Data Visualisation in Python

Thijs Smolders
Department of Chemistry - Ångström

Data Visualisation 101

- Data visualization plays a pivotal role in aiding chemical intuition and understanding complex scientific data. It allows researchers and chemists to visually explore and analyze data, uncover patterns, and gain valuable insights. Effective visualizations help communicate scientific findings, accelerate the discovery process, and support decision-making in various areas of chemistry.

- In Python, there are powerful libraries available for data visualization that are particularly useful in a scientific context. Matplotlib, Seaborn, and Plotly offer versatile tools to create visually appealing and informative plots.

Data visualisation examples

1. Molecular Structures: Visualize 3D molecular structures to gain insights into the geometry, symmetry, and arrangement of atoms. Tools like PyMOL and RDKit can be used to render high-quality molecular visualizations.
2. Spectroscopy Data: Plot absorption, emission, or NMR spectra to analyze and interpret spectral data. Visualizations help identify peak positions, intensities, and patterns associated with molecular transitions or structural features.
3. Reaction Pathways: Plot potential energy surfaces, reaction coordinate diagrams, or free energy profiles to visualize energy changes during chemical reactions. These visualizations aid in understanding reaction mechanisms and identifying key intermediates and transition states.
4. Crystal Structures: Display crystal structures using tools like CrystalMaker or ASE. Visualize unit cells, lattice parameters, and atomic arrangements to analyze crystallographic data and study crystal properties.
5. Data Distribution: Use histograms, box plots, or violin plots to visualize distributions of chemical properties such as molecular weight, boiling point, or solubility. These visualizations help identify trends, outliers, and relationships within datasets.

Plotting using matplotlib

Using the matplotlib library

Matplotlib is a widely used data visualization library in Python that was originally created by John D. Hunter in 2003 as a tool for plotting 2D graphics. It was inspired by the plotting capabilities of MATLAB and aimed to provide a similar interface for generating high-quality plots in Python. Over the years, Matplotlib has grown to become a versatile library capable of creating a wide range of visualizations.

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

Matplotlib pros

Pros of Matplotlib:

1. Wide Range of Plots: Matplotlib offers a comprehensive set of plotting functions, allowing you to create various types of plots, including line plots, scatter plots, bar plots, histograms, and more.
2. Customization Options: Matplotlib provides extensive customization options to tailor your plots to specific needs. You can control colors, line styles, markers, labels, titles, axes, and more.
3. Publication-Quality Output: Matplotlib enables the creation of high-quality plots suitable for publication or presentation purposes. You can save your plots in various file formats, including PNG, PDF, SVG, and more.
4. Integration with Python Ecosystem: Matplotlib seamlessly integrates with other Python libraries and frameworks, such as NumPy, Pandas, and SciPy, making it a powerful tool for visualizing data in scientific computing and data analysis workflows.

Matplotlib cons

Cons of Matplotlib:

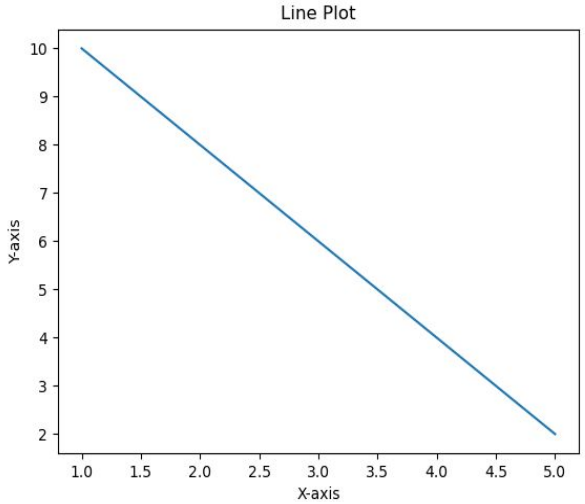
1. Steep Learning Curve: Matplotlib has a relatively steep learning curve, especially for beginners. Understanding its syntax and customization options may require some time and practice.
2. Default Aesthetics: The default plot aesthetics in Matplotlib may not always produce visually appealing plots. You may need to invest time in tweaking and customizing your plots to achieve the desired look and feel.
3. Lack of Interactivity: Matplotlib primarily focuses on static plotting and lacks built-in interactivity features compared to some other visualization libraries like Plotly or Bokeh.

