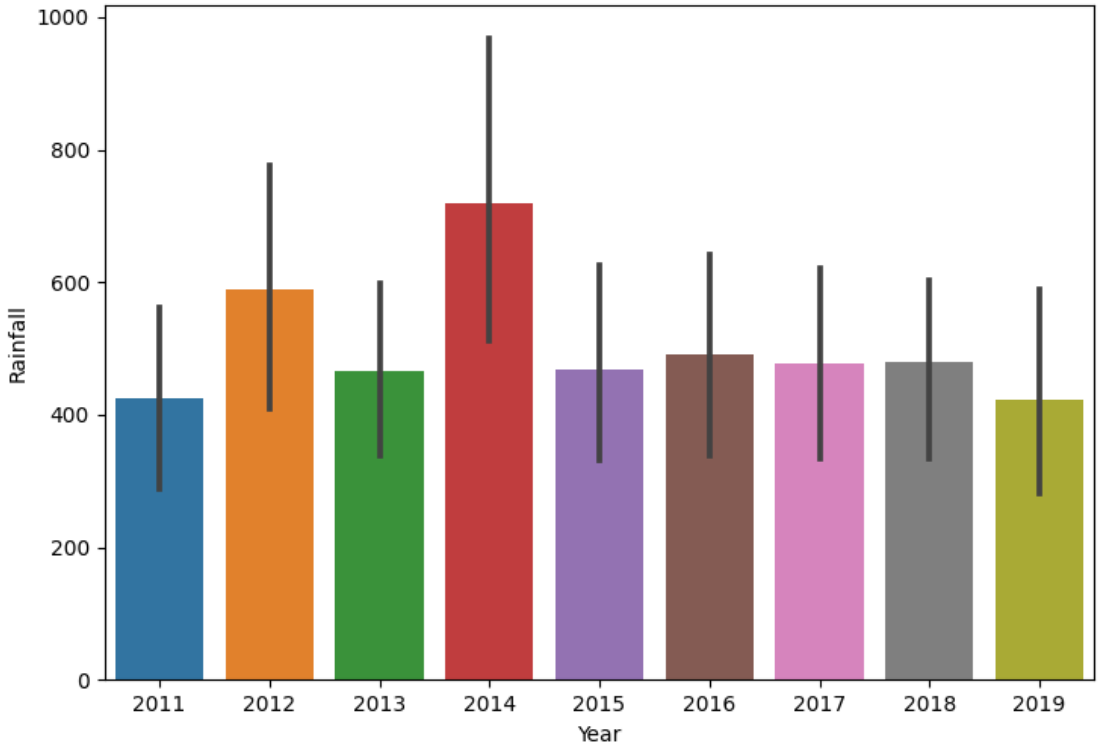
Example2-2_Histplot - season.py

Bar plot in Seaborn

- **barplot()**

display average values with standard deviation

```
df = pd.read_csv('london_rainfall.csv')
df_melted = df.melt(id_vars=['Year'],
                    var_name='Month', value_name='Rainfall')

# Filter the data to include only years since 2010
df_melted = df_melted[df_melted['Year']>2010]

# Create a bar plot - Average rainfall per year
sns.barplot(data=df_melted, x='Year', y='Rainfall')
plt.show()
```

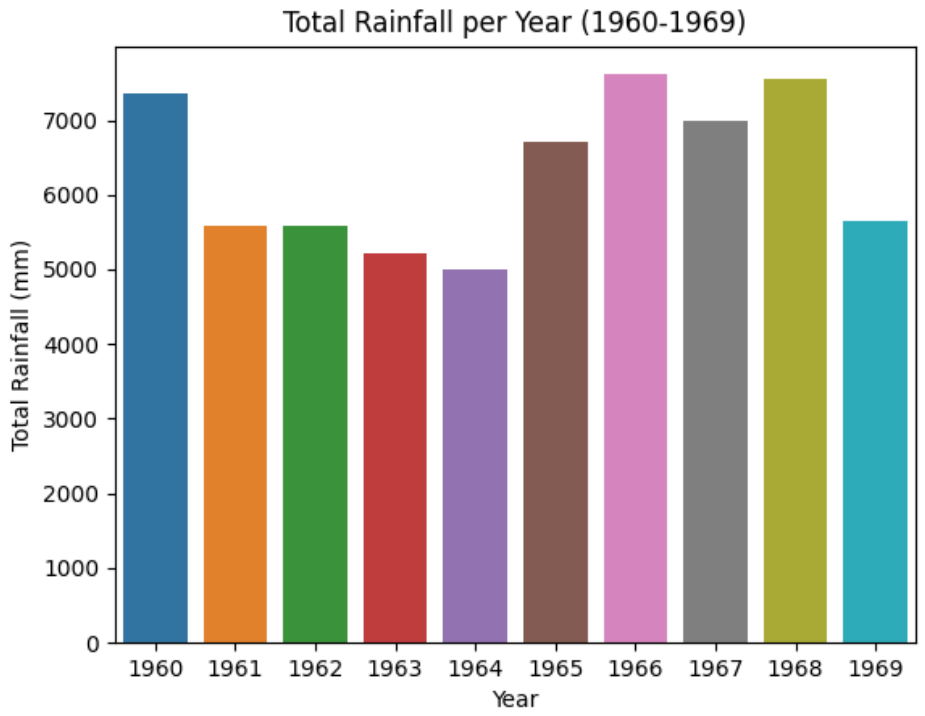


Example3_bar_plot.py

Bar plot in Seaborn

- **barplot()** display count data
- **Summing Values**: Aggregate values within each group using groupby().sum().

```
# Read and prepare the data
df = pd.read_csv('london_rainfall.csv')
# Filter the data to include only years 1960-1969
df = df[0:10]

df_melted = df.melt(id_vars=['Year'],
                    var_name='Month', value_name='Rainfall')

# Group by 'Year' and calculate the total rainfall
total = df_melted.groupby('Year')['Rainfall'].sum().reset_index()

# Create a bar plot - Total rainfall per year
sns.barplot(data=total, x='Year', y='Rainfall')

# Add labels and title
plt.ylabel('Total Rainfall (mm)')
plt.title('Total Rainfall per Year (1960-1969)')
```



Box plot in Seaborn

sns.boxplot(): box-and-whisker plot

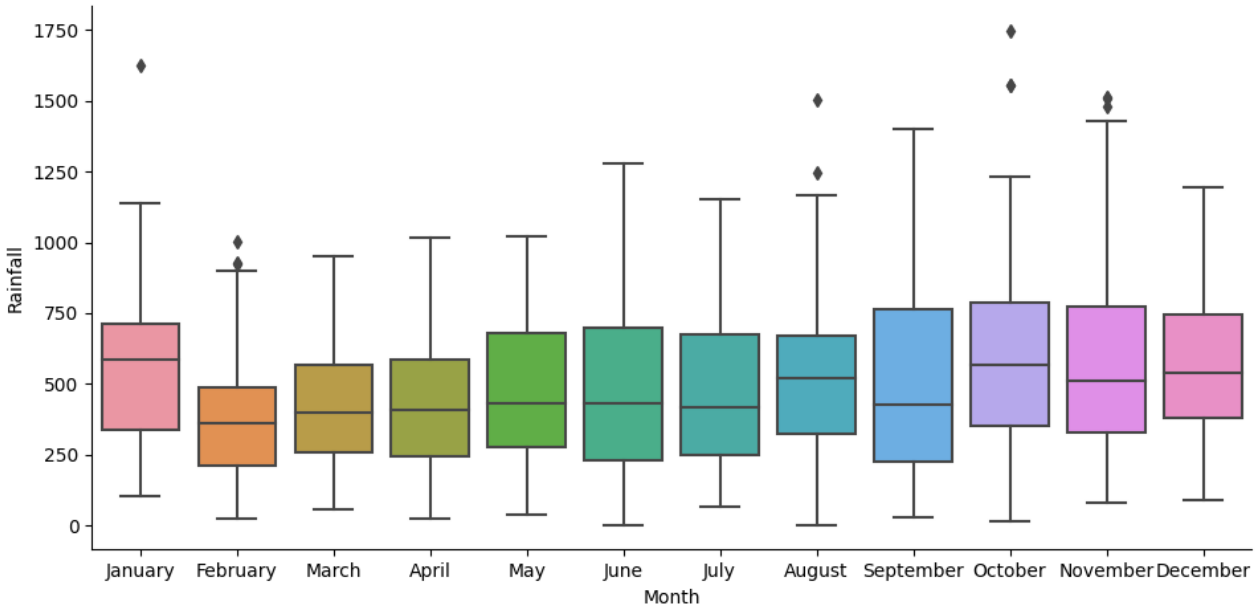
shows distributions with respect to categories.

shows the quartiles of the dataset

whiskers extend to show the rest of the distribution

“outliers” points

```
# Box plot - Monthly rainfall distribution
sns.boxplot(data=df_melted,
            x='Month', y='Rainfall')

sns.despine()
plt.show()
```



Example4_box_plot monthly vs violin plot.py

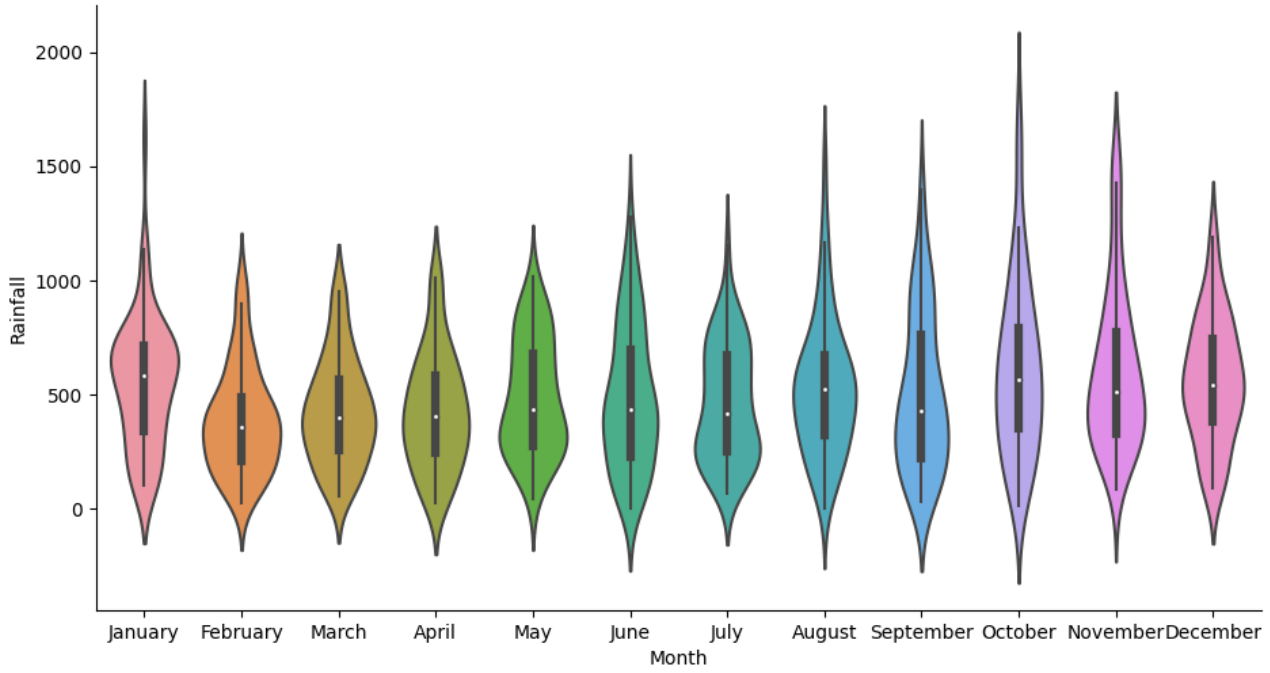
Violin plot in Seaborn

sns.violinplot()

A plot that combines aspects of a box plot and a kernel density plot to show the distribution, probability density, and summary statistics.