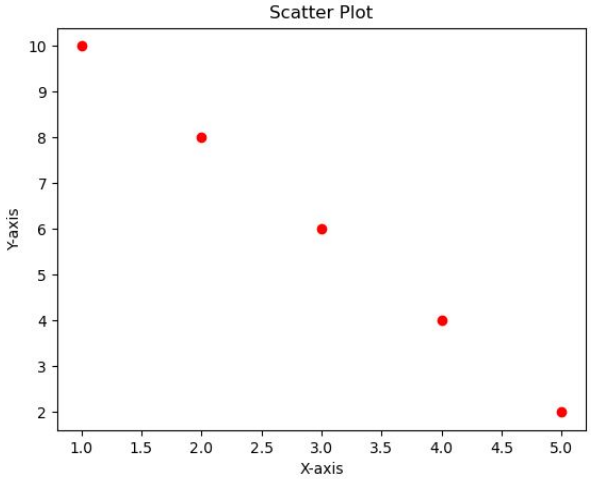
Matplotlib tips

Tips for Using Matplotlib:

1. Import Matplotlib: Start by importing the Matplotlib library using the import matplotlib.pyplot as plt statement.
2. Create Figures and Axes: Use the plt.figure() function to create a figure and plt.axes() to create an axes object.
3. Plotting Data: Utilize the various plotting functions available in Matplotlib, such as plt.plot(), plt.scatter(), plt.bar(), etc., to visualize your data.
4. Customize Your Plot: Take advantage of the customization options in Matplotlib, including labels, titles, colors, line styles, markers, grid lines, and more.
5. Show and Save Plots: Use plt.show() to display the plot on the screen and plt.savefig() to save the plot to a file in your desired format.

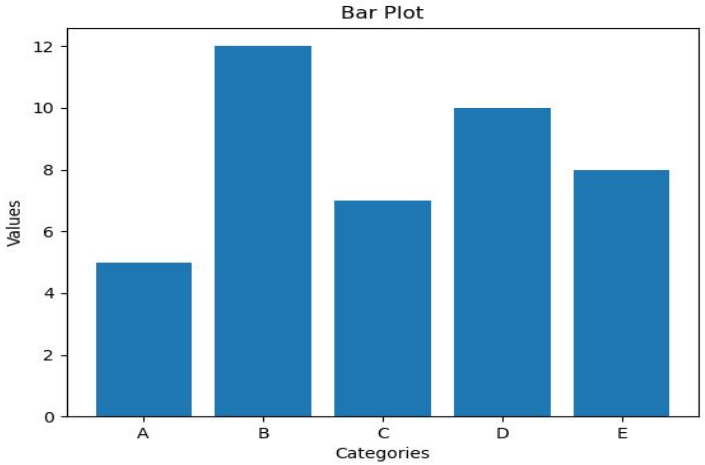
```python
import matplotlib.pyplot as plt

# Line plot
x = [1, 2, 3, 4, 5]
y = [10, 8, 6, 4, 2]
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()

# Scatter plot
x = [1, 2, 3, 4, 5]
y = [10, 8, 6, 4, 2]
plt.scatter(x, y, color='red', marker='o')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```



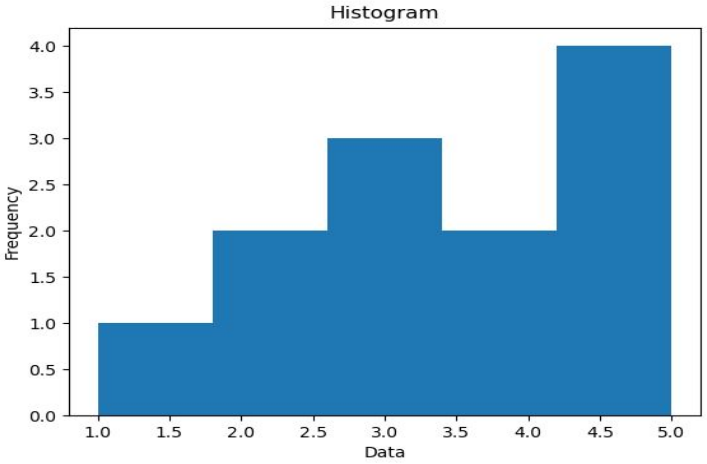


UPPSALA UNIVERSITET