Helps visualize the distribution of data across different categories, making it easy to see both summary statistics and the overall distribution shape.

```
# Violin plot - Monthly rainfall distribution
sns.violinplot(data=df_melted,
               x='Month', y='Rainfall')
sns.despine()
plt.show()
```



Example4_box_plot monthly vs violin plot.py

Customizing a Plot with Matplotlib

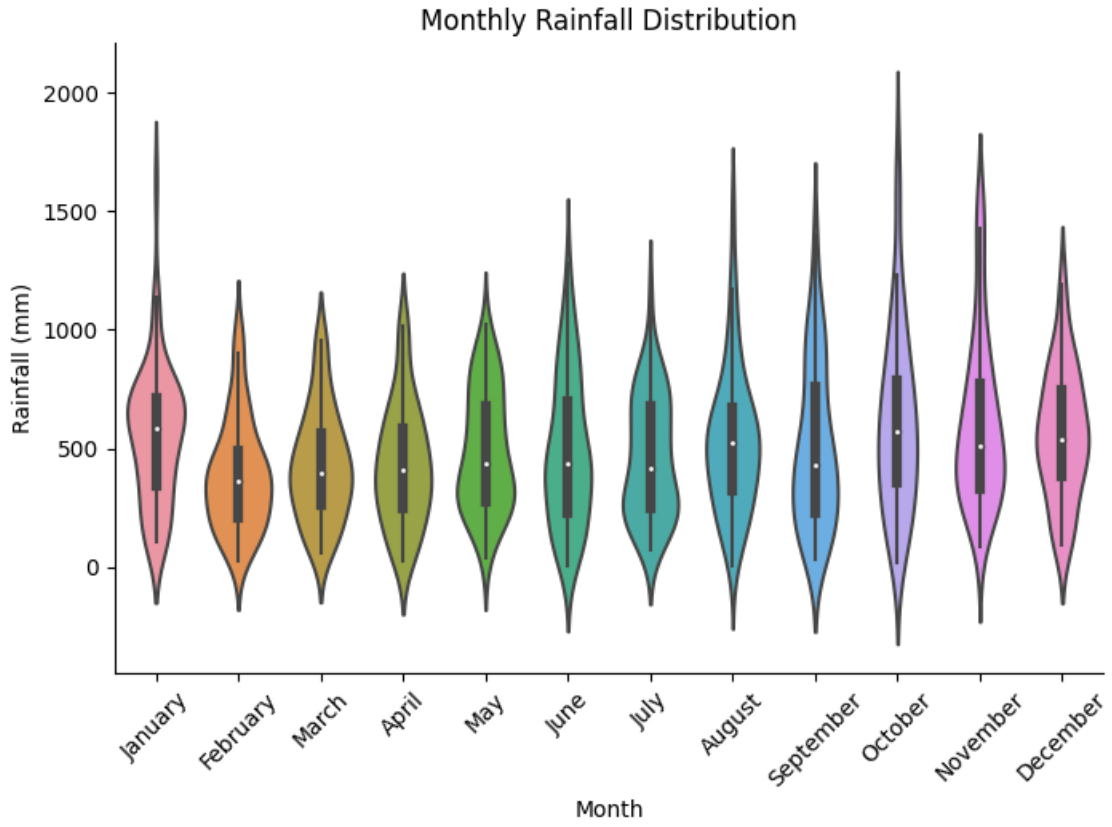
plt.figure() – Creates a new figure

plt. tick_params() – Change the appearance of ticks, tick labels, and gridlines

plt.xlable() – Set the label for the x-axis.

plt.title() – Set a title for the Axes.

```
plt.figure(figsize=(12, 8))
# Violin plot - Monthly rainfall distribution
sns.violinplot(data=df_melted,
               x='Month', y='Rainfall')
# Rotate x-axis labels
plt.tick_params(axis='x', rotation=45)
# Add labels and title
plt.xlabel('Month')
plt.ylabel('Rainfall (mm)')
plt.title('Monthly Rainfall Distribution')
```

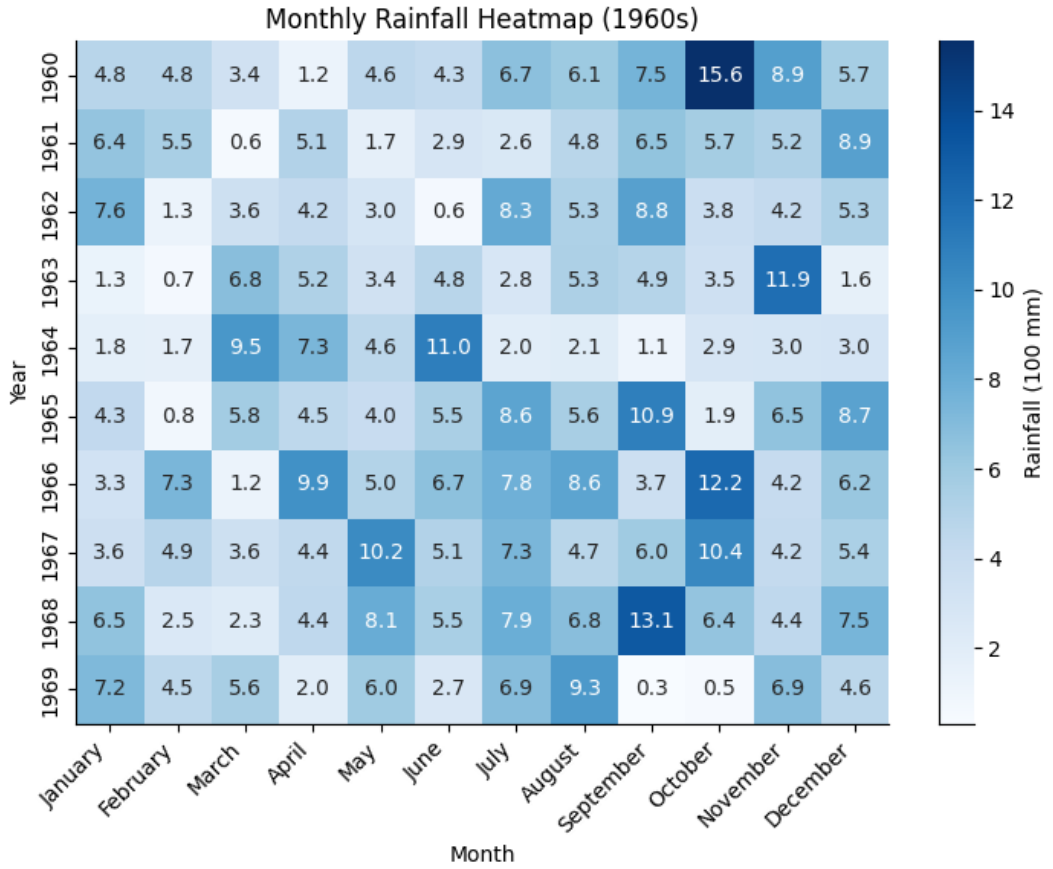


Heat Map in Seaborn

Heat map

- provide an easy way to visualize patterns and compare multiple categories at once
- Use table with wide or summary form

```
# Read and prepare the data
df = pd.read_csv('london_rainfall.csv')
df = df[(df['Year'] <= 1969)]

# Pivot data to prepare for heatmap
df_heatmap = df.set_index('Year').loc[:, 'January':'December']

# Convert rainfall to units of 100 mm
df_heatmap = df_heatmap / 100

# Plot heatmap with annotations
sns.heatmap(df_heatmap, cmap='Blues', annot=True,
            fmt=".1f", cbar_kws={'label': 'Rainfall (100 mm)'})
```



Example5_Heatmap of Monthly Rainfall by Year.py

FacetGrid

sns.FacetGrid()

- create a grid of subplots.
- Ideal for visualizing the distribution or trends of data across multiple subsets.
- Provides an easy way to split a dataset based on the values of specific columns.
- Generates the same plot type for each subset, allowing for consistent comparison.
- Great for comparing multiple categories or observing trends over different groups.

FacetGrid

sns.FacetGrid()

- col="Month": specifies the data should be divided by the Month column.

  Each unique value in the Month column (e.g., January, February, etc.) will create a separate subplot (or facet).

- col_wrap=4: how many columns are displayed per row
- height=3: the height (in inches) of each subplot.
- aspect=1.2: the aspect ratio of each subplot, where aspect is the width divided by the height.

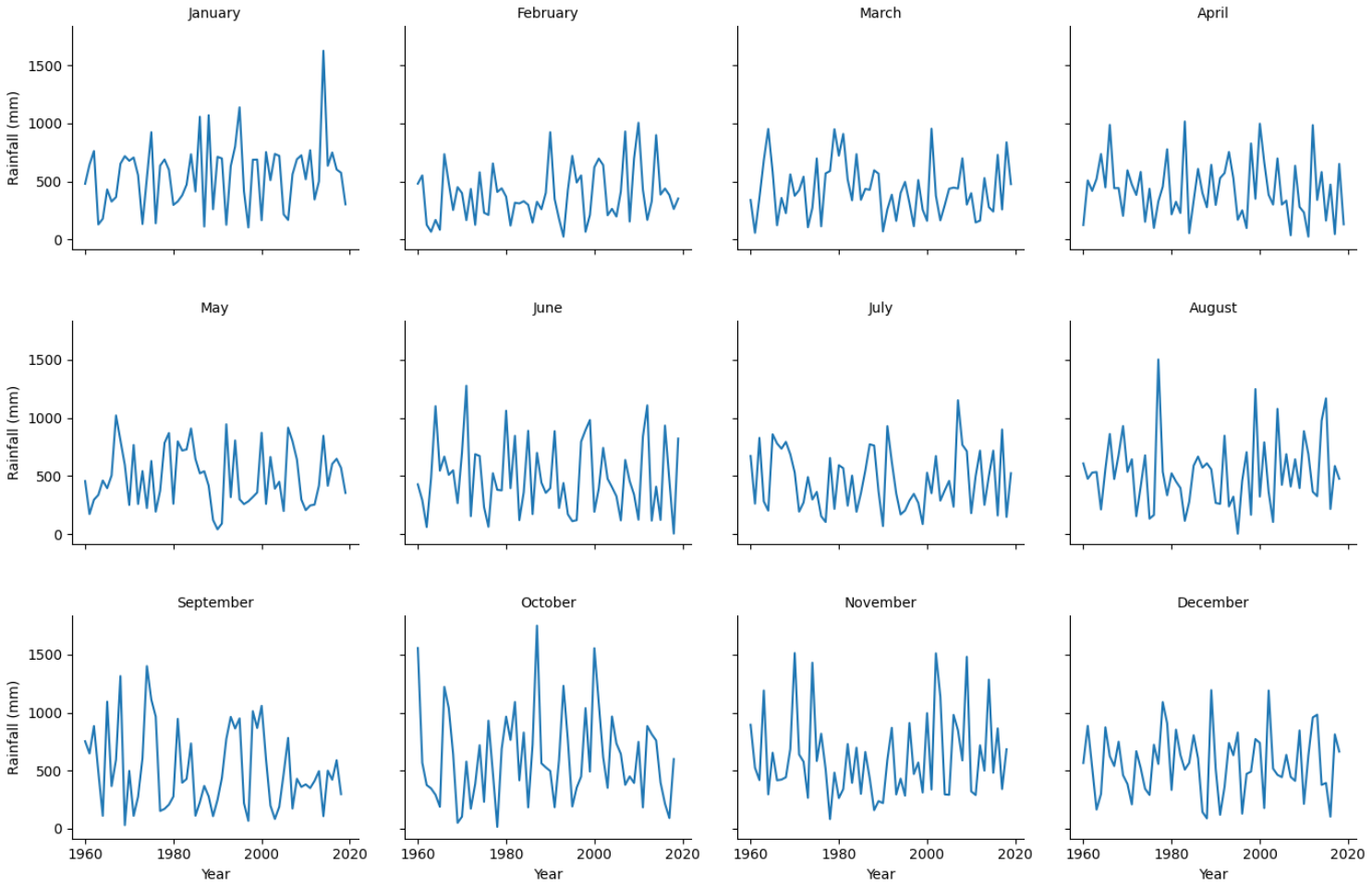
g.map()

- apply a plotting function to each subset of data. Here, we're using sns.lineplot() to draw a line plot on each subplot.
- sns.lineplot: Specifies that a line plot should be used in each facet.
- x-axis and y-axis

```
# Facet plot for each month
g = sns.FacetGrid(df_melted,
                  col="Month",
                  col_wrap=4,
                  height=3,
                  aspect=1.2)
g.map(sns.lineplot,
      "Year", "Rainfall")

sns.despine()
g.set_titles("{col_name}")
g.set_axis_labels("Year",
                  "Rainfall (mm)")
```

FacetGrid

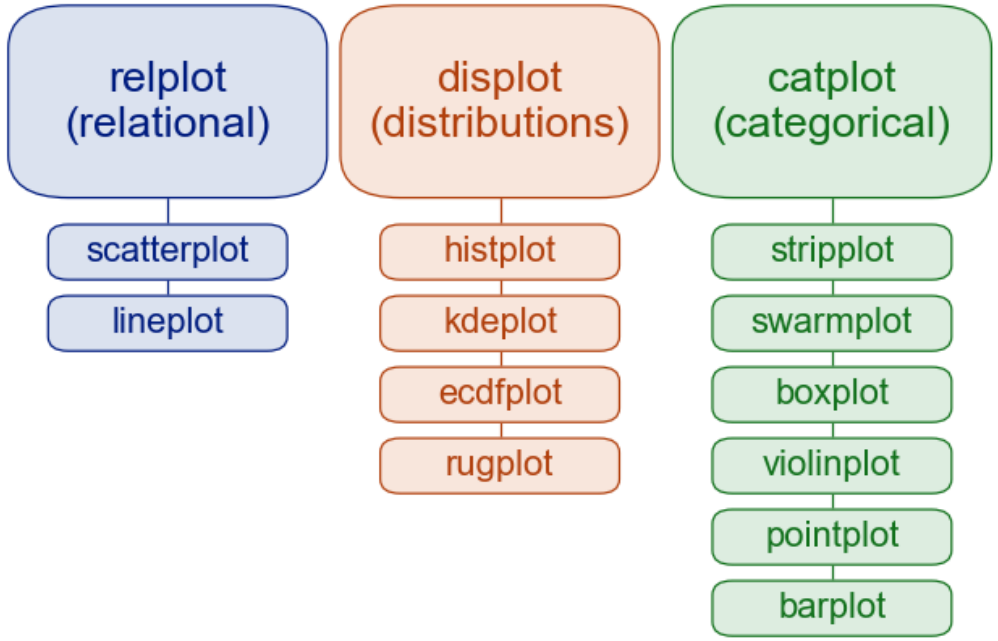
```
# Facet plot for each month
g = sns.FacetGrid(df_melted,
                  col="Month",
                  col_wrap=4,
                  height=3,
                  aspect=1.2)
g.map(sns.lineplot,
      "Year", "Rainfall")

sns.despine()
g.set_titles("{col_name}")
g.set_axis_labels("Year",
                  "Rainfall (mm)")
```



Example6_FacetGrid of Monthly Rainfall by Decade.py

Figure-Level Functions in Seaborn

- **Figure-Level Functions**:
  - High-level plotting functions that manage the entire figure, including the axes and legends.
  - Include functions like sns.catplot(), sns.relplot(), and sns.pairplot().
  - Unlike axes-level functions (e.g., sns.scatterplot()), figure-level functions create a full figure and can easily manage multiple subplots.
- **Benefits of Figure-Level Functions**:
  - Automatically manage aspects like the figure size, titles, and axis labels.
  - Provide consistent formatting and styling when plotting across multiple categories or subsets of data.



Catplot

Categorical scatterplots:

- `stripplot()` (with `kind="strip"`; the default)
- `swarmplot()` (with `kind="swarm"`)

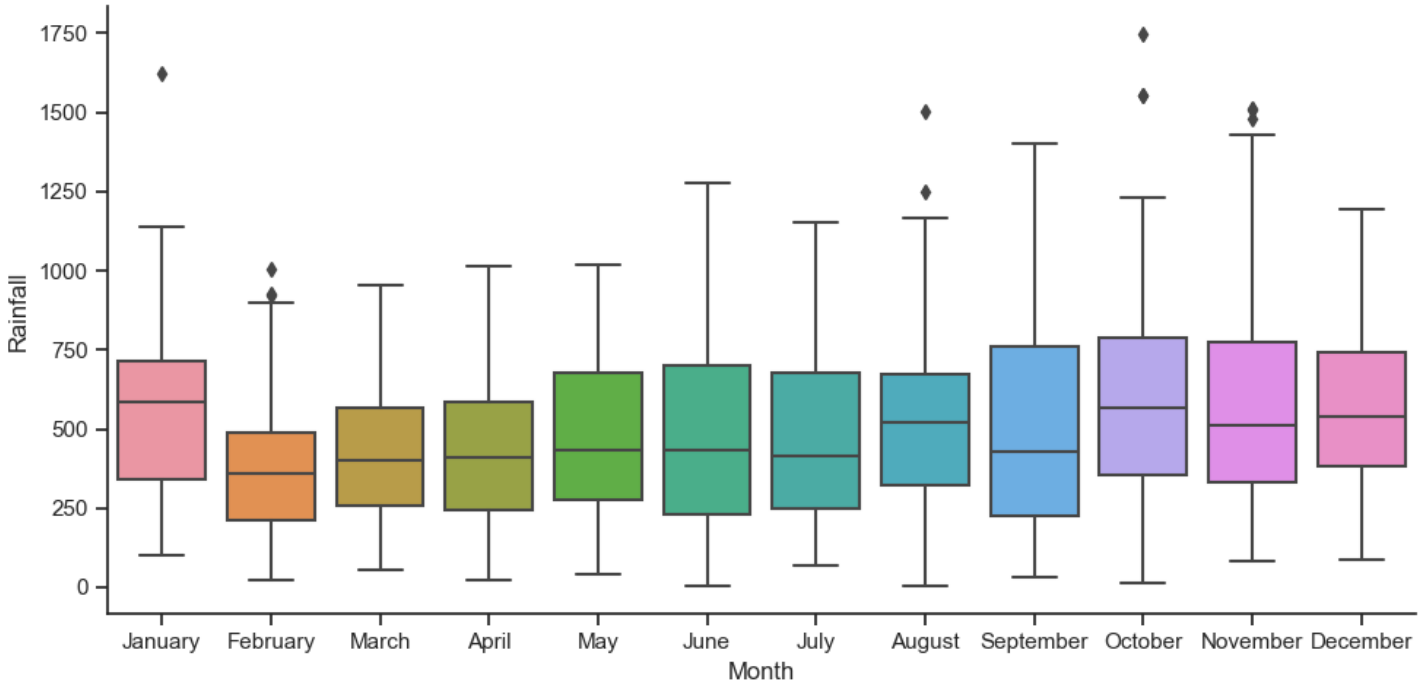
Categorical distribution plots:

- `boxplot()` (with `kind="box"`)
- `violinplot()` (with `kind="violin"`)
- `boxenplot()` (with `kind="boxen"`)

Categorical estimate plots:

- `pointplot()` (with `kind="point"`)
- `barplot()` (with `kind="bar"`)
- `countplot()` (with `kind="count"`)

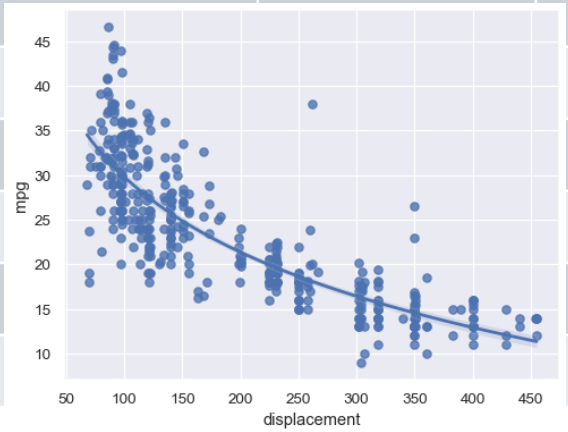
```
sns.catplot(data=df_melted, x='Month', y='Rainfall',
            kind='box',  height=5, aspect=2)
```

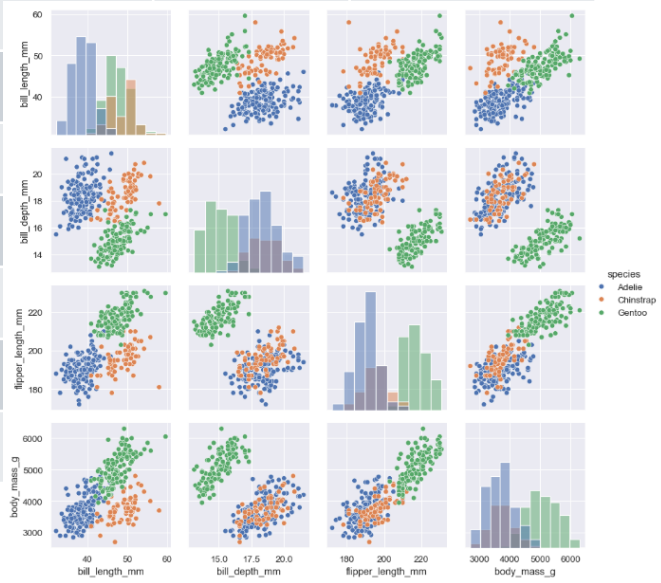


Example7_catplot box.py

Plot Summary

Function interface					Multi-plot grids		
Relational plots	**Distribution plots**	**Categorical plots**	**Regression plots**	**Matrix plots**	**Facet grids**	**Pair grids**	**Joint grids**
relplot	displot	catplot	lmplot	heatmap	FacetGrid	pairplot	jointplot
scatterplot	histplot	stripplot	regplot	clustermap		PairGrid	JointGrid
lineplot	kdeplot	swarmplot	residplot				
	ecdfplot	boxplot					
	rugplot	violinplot					
	distplot	boxenplot					
		pointplot					
		barplot					
		countplot					





https://seaborn.pydata.org/api.html#function-interface

Figure Themeing

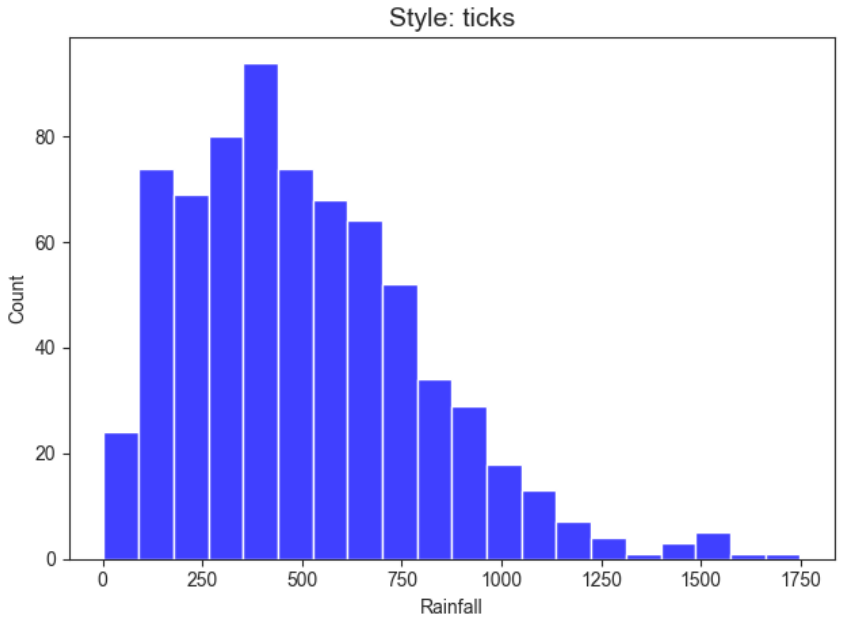
seaborn.**set_theme**(*context='notebook', style='darkgrid', palette='deep', font='sans-serif', font_scale=1, color_codes=True)*

seaborn.**set**(**args, **kwargs*)

Alias for set_theme(), which is the preferred interface.

- Seaborn figure styles
  - sns.set(style="ticks")
  - sns.set_style("ticks")
- Removing axes spines
  - sns.despine(top=True, right=True)
  - sns.despine()

  Remove the top and right spines



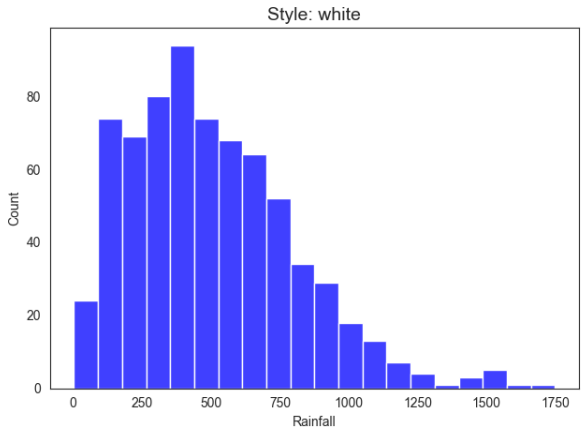
Seaborn Styles

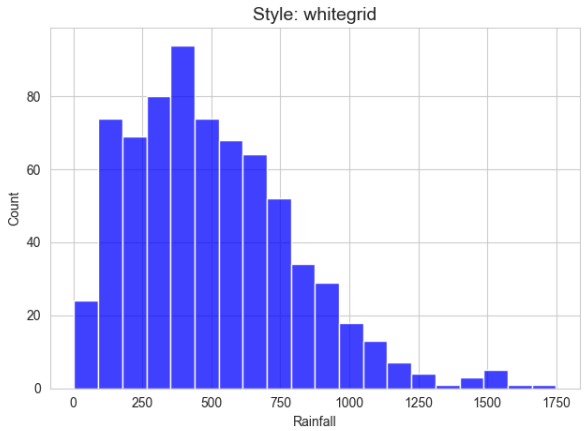
White

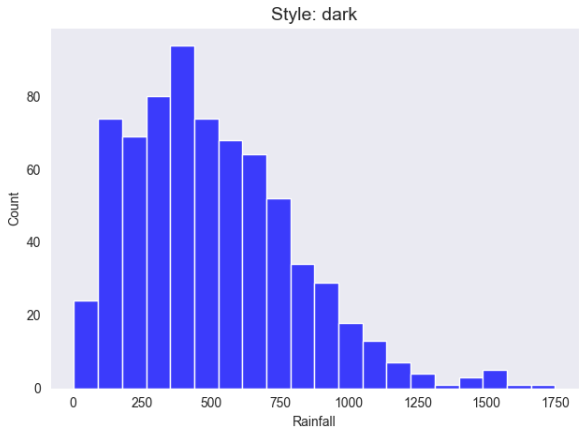
Whitegrid

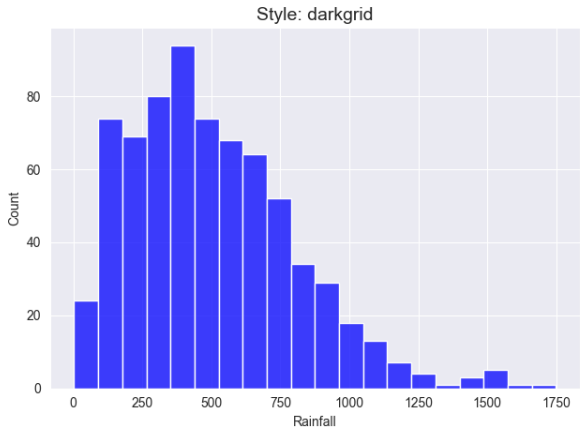
Dark

Darkgrid









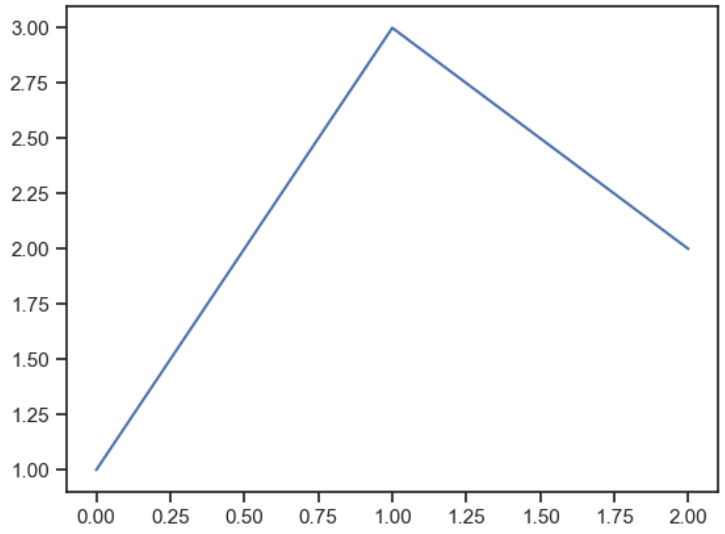
Seaborn Context

seaborn.**set_context**(*context="notebook", font_scale=1, rc=None*)