```python
# Bar plot
categories = ['A', 'B', 'C', 'D', 'E']
values = [5, 12, 7, 10, 8]
plt.bar(categories, values)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```




```python
# Histogram
data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]
plt.hist(data, bins=5)
plt.xlabel('Data')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```



Creating multiple axes and subplots

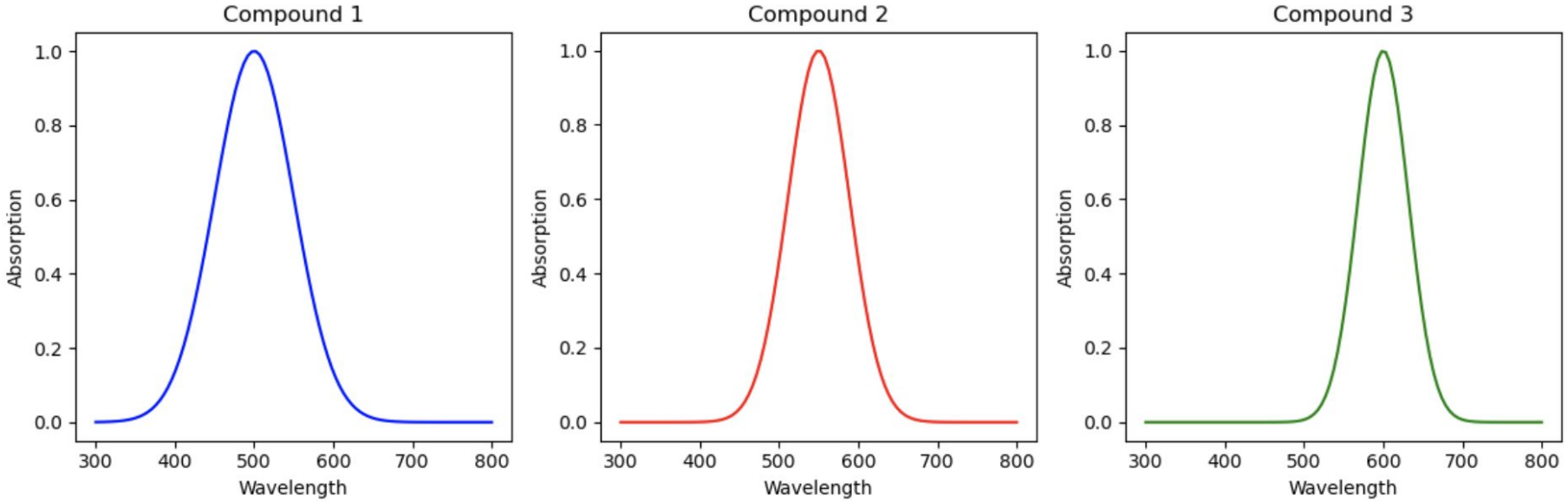
In chemistry research, it is often necessary to compare and analyze multiple datasets side by side to gain insights and draw meaningful conclusions. Matplotlib provides a powerful feature called subplots, which allows us to create multiple axes within a single figure. Subplots enable us to visualize different aspects of the data simultaneously, facilitating comparisons and explorations. In this example, we will demonstrate how to use subplots in Matplotlib to compare the absorption spectra of different chemical compounds.

Creating multiple axes and subplots

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

# Simulated absorption spectra for three chemical compounds
wavelengths = np.linspace(300, 800, 100)
compound1_absorption = np.exp(-(wavelengths - 500) ** 2 / 5000)
compound2_absorption = np.exp(-(wavelengths - 550) ** 2 / 3000)
compound3_absorption = np.exp(-(wavelengths - 600) ** 2 / 2000)

# Creating subplots
fig, axs = plt.subplots(1, 3, figsize=(12, 4))

# Plotting absorption spectra on individual axes
axs[0].plot(wavelengths, compound1_absorption, color='blue')
axs[0].set_title('Compound 1')
axs[0].set_xlabel('Wavelength')
axs[0].set_ylabel('Absorption')

axs[1].plot(wavelengths,
compound2_absorption, color='red')
axs[1].set_title('Compound 2')
axs[1].set_xlabel('Wavelength')
axs[1].set_ylabel('Absorption')

axs[2].plot(wavelengths,
compound3_absorption, color='green')
axs[2].set_title('Compound 3')
axs[2].set_xlabel('Wavelength')
axs[2].set_ylabel('Absorption')

# Adjusting spacing between subplots
plt.tight_layout()

# Displaying the plot
plt.show()
```

UPPSALA UNIVERSITET

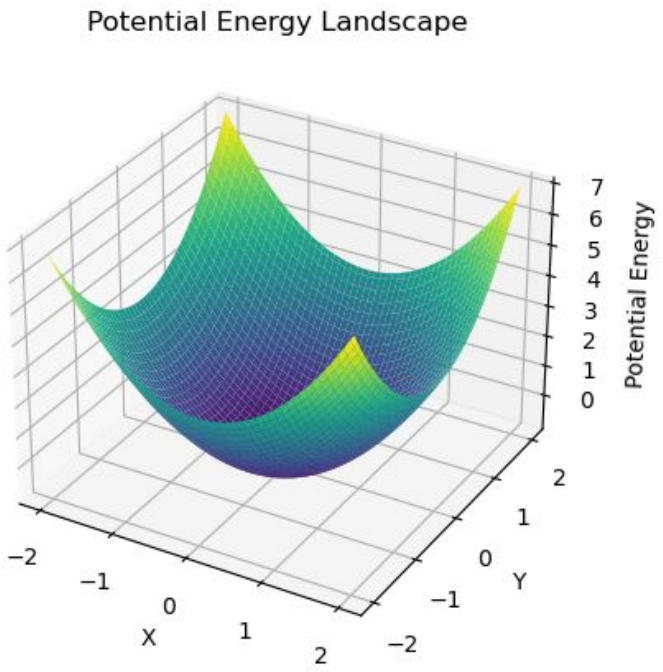
Creating multiple axes and subplots



3D plotting

In chemistry, potential energy landscapes play a crucial role in understanding reaction pathways and the stability of molecules. Matplotlib provides powerful tools for creating 3D plots, allowing us to visualize and analyze the potential energy surfaces of chemical systems. In this example, we will demonstrate how to use Matplotlib to plot a 3D representation of a potential energy landscape, providing insights into the energetics and behavior of a chemical reaction or molecular system.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate x, y coordinates
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)