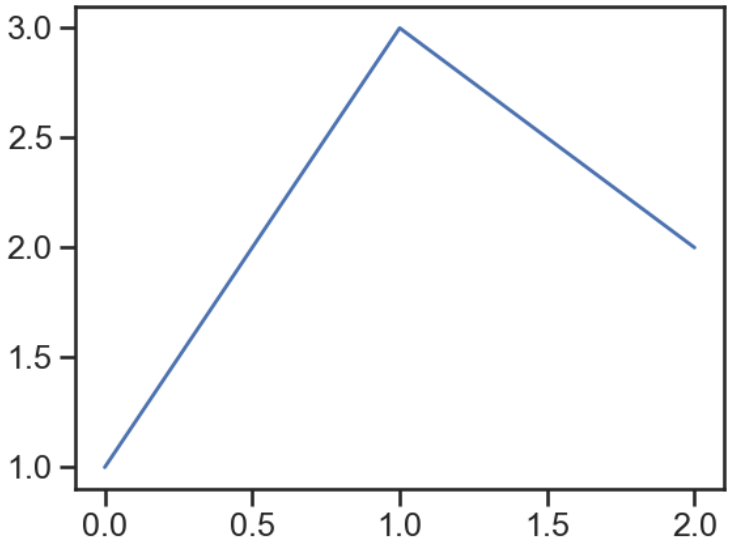
The base context is “notebook”

The other contexts are “paper”, “talk”, and “poster”

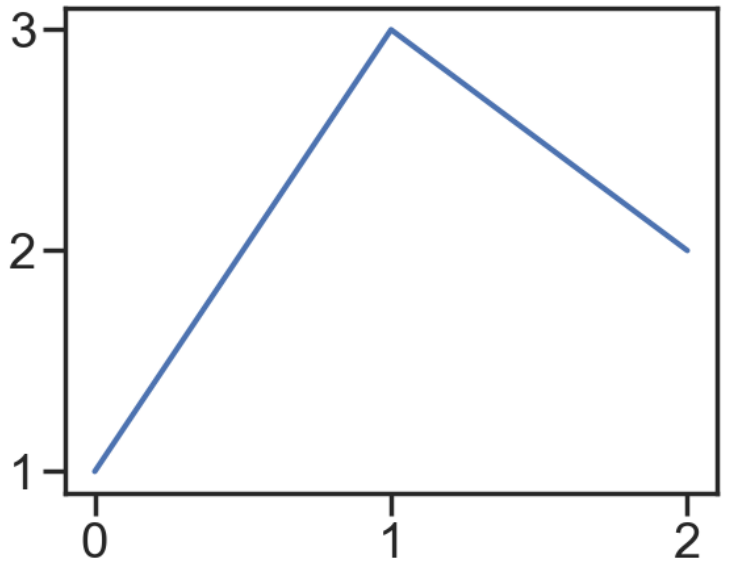
```
sns.set_context("notebook")
sns.lineplot(x=[0, 1, 2], y=[1, 3, 2])
```




```
sns.set_context( context: "talk", font_scale=1.1,
                 rc={"lines.linewidth": 2})
sns.lineplot(x=[0, 1, 2], y=[1, 3, 2])
```



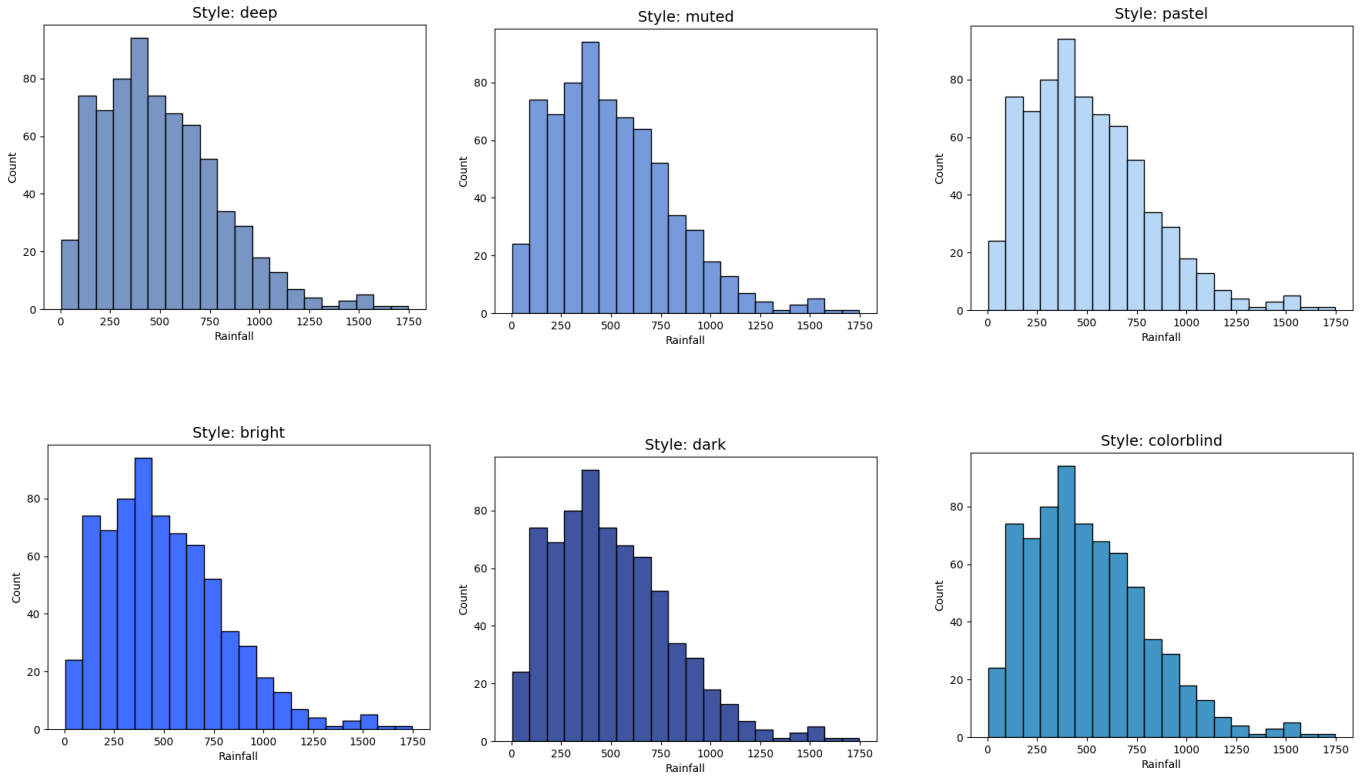

```
sns.set_context("poster", font_scale=1.25,
                rc={"lines.linewidth": 3})
sns.lineplot(x=[0, 1, 2], y=[1, 3, 2])
```



Color palettes

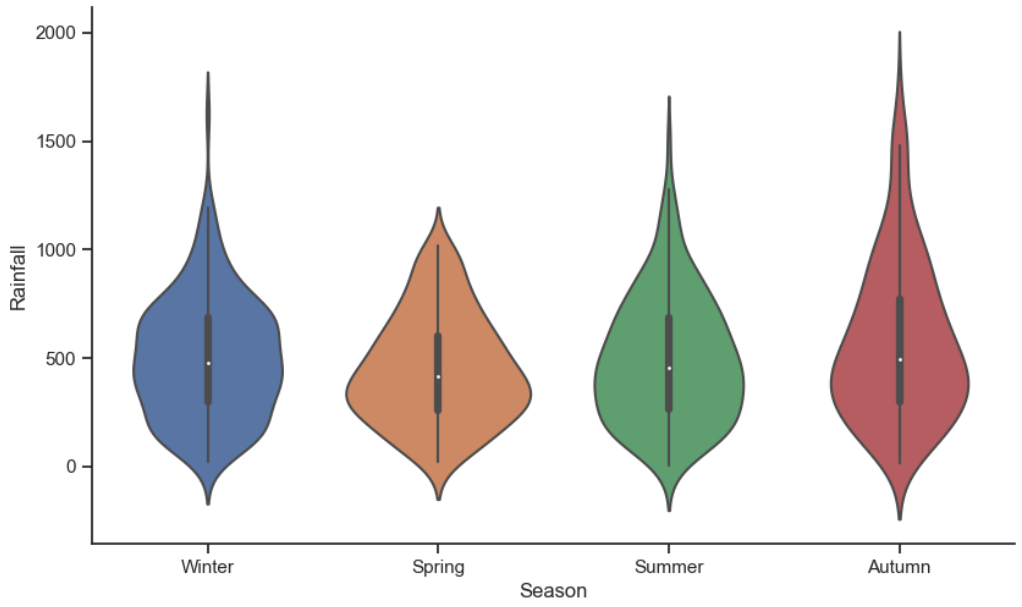
sns.color_palette(): Set the palette

sns.palplot(): function displays a palette

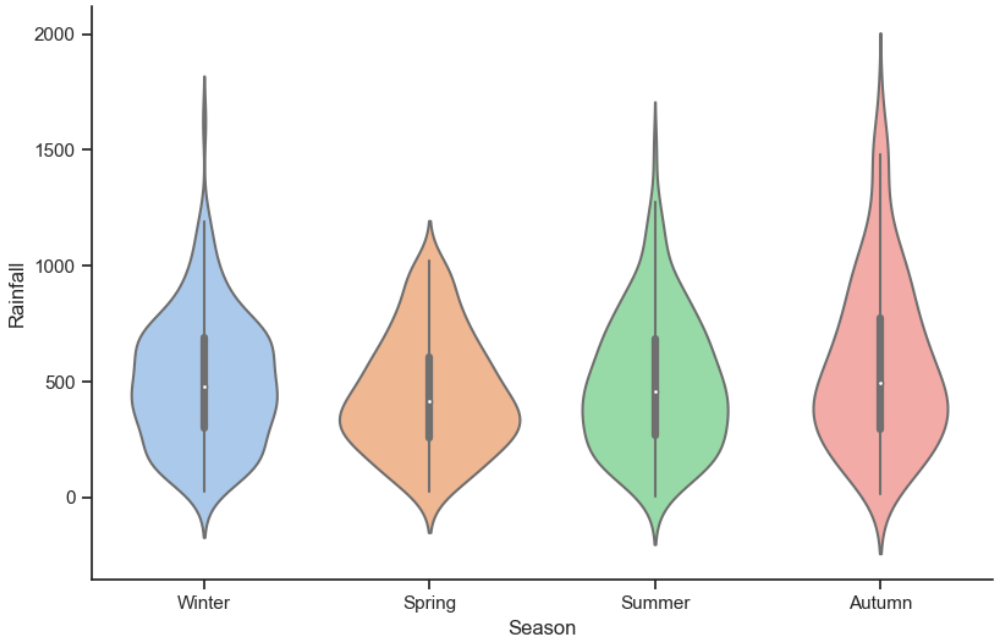


Define a color palette

```
sns.catplot(data=df_melted, x='Season', y='Rainfall',
            kind='violin',  height=5, aspect=2)
```




```
sns.catplot(data=df_melted, x='Season', y='Rainfall',
            kind='violin',  height=5, aspect=2,
            palette="pastel")
```