# Calculate potential energy values
Z = -1 + X**2 + 2*Y**2

# Creating a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plotting the potential energy landscape
ax.plot_surface(X, Y, Z, cmap='viridis')

# Adding labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Potential Energy')
ax.set_title('Potential Energy Landscape')

# Displaying the plot
plt.show()
```



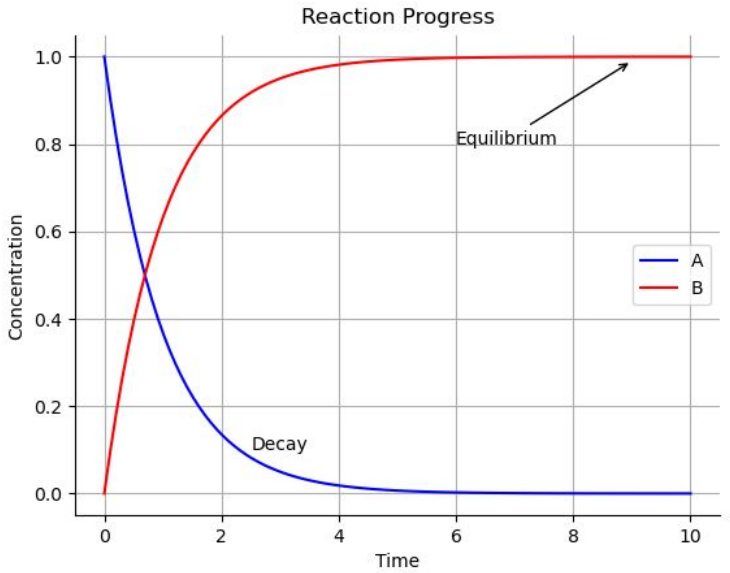
UPPSALA UNIVERSITET

Plot customization

In chemistry, visualizing data in a clear and visually appealing way is essential for effectively communicating scientific findings. Matplotlib offers a wide range of advanced customization options that allow you to tailor your plots to meet specific requirements and enhance their visual impact. In this example, we will demonstrate how to use Matplotlib's advanced customization features to create a visually appealing plot of a chemical reaction, incorporating custom colors, annotations, and a visual style to highlight important features and improve the overall presentation.

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

# Generate sample data
time = np.linspace(0, 10, 100)
concentration_A = np.exp(-time)
concentration_B = 1 - concentration_A

# Create a figure and axis objects
fig, ax = plt.subplots()

# Plot the data
ax.plot(time, concentration_A, color='blue',
     label='A')
ax.plot(time, concentration_B, color='red',
     label='B')

# Customize the plot
ax.set_xlabel('Time')
ax.set_ylabel('Concentration')
ax.set_title('Reaction Progress')
ax.legend()
ax.grid(True)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

```python
# Add annotations and text
ax.annotate('Equilibrium', xy=(5, 0.5), xytext=(9, 0.99),
            arrowprops=dict(facecolor='black', arrowstyle='->'))
ax.text(2, 0.2, 'Decay', fontsize=10, ha='center')

# Apply a visual style
plt.style.use('ggplot')

# Display the plot
plt.show()
```

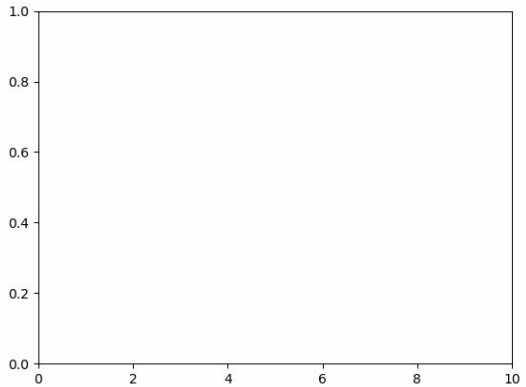


Animations

In chemistry, animations and dynamic visualizations can be powerful tools for studying dynamic processes, such as molecular motions, reaction kinetics, or phase transitions. Matplotlib provides functionality to create animated plots and interactive visualizations that can help in understanding these dynamic phenomena. In this example, we will demonstrate how to utilize Matplotlib's animation capabilities to create an animated visualization of a chemical reaction, allowing us to observe the changes in reactant concentrations over time.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Generate sample data
time = np.linspace(0, 10, 100)
concentration_A = np.exp(-time)
concentration_B = 1 - concentration_A

# Create a figure and axis objects
fig, ax = plt.subplots()

# Create initial empty plot
line, = ax.plot([], [], lw=2)

# Set plot limits
ax.set_xlim(0, 10)
ax.set_ylim(0, 1)

# Define initialization function
def init():
    line.set_data([], [])
    return line,
```

```python
# Define animation update function
def update(frame):
    line.set_data(time[:frame], concentration_A[:frame])
    return line,

# Create animation
animation = FuncAnimation(fig, update, frames=len(time),
init_func=init, blit=True)

# Save the animation as a GIF
animation.save('output.gif', writer='imagemagick')
```