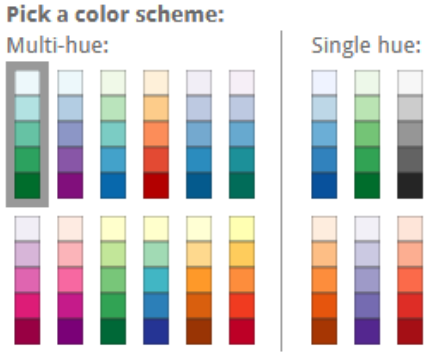


The color toolkit

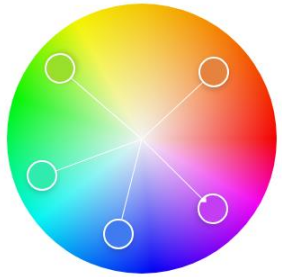
https://colorbrewer2.org/

Three types of color schemes

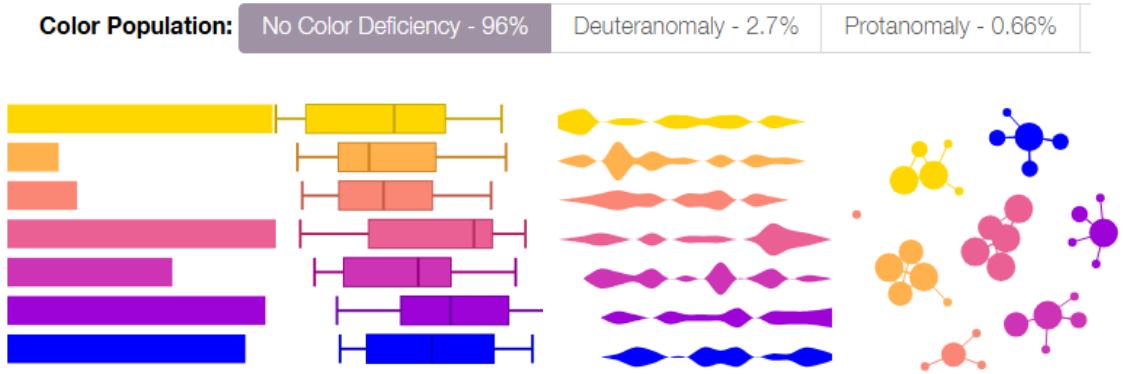
- **Sequential**
- **Diverging**
- **Qualitative**



https://color.adobe.com/create/color-wheel

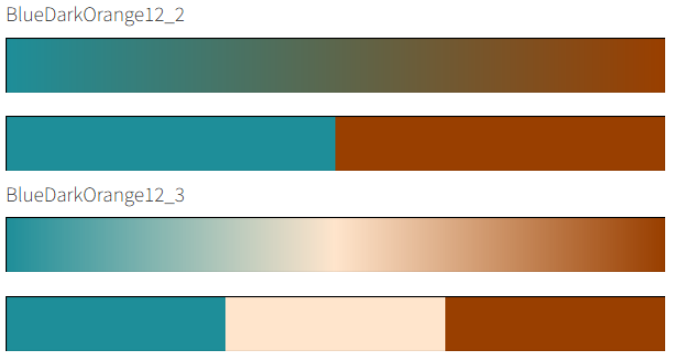


- https://projects.susielu.com/viz-palette



- https://jiffyclub.github.io/palettable/lightbartlein/

- **Diverging**
- **Sequential**



Customizing multiple subplots with Matplotlib

plt.subplots() returns both a figure and an axes, which is useful when:

- You have **multiple subplots**.
- You need **direct access** to the axes for detailed customization using methods like ax.set() or ax.set_title() .

1. ax.set()

- **convenient** when you want to set several properties at once in a clean and concise manner.
- **less control over the individual formatting** of the labels (e.g., you can't change the font size or other stylistic features directly using this method).

```
fig, ax = plt.subplots()
sns.scatterplot(data=df_melted, x='Year', y='Rainfall', ax=ax)
ax.set(xlabel='Year', ylabel='Rainfall (mm)',   title='Seasonal Rainfall (mm)')
```

Customizing multiple subplots with Matplotlib

2. Specific Methods: ax.set_title(), ax.set_xlabel(), ax.set_ylabel()

- Parameters allow you to provide additional parameters, such as fontsize, fontweight
- Offers greater control over the formatting.
- more flexible since you can change the font size, color, or other attributes of each element independently.

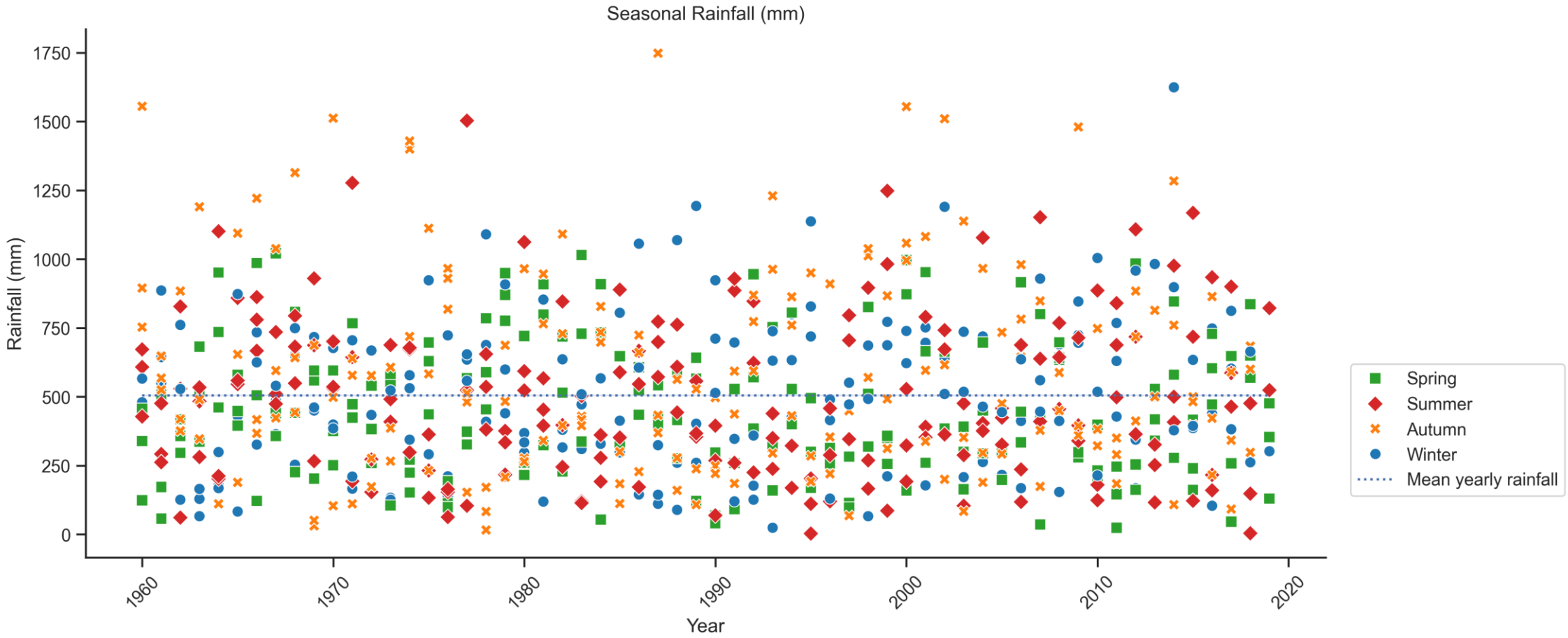
```
fig, ax = plt.subplots()
sns.scatterplot(data=df_melted, x='Year’, y='Rainfall’, ax=ax)
# Set the title and labels for the axis
ax.set_title('Seasonal Rainfall (mm)', fontsize=12)
ax.set_xlabel('Year', fontsize=12)
ax.set_ylabel('Rainfall (mm)', fontsize=12)
# Rotate x-axis labels for better readability
ax.tick_params(axis='x', rotation=45)
# Customize the legend position and format
ax.legend(loc='lower right’,
                bbox_to_anchor=(1.2, 0.1))
```

Scatter Plot with a Line Plot

```
# Set up the figure
fig, ax = plt.subplots(figsize=(14, 6))

hue_ranking = ['Spring', 'Summer', 'Autumn', 'Winter']
palette = {'Winter': "tab:blue", 'Autumn': "tab:orange",
           'Spring': "tab:green", 'Summer': "tab:red"}
markers = {'Spring':'s' ,'Summer': 'D',
           'Autumn':'X',  'Winter':'o'}
# Scatter plot - rainfall for Jan - Mar
sns.scatterplot(data=df_melted, x='Year', y='Rainfall',
                hue='Season', style='Season',
                hue_order=hue_ranking,markers=markers,
                palette=palette,s=50,legend=True,ax=ax)
sns.lineplot(x=df_melted['Year'], y=504,  linestyle=":",
             label='Mean yearly rainfall', color='b', ax=ax)
```

```
sns.despine()
# Set the title and labels for the axis
ax.set_title('Seasonal Rainfall (mm)', fontsize=12)
ax.set_xlabel('Year', fontsize=12)
ax.set_ylabel('Rainfall (mm)', fontsize=12)

# Rotate x-axis labels for better readability
ax.tick_params(axis='x', rotation=45)

# Customize the legend position and format
ax.legend(loc='lower right',
          bbox_to_anchor=(1.2, 0.1))

# Save the figure to a PNG file
fig.savefig('yearly_rainfall_distribution.png',
            dpi=300, bbox_inches='tight')

plt.tight_layout()
plt.show()
```

Scatter Plot with a Line Plot



Example8_scatter_plot_customize.py

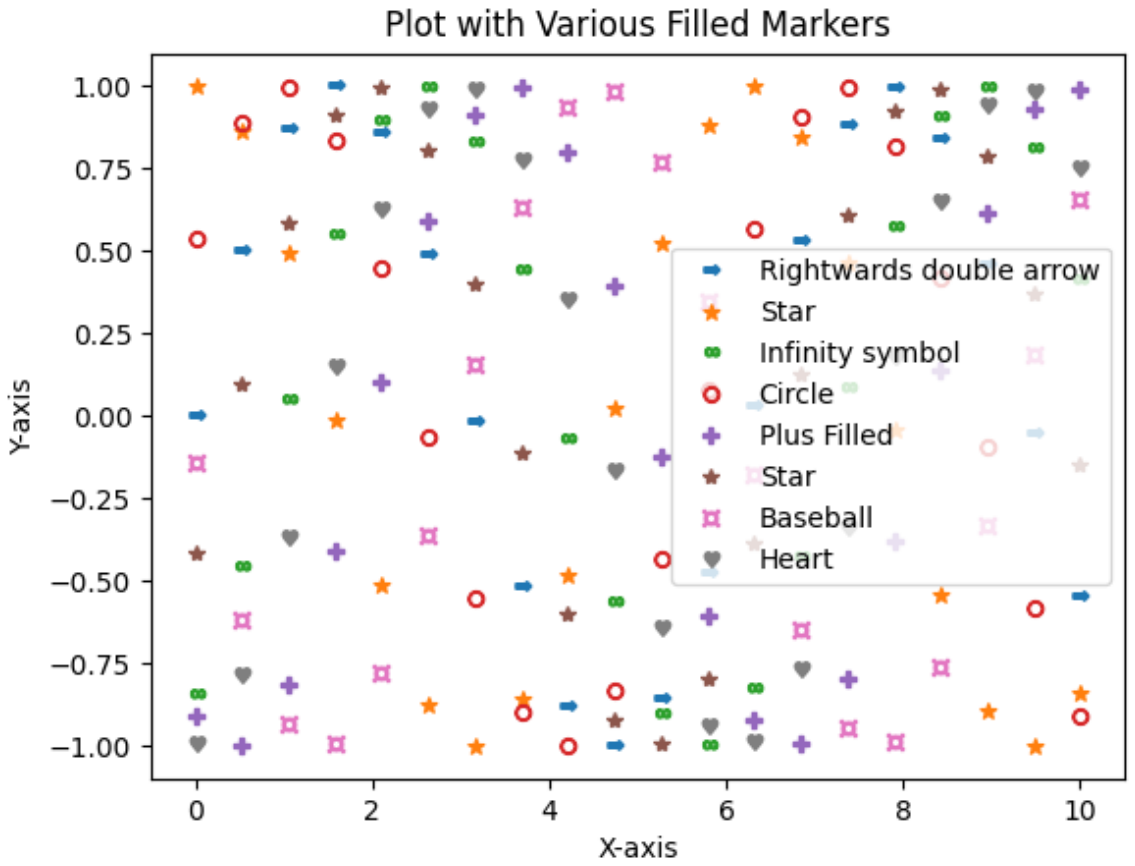
Customized Markers

markers are used to specify the style of the points or vertices in a plot

Markers:

.	Point
,	Pixel
o	Circle
v	Triangle Down
^	Triangle Up
<	Triangle Left
>	Triangle Right
|	Vline
_	Hline

!! Filled and line markers can not be used together



Read More Here: Properties of Mark objects: https://seaborn.pydata.org/tutorial/properties.html

Multiple panels in one figure

- **Steps for Setting up a 2x2 Grid of Subplots:**

1. **Create a 2x2 Subplot Grid**:
    - Use plt.subplots(2, 2) to create a **2x2 grid** of subplots, which means there will be **4 panels** in total.
    - The figsize parameter adjusts the overall size of the figure (in inches).
2. **Access Individual Panels**:
    - axes is a **2D NumPy array** where each element represents an individual subplot.
    - You can access each subplot using indexing (e.g., axes[0, 0] refers to the first panel in the top-left corner).

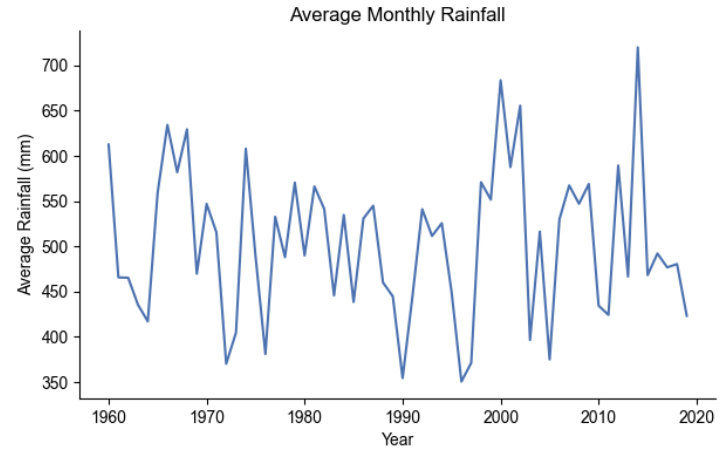
```
# Set up a 2x2 subplot grid
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Line plot
sns.lineplot(df, ax=axes[0, 0])
axes[0, 0].set()

# 2. Violin plot
sns.violinplot(df, ax=axes[0, 1])

# 3. Box plot
sns.boxplot(df, ax=axes[1, 0])

# 4. Swarm plot
sns.swarmplot(df, ax=axes[1, 1])
```

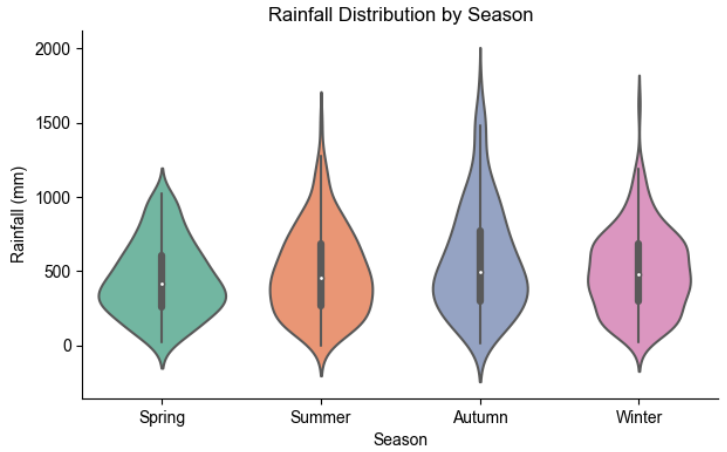
Multiple panels in one figure

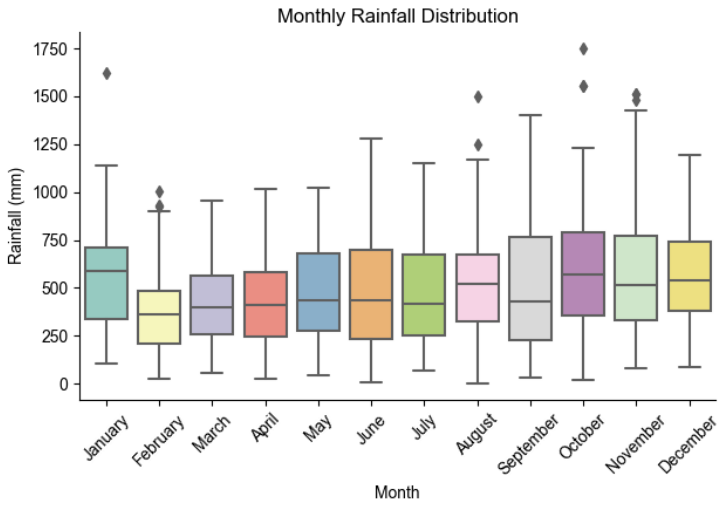
```
# Set up a 2x2 subplot grid
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
sns.set_theme(style="ticks")

# 1. Line plot - Average monthly rainfall over the years
sns.lineplot(data=melted_df.groupby('Year', as_index=False).mean(),
             x='Year', y='Rainfall', ax=axes[0, 0], color='b')
axes[0, 0].set_title('Average Monthly Rainfall')
axes[0, 0].set_ylabel('Average Rainfall (mm)')

# Create custom order and color palette for Seasons
order=['Spring', 'Summer', 'Autumn', 'Winter']
season_palette = {
    'Spring': '#66c2a5',  # Light Green
    'Summer': '#fc8d62',  # Orange
    'Autumn': '#8da0cb',  # Light Blue
    'Winter': '#e78ac3'   # Pink
}
# 2. Violin plot - Rainfall distribution by season
sns.violinplot(data=melted_df, x='Season', y='Rainfall', ax=axes[0, 1],
               palette=season_palette, order=order)
axes[0, 1].set_title('Rainfall Distribution by Season')
axes[0, 1].set_ylabel('Rainfall (mm)')
```

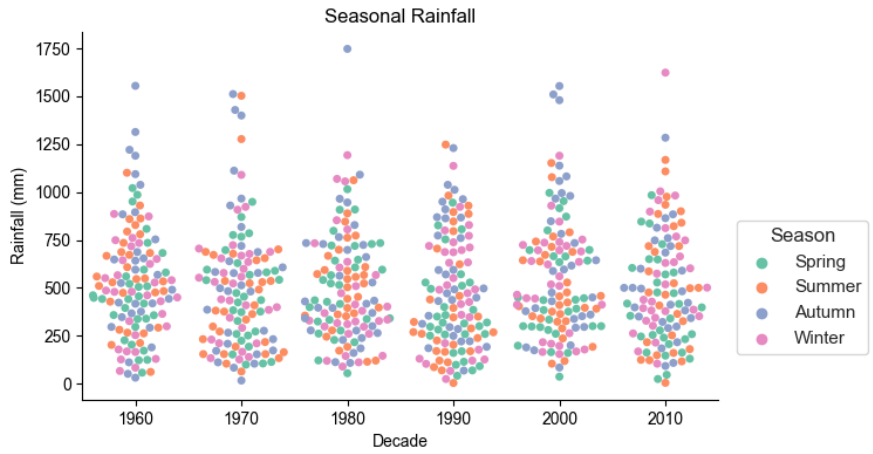
```
order_mth = ['January', 'February', 'March', 'April','May', 'June', 'July',
             'August','September','October', 'November','December']
# 3. Box plot - Monthly rainfall distribution across years
sns.boxplot(data=melted_df, x='Month', y='Rainfall', ax=axes[1, 0],
            palette='Set3', order=order_mth)
axes[1, 0].set_title('Monthly Rainfall Distribution')
axes[1, 0].set_ylabel('Rainfall (mm)')
axes[1, 0].tick_params(axis='x', rotation=45)

# 4. Swarm plot - Total rainfall per year
sns.swarmplot(data=melted_df, x='Decade', y='Rainfall',ax=axes[1, 1],
              hue='Season', hue_order=order, palette=season_palette)
axes[1, 1].set_title('Seasonal Rainfall')
axes[1, 1].set_ylabel('Rainfall (mm)')
axes[1, 1].legend(title='Season', bbox_to_anchor=(1.25, 0.1), loc='lower right')
# Adjust layout and remove spines
sns.despine()

plt.tight_layout()
plt.show()
```

Multiple panels in one figure









[Example9_line plot, violin plot, box plot, and swarm plot.py]

Limitations of Seaborn

Limited Types of Charts

- Seaborn does not directly support creating certain chart types like Stacked bar plots, pie charts or 3D plots.
- Requires the use of Matplotlib for less common plots that are not natively supported.

Customization Complexity

- While Seaborn offers a high-level interface, fine-tuning some elements requires integrating with Matplotlib, which can add complexity.
- Large Datasets: Seaborn may struggle with extremely large datasets due to rendering overhead.
- Works best with Pandas DataFrames. Limited compatibility with other data formats compared to Matplotlib, which is more flexible.

Less Versatile for Non-Statistical Plots

- Focuses on statistical data visualization. General-purpose plots like network graphs or geographic maps are outside its scope.

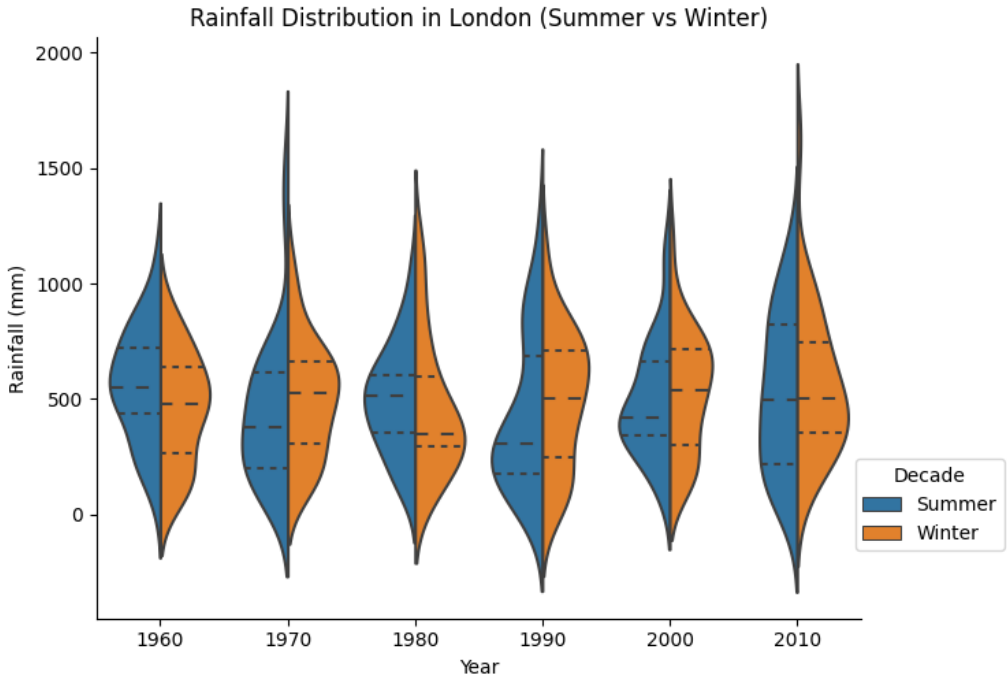
Github Page **BioInfoPythonScripts**

https://github.com/dzhao2019/BioInfoPythonScripts

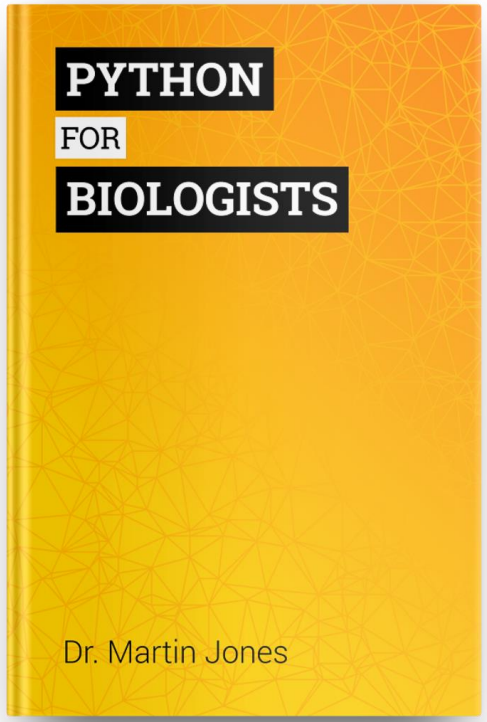
- Example1_Exploring_Data_with_Pandas.py
- Example2_Basic_Data_selection.py
- Example3_Handling_Missing_Data.py

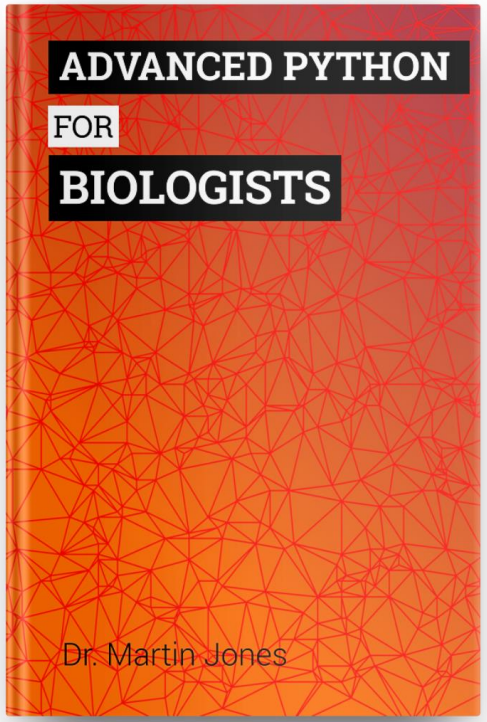
```
12    # Print the shape of the DataFrame
13    # 'euk.shape' provides the number of rows and columns in the
14    print("\neuk.shape", euk.shape)
15
16    # Print the first 5 rows of the DataFrame
17    # 'euk.head()' returns the first 5 rows, giving a preview of
18    print("\neuk.head()", euk.head())
19
20    # Print the last 2 rows of the DataFrame
21    # 'euk.last(2)' returns the first 2 rows, giving a preview of
22    print("\neuk.head()", euk.head(2))
23
```

violin_plot Summer & Winter in decades



Recommended Resources





Example Datasets from BDE:

london_rainfall.csv
eukaryotes.tsv

https://seaborn.pydata.org/tutorial.html

Python Resources by Shelby https://perun.biochem.dal.ca/user-wiki/doku.php?id=python_resources