UPPSALA UNIVERSITET

Matplotlib for beginners

Matplotlib is a library for making 2D plots in Python. It is designed with the philosophy that you should be able to create simple plots with just a few commands:

1 Initialize

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

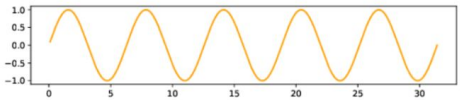
2 Prepare

```
X = np.linspace(0, 4*np.pi, 1000)
Y = np.sin(X)
```

3 Render

```
fig, ax = plt.subplots()
ax.plot(X, Y)
fig.show()
```

4 Observe



Choose

Matplotlib offers several kind of plots (see Gallery):

```
X = np.random.uniform(0, 1, 100)
Y = np.random.uniform(0, 1, 100)
ax.scatter(X, Y)
```

```
X = np.arange(10)
Y = np.random.uniform(1, 10, 10)
ax.bar(X, Y)
```

```
Z = np.random.uniform(0, 1, (8,8))

ax.imshow(Z)
```

```
Z = np.random.uniform(0, 1, (8,8))

ax.contourf(Z)
```

```
Z = np.random.uniform(0, 1, 4)

ax.pie(Z)
```

```
Z = np.random.normal(0, 1, 100)

ax.hist(Z)
```

```
X = np.arange(5)
Y = np.random.uniform(0, 1, 5)
ax.errorbar(X, Y, Y/4)
```

```
Z = np.random.normal(0, 1, (100,3))

ax.boxplot(Z)
```

Tweak

You can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, color="black")
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linestyle="--")
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linewidth=5)
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, marker="o")
```

Organize

You can plot several data on the the same figure, but you can also split a figure in several subplots (named *Axes*):

```
X = np.linspace(0, 10, 100)
Y1, Y2 = np.sin(X), np.cos(X)
ax.plot(X, Y1, X, Y2)
```

```
fig, (ax1, ax2) = plt.subplots(2,1)
ax1.plot(X, Y1, color="C1")
ax2.plot(X, Y2, color="C0")
```

```
fig, (ax1, ax2) = plt.subplots(1,2)
ax1.plot(Y1, X, color="C1")
ax2.plot(Y2, X, color="C0")
```

Label (everything)

```
ax.plot(X, Y)
fig.suptitle(None)
ax.set_title("A Sine wave")
```


```
ax.plot(X, Y)
ax.set_ylabel(None)
ax.set_xlabel("Time")
```

Explore

Figures are shown with a graphical user interface that allows to zoom and pan the figure, to navigate between the different views and to show the value under the mouse.

Save (bitmap or vector format)

```
fig.savefig("my-first-figure.png", dpi=300)
fig.savefig("my-first-figure.pdf")
```

Plotting using Seaborn

Using the seaborn library

Seaborn is a powerful Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating visually appealing and informative statistical graphics. Seaborn simplifies the process of creating complex plots by providing a wide range of pre-defined styles and color palettes. It seamlessly integrates with NumPy and Pandas, making it a popular choice for data exploration and analysis. Compared to Matplotlib, Seaborn offers enhanced aesthetics, improved default settings, and built-in statistical plotting capabilities. With Seaborn, you can effortlessly create visually stunning plots to gain insights from your data and effectively communicate your findings.

```
import seaborn as sns
```

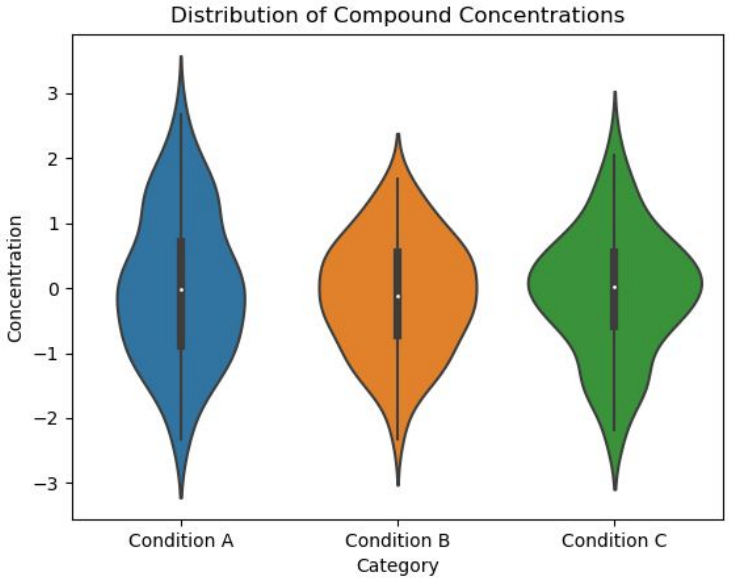
Seaborn pros

1. Enhanced Aesthetics: Seaborn provides visually appealing default styles and color palettes, making it easier to create professional-looking plots without much customization.
2. Statistical Plotting: Seaborn offers built-in statistical plotting functions that allow for the visualization of complex statistical relationships and patterns in the data.
3. Simplified Plotting: Seaborn simplifies the process of creating complex plots by providing high-level functions that abstract away some of the low-level details and boilerplate code required in Matplotlib.
4. Integration with Pandas: Seaborn seamlessly integrates with Pandas, allowing for easy visualization of data stored in DataFrame objects, making it a powerful tool for exploratory data analysis.
5. Convenient Customization: Seaborn provides convenient methods for customizing plots, such as adjusting color palettes, changing plot aesthetics, and adding informative annotations.

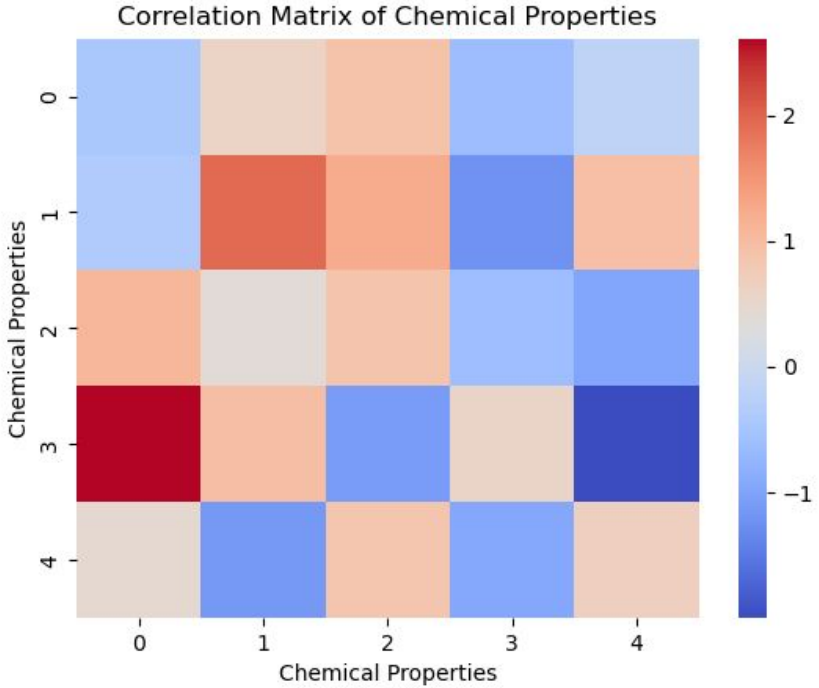
Seaborn cons

1. Limited Plot Types: Seaborn has a narrower scope compared to Matplotlib and may not offer the same level of flexibility when it comes to creating highly customized or specialized plots.
2. Matplotlib Dependence: While Seaborn is built on top of Matplotlib, it still requires some understanding of matplotlib's syntax and concepts to make advanced customizations.
3. Learning Curve: Although Seaborn simplifies the plot creation process, mastering its advanced features and functionalities may still require some learning and practice.
4. Less Control: Seaborn's high-level interface may limit the fine-grained control that Matplotlib provides, making it less suitable for highly customized or unconventional plot designs.
5. Community Size: While Seaborn has a growing user community, it may not have the same level of extensive resources, online examples, and community support as Matplotlib due to its more recent development.

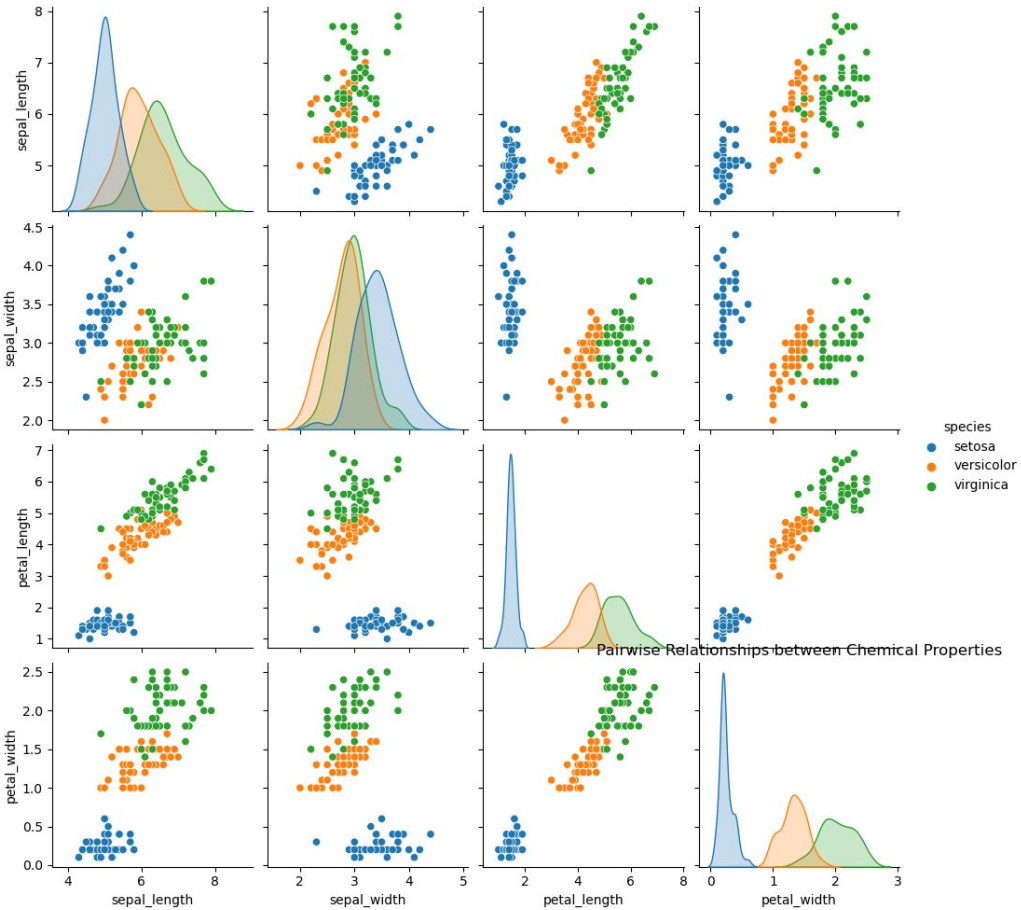
```python
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Sample dataset of chemical compound concentrations
across different categories
categories = ['Condition A', 'Condition B', 'Condition C']
data = {
    'Category': categories * 100,
    'Concentration': np.random.randn(300)
}