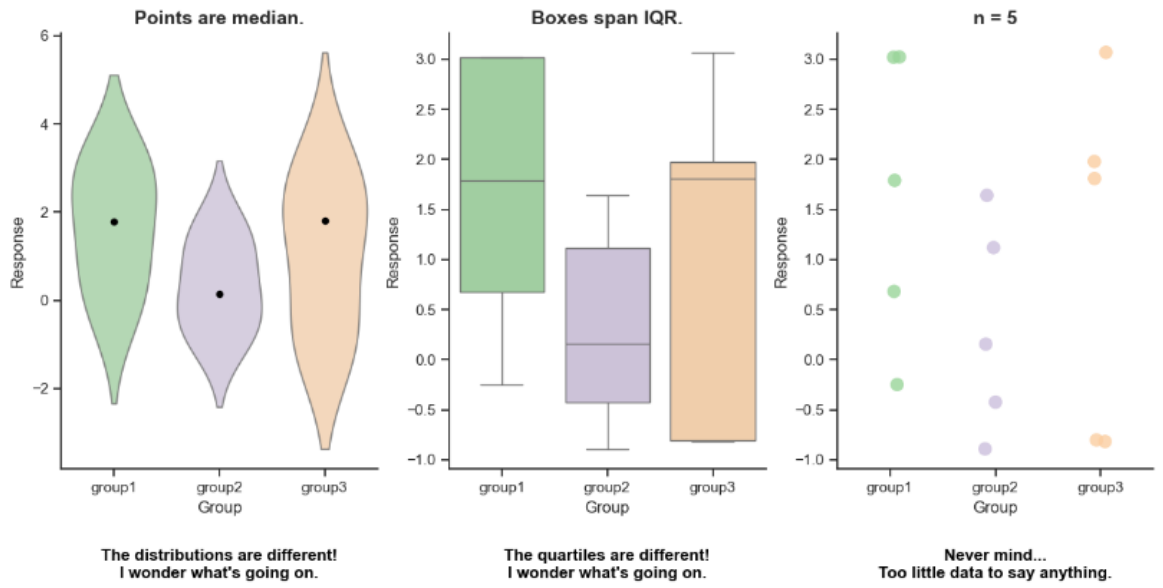
Recommended Resources

https://github.com/cxli233/FriendsDontLetFriends

Python Version: https://github.com/dzhao2019/FriendsDontLetFriends-Python (Ongoing)

2. Small Sample Sizes

Violin plots or any smoothed distribution curves are unreliable for small sample sizes. When sample sizes are small, distributions and quartiles can vary widely, even if the data points are similar. These measures only become meaningful when sample sizes are larger, generally stabilizing when n exceeds 50.



Summary

Data Manipulation Using Pandas

- Load, explore, and manipulate datasets efficiently.
- Techniques for filtering, aggregation, and reshaping data.

Creating Graphics Using Seaborn

- A high-level interface for creating elegant statistical graphics.
- Tools for plotting relationships, distributions, and categorical data.
- Using Seaborn and Matplotlib for advanced plot customization.
- Adding legends, adjusting figure size, and styling visual elements.

External Resources

- Github page for codes and figures: **BioInfoPythonScripts**
- Book: Biological data exploration with Python, pandas and seaborn

Questions?

More slides about exploring data with pandas and how to choose proper plot for your data

Exploring Data with Pandas

dtype argument in read_csv()

- Assign columns types based on a dictionary map

na_values argument in read_csv()

- Tell pandas dash “-” means missing data

```
my_types = {"Species": "string", "Kingdom": "string","Class": "string",
     "Assembly status": "string","Number of genes": "Int64",
    "Number of proteins": "Int64"}

euk = pd.read_csv("eukaryotes.tsv", sep="\t",
    dtype=my_types, na_values=["-"])
```

Exploring Data with Pandas

dtype argument in read_csv()

- Assign columns types based on a dictionary map

na_values argument in read_csv()

- Tell pandas dash "-" means missing data

```
my_types = {"Species": "string", "Kingdom": "string","Class": "string",
     "Assembly status": "string","Number of genes": "Int64",
    "Number of proteins": "Int64"}

euk = pd.read_csv("eukaryotes.tsv", sep="\t",
    dtype=my_types, na_values=["-"])
```

Result from euk.info(): (Partial)

```
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Species             8302 non-null   string
 1   Kingdom             8302 non-null   string
 2   Class               8302 non-null   string
 3   Size (Mb)           8302 non-null   float64
 4   GC%                 7895 non-null   float64
 5   Number of genes     2372 non-null   Int64
 6   Number of proteins  2371 non-null   Int64
 7   Publication year    8302 non-null   int64
 8   Assembly status     8302 non-null   string
dtypes: Int64(2), float64(2), int64(1), string(4)
memory usage: 600.1 KB
```

```
print(euk.describe().round(1))
```

	Size (Mb)	GC%	Number of genes	Number of proteins	Publication year
count	8302.0	7895.0	2372.0	2371.0	8302.0
mean	401.9	43.2	15098.3	17137.3	2015.8
std	1111.5	8.0	11505.4	14735.1	2.9
min	0.0	3.1	3.0	3.0	1992.0
25%	19.2	38.2	7361.5	7010.5	2015.0
50%	39.6	42.7	11998.5	12051.0	2017.0
75%	258.8	49.0	18303.5	21671.0	2018.0
max	32396.4	73.5	119453.0	123467.0	2019.0

Loading Large Datasets

- **Loading only the columns needed:**

  > Blast = pd.read_csv('Blast.tsv', sep='\t',

  usecols=['qseqid', 'pident', 'sscinames'])

- **Saving Data:**

  # To a TSV file:

  > Blast.to_csv('Blast_simple.tsv', sep='\t', index=False)

```
>>> Blast2.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 523312 entries, 0 to 523311
Data columns (total 3 columns):
 #   Column     Non-Null Count   Dtype
---  ------     --------------   -----
 0   qseqid     523312 non-null  object
 1   pident     523312 non-null  float64
 2   sscinames  521389 non-null  object
dtypes: float64(1), object(2)
memory usage: 12.0+ MB
```

```
>>> Blast.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 523312 entries, 0 to 523311
Data columns (total 18 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   qseqid       523312 non-null  object
 1   sseqid       523312 non-null  object
 2   stitle       523312 non-null  object
 3   evalue       523312 non-null  float64
 4   pident       523312 non-null  float64
 5   qcovhsp      523312 non-null  int64
 6   nident       523312 non-null  int64
 7   mismatch     523312 non-null  int64
 8   length       523312 non-null  int64
 9   slen         523312 non-null  int64
 10  qlen         523312 non-null  int64
 11  qstart       523312 non-null  int64
 12  qend         523312 non-null  int64
 13  sstart       523312 non-null  int64
 14  send         523312 non-null  int64
 15  staxids      523312 non-null  object
 16  sscinames    521389 non-null  object
 17  sskingdoms   516745 non-null  object
dtypes: float64(2), int64(10), object(6)
memory usage: 71.9+ MB
```

Adding Headers to Datasets

- **Setting the names argument when using read_csv():**

  > Blast = pd.read_csv('Blast.tsv', sep='\t',

  names=['qseqid', 'evalue', 'pident', 'qcovhsp', 'length', 'sscinames'])

- **Setting the columns property to an existing DataFrame:**

  > Blast = pd.read_csv('Blast.tsv', sep='\t', headers=None)

  > Blast.columns = ['qseqid', 'evalue', 'pident', 'qcovhsp', 'length', 'sscinames']

Handling Missing Data in Pandas

Missing data can distort your analysis, so handling it correctly is essential.

Methods:

- **dropna()**: Remove rows or columns with null values.
- **fillna() or fill_value**: Replace null values with a specific value.

	Year	Rainfall_Jan	Rainfall_Feb
0	2020	5.0	7.1
1	2021	NaN	2.4
2	2022	8.2	NaN
3	2023	NaN	NaN

```python
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {
    'Year': [2020, 2021, 2022, 2023],
    'Rainfall_Jan': [5.0, np.nan, 8.2, np.nan],
    'Rainfall_Feb': [7.1, 2.4, np.nan, np.nan]
}
df = pd.DataFrame(data)
# Drop rows where any values are null
df_dropped = df.dropna(how='any')
print(df_dropped)

# Keep only the rows with at least 2 non-NA values.
df_dropped = df.dropna(thresh=2)
print(df_dropped)

# Fill missing values with a specified value (e.g., replace NaN with 0)
df_filled = df.fillna(0)
print(df_filled)
```

Handling Missing Data in Pandas

Missing data can distort your analysis, so handling it correctly is essential.

Methods:

- **dropna()**: Remove rows or columns with null values.
- **fillna() or fill_value**: Replace null values with a specific value.

```
   Year  Rainfall_Jan  Rainfall_Feb
0  2020           5.0           7.1
```

```
   Year  Rainfall_Jan  Rainfall_Feb
0  2020           5.0           7.1
1  2021           NaN           2.4
2  2022           8.2           NaN
```

```python
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {
    'Year': [2020, 2021, 2022, 2023],
    'Rainfall_Jan': [5.0, np.nan, 8.2, np.nan],
    'Rainfall_Feb': [7.1, 2.4, np.nan, np.nan]
}
df = pd.DataFrame(data)
# Drop rows where any values are null
df_dropped = df.dropna(how='any')
print(df_dropped)

# Keep only the rows with at least 2 non-NA values.
df_dropped = df.dropna(thresh=2)
print(df_dropped)

# Fill missing values with a specified value (e.g., replace NaN with 0)
df_filled = df.fillna(0)
print(df_filled)
```

Handling Missing Data in Pandas

Missing data can distort your analysis, so handling it correctly is essential.

Methods:

- **dropna()**: Remove rows or columns with null values.
- **fillna() or fill_value**: Replace null values with a specific value.

```
   Year  Rainfall_Jan  Rainfall_Feb
0  2020           5.0           7.1
1  2021           0.0           2.4
2  2022           8.2           0.0
3  2023           0.0           0.0
```

```python
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {
    'Year': [2020, 2021, 2022, 2023],
    'Rainfall_Jan': [5.0, np.nan, 8.2, np.nan],
    'Rainfall_Feb': [7.1, 2.4, np.nan, np.nan]
}
df = pd.DataFrame(data)
# Drop rows where any values are null
df_dropped = df.dropna(how='any')
print(df_dropped)

# Keep only the rows with at least 2 non-NA values.
df_dropped = df.dropna(thresh=2)
print(df_dropped)

# Fill missing values with a specified value (e.g., replace NaN with 0)
df_filled = df.fillna(0)
print(df_filled)
```

Violin plot in Seaborn

Lambda Function

- decades = lambda x: (x // 10) * 10

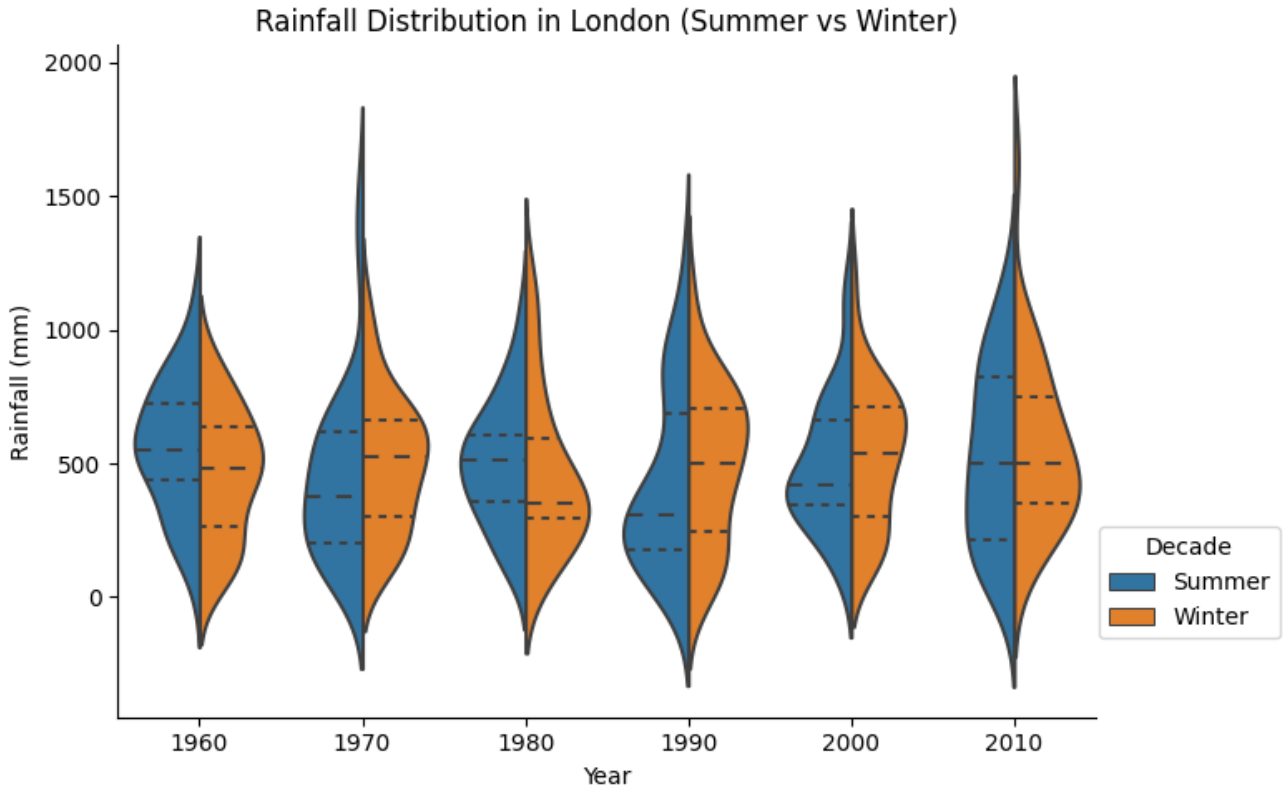
x // 10 gets the decade value, for example, 1960 // 10 gives 196.