# Create a violin plot
sns.violinplot(x='Category', y='Concentration', data=data)

# Add labels and title
plt.xlabel('Category')
plt.ylabel('Concentration')
plt.title('Distribution of Compound Concentrations')

# Display the plot
plt.show()
```



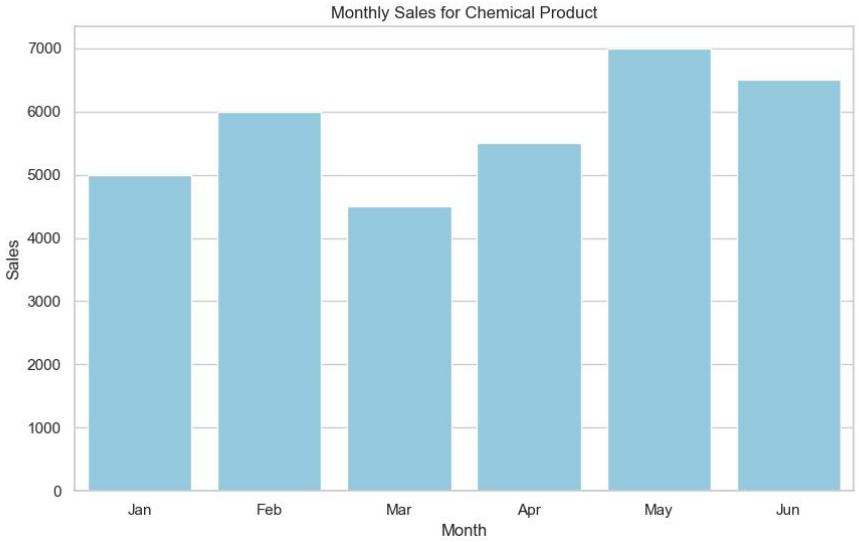
```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Sample dataset of correlation matrix
correlation_matrix = np.random.randn(5, 5)

# Create a heatmap
sns.heatmap(correlation_matrix,
cmap='coolwarm')

# Add labels and title
plt.xlabel('Chemical Properties')
plt.ylabel('Chemical Properties')
plt.title('Correlation Matrix of Chemical
Properties')

# Display the plot
fig = plt.gcf()
fig.savefig(plot_folder + "sns_heatmap.png")
plt.show()
```



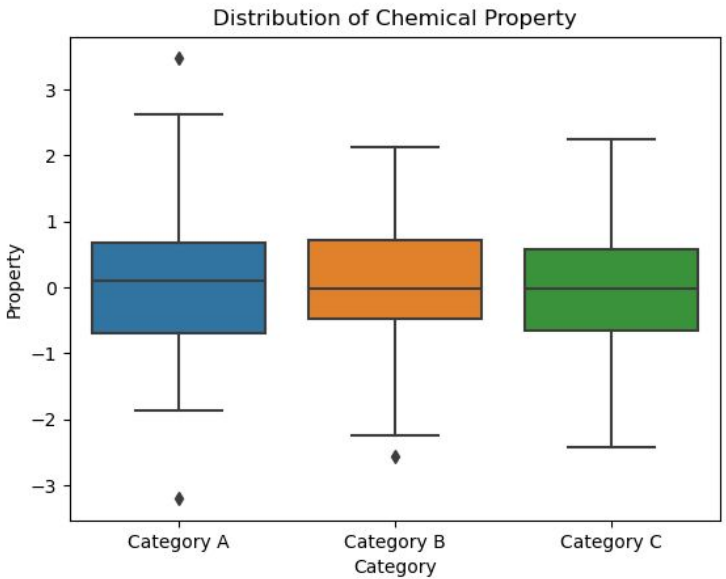
```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Sample dataset of chemical
properties
data = sns.load_dataset('iris')