(x // 10) * 10 converts it back to the starting year of the decade, such as 1960 for the decade of the 1960s.

- df['Decade'] = df['Year'].apply(decades)

adds a new column to your DataFrame where each year is replaced with the starting year of its decade.

```
decades = lambda x: (x // 10) * 10
df_melted['Decade'] = df_melted['Year'].apply(decades)

# Draw a nested violinplot
sns.violinplot(data=df_melted, x='Decade',
               y='Rainfall', hue='Season',
               split=True, inner='quart',
               hue_order=['Summer','Winter'])

plt.legend(title='Decade', bbox_to_anchor=(1.20, 0.1),
           loc='lower right')
```



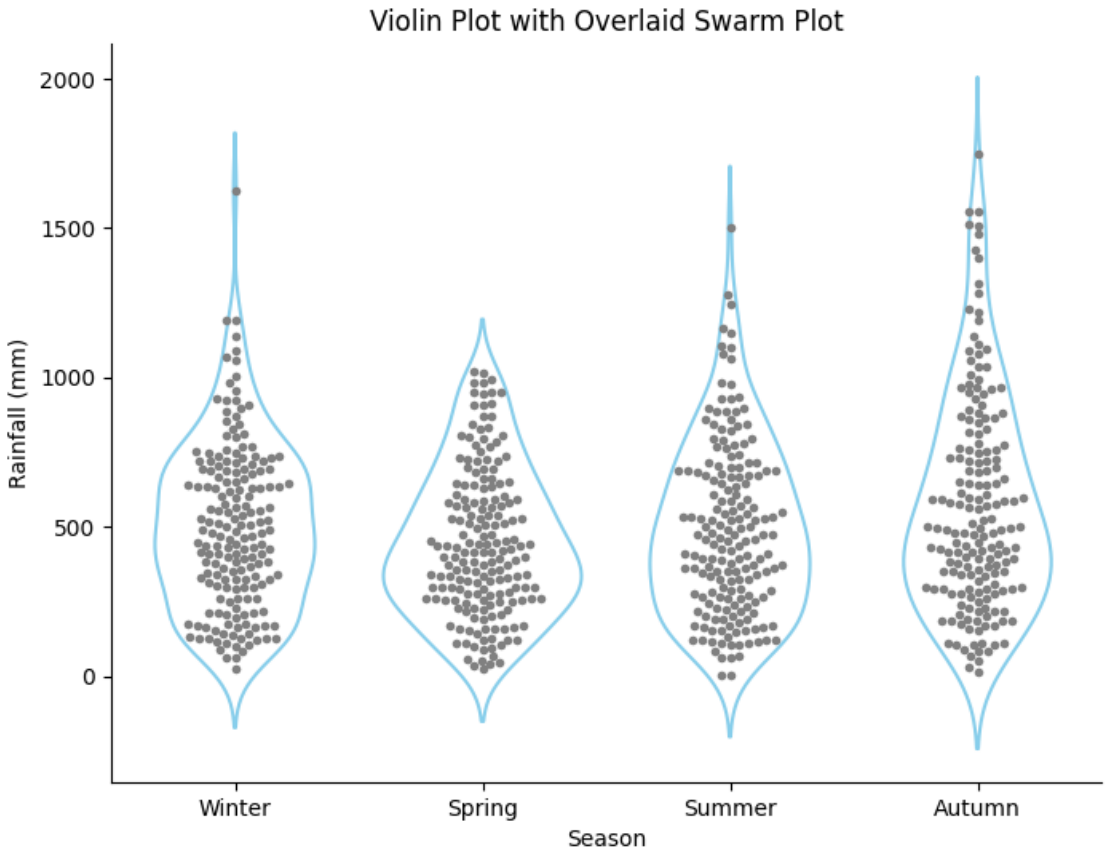
violin_plot Summer & Winter in decades.py

Violin Plot with Overlaid Swarm Plot

sns.violinplot()

- color='skyblue': Sets the color of the violin plot.
- inner=None: Removes inner elements (e.g., quartiles or medians) from the violin plot to avoid cluttering when the swarm plot is added.
- Fill=False: draw as line art, not solid patch

```
# Violin plot
sns.violinplot(data=df_melted,
               x='Season', y='Rainfall',
               color='skyblue', inner=None,
               fill=False)
# Swarm plot
sns.swarmplot(data=df_melted,
              x='Season', y='Rainfall',
              color='grey', size=4)
```



Violin Plot with Overlaid Swarm Plot.py

How to choose proper plot for your data

Choosing the Right Plot Type

Understand Your Data Type

- Categorical Data: Use bar plots, count plots, or box plots to compare categories.
- Continuous Data: Use histograms, scatter plots, line plots, or violin plots to visualize trends or distributions.

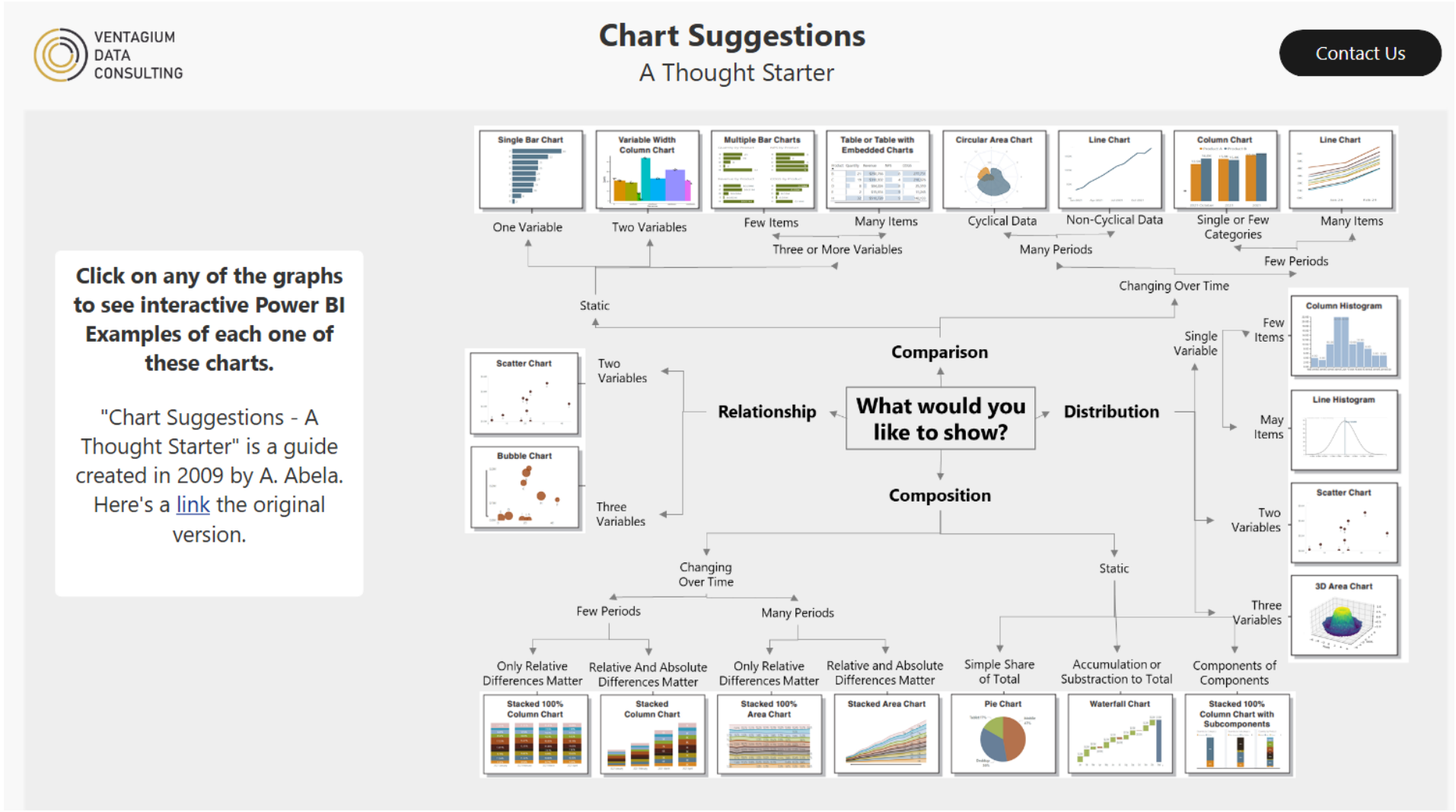
Purpose of Visualization

- Comparison: Use bar plots, line plots, or scatter plots to compare values across categories or time.
- Distribution: Use histograms, KDE plots, or ECDF plots to show the spread and density of data.
- Relationships: Use scatter plots, heatmaps, violin plots or pair plots to explore relationships between variables.
- Composition: Use stacked bar plots or pie charts to show the proportions of different groups.

Quick Guide to Choosing Plots

- Distribution: Histogram, KDE plot, density plot, box plot, violin plot, dot map
- Trends Over Time: Line plot, bar plot
- Relationship Between Variables: Scatter plot, pair plot, heatmap
- Categories Comparison: Bar plot, count plot, violin plot, box plot
- Proportions: Stacked bar plot, pie chart

Interactive version of the "Chart Chooser"



https://bit.ly/Ventagium-Chart-Chooser

Scatter Plot

← Comparison > Static > **Two Variables** and Relationship / **Two Variables**

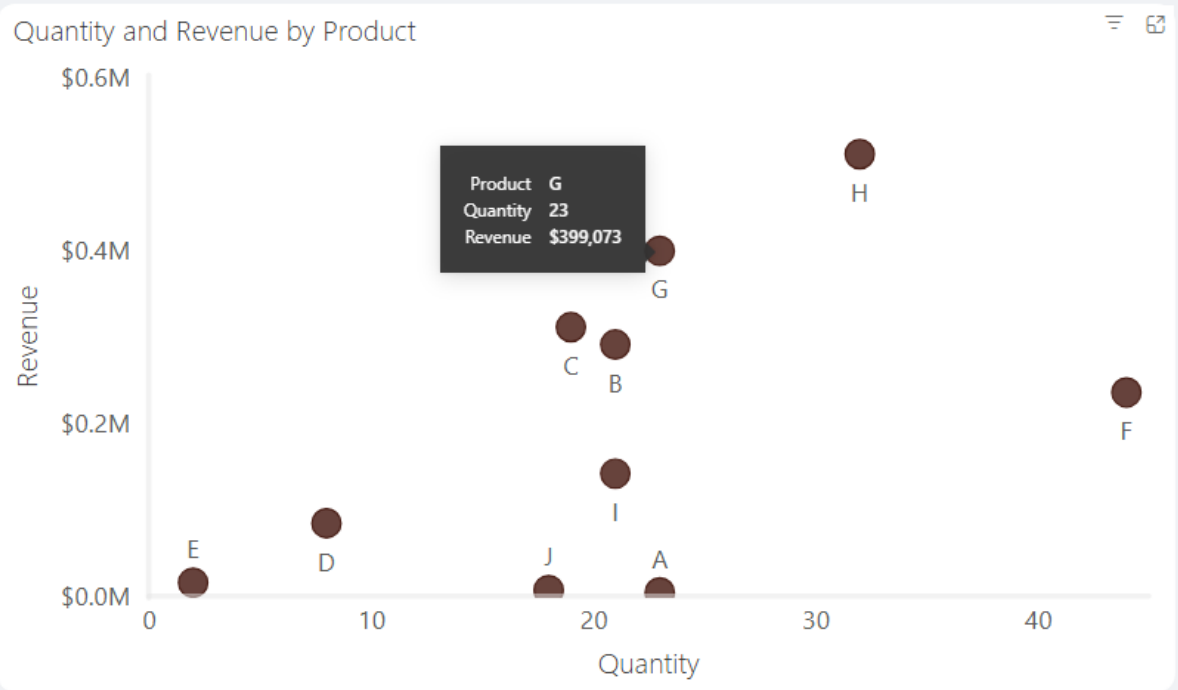
Table

Here´s the data as shown in a table for Revenue and Quantity Sold by Product.

Product	Quantity	Revenue
A	23	$4,324
B	21	$290,766
C	19	$310,802
D	8	$84,224
E	2	$15,614
F	44	$235,356
G	23	$399,073
H	32	$510,720
I	21	$141,624
J	18	$7,092
Total	**211**	**$1,999,595**

Scatter Plot

When you want to show how items compare or relate among two features, Scatter Plots show a clear way of distinction among variables. In this example, it is easy to compare among products like H and F.



Column Histogram

VENTAGIUM DATA CONSULTING

Example Data

This table shows the height of 100 people.

Person	Height (cm)
Christine	184
Debra	185
Emma	186
Bryan	187
Helen	188
Karen	189
Gerald	190
Katherine	190
Jesse	192
Lawrence	193
Ashley	194

Column Histogram

Column Histograms are a great way to represent graphically a distribution. They allow us to comprehend rapidly the frequency in each of the buckets or bins in which we group the data. Here we can see how heights can be grouped into buckets and how many people are part of that bucket. For example, there are 10 people that have a height between 170 and 174.8 cm.

Column Histogram Visualization