# Create a pair plot
sns.pairplot(data, hue='species')

# Add labels and title
plt.title('Pairwise Relationships
between Chemical Properties')

# Display the plot
fig = plt.gcf()
fig.savefig(plot_folder +
"sns_heatmap.png")
plt.show()
```



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

# Sample time series data
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
    'Sales': [5000, 6000, 4500, 5500, 7000, 6500]
}

# Convert data to pandas DataFrame
df = pd.DataFrame(data)

# Set the figure size
plt.figure(figsize=(10, 6))

# Bar plot for monthly sales
sns.barplot(x='Month', y='Sales', data=df, color='skyblue')

# Set labels and title
plt.xlabel('Month')
plt.ylabel('Sales')
plt.title('Monthly Sales for Chemical Product')

# Display the plot
plt.show()
```



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

# Sample dataset of chemical compound properties
categories = ['Category A', 'Category B', 'Category C']
data = {
    'Category': categories * 100,
    'Property': np.random.randn(300)
}

# Create a box plot
sns.boxplot(x='Category', y='Property', data=data)

# Add labels and title
plt.xlabel('Category')
plt.ylabel('Property')
plt.title('Distribution of Chemical Property')

# Display the plot
fig = plt.gcf()
fig.savefig(plot_folder + "box_plot.png")
plt.show()
```



datacamp

Python For Data Science
Seaborn Cheat Sheet

Learn Seaborn online at www.DataCamp.com

Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on matplotlib and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

1. Prepare some data
2. Control figure aesthetics
3. Plot with Seaborn
4. Further customize your plot
5. Show your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips") #Step 1
>>> sns.set_style("whitegrid") #Step 2
>>> g = sns.lmplot(x="tip", #Step 3
                   y="total_bill",
                   data=tips,
                   aspect=2)
>>> g = (g.set_axis_labels("Tip","Total bill(USD)").
set(xlim=(0,10),ylim=(0,100)))
>>> plt.title("title") #Step 4
>>> plt.show(g) #Step 5
```

1 Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101),
                         'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

Also see Matplotlib

```
>>> f, ax = plt.subplots(figsize=(5,6)) #Create a figure and one subplot
```

Seaborn styles

```
>>> sns.set() #(Re)set the seaborn default
>>> sns.set_style("whitegrid") #Set the matplotlib parameters
>>> sns.set_style("ticks", #Set the matplotlib parameters
                  {"xtick.major.size":8,
                   "ytick.major.size":8})
#Return a dict of params or use with with to temporarily set the style
>>> sns.axes_style("whitegrid")
```

Context Functions

```
>>> sns.set_context("talk") #Set context to "talk"
>>> sns.set_context("notebook", #Set context to "notebook",
                    font_scale=1.5, #Scale font elements and
                    rc={"lines.linewidth":2.5}) #override param mapping
```

Color Palette

```
>>> sns.set_palette("husl",3) #Define the color palette
>>> sns.color_palette("husl") #Use with with to temporarily set palette
>>> flatui = ["#9b59b6","#3498db","#95a5a6","#e74c3c","#34495e","#2ecc71"]
>>> sns.set_palette(flatui) #Set your own color palette
```

3 Plotting With Seaborn

Axis Grids

```
>>> g = sns.FacetGrid(titanic, #Subplot grid for plotting conditional relationships
                       col="survived",
                       row="sex")
>>> g = g.map(plt.hist,"age")
>>> sns.factorplot(x="pclass", #Draw a categorical plot onto a Facetgrid
                   y="survived",
                   hue="sex",
                   data=titanic)
>>> sns.lmplot(x="sepal_width", #Plot data and regression model fits across a FacetGrid
               y="sepal_length",
               hue="species",
               data=iris)
>>> h = sns.PairGrid(iris) #Subplot grid for plotting pairwise relationships
>>> h = h.map(plt.scatter)
>>> sns.pairplot(iris) #Plot pairwise bivariate distributions
>>> i = sns.JointGrid(x="x", #Grid for bivariate plot with marginal univariate plots
                      y="y",
                      data=data)
>>> i = i.plot(sns.regplot,
               sns.distplot)
>>> sns.jointplot("sepal_length", #Plot bivariate distribution
                  "sepal_width",
                  data=iris,
                  kind='kde')
```

4 Further Customizations

Also see Matplotlib

Axisgrid Objects

```
>>> g.despine(left=True) #Remove left spine
>>> g.set_ylabels("Survived") #Set the labels of the y-axis
>>> g.set_xticklabels(rotation=45) #Set the tick labels for x
>>> g.set_axis_labels("Survived", #Set the axis labels
                      "Sex")
>>> h.set(xlim=(0,5), #Set the limit and ticks of the x-and y-axis
          ylim=(0,5),
          xticks=[0,2.5,5],
          yticks=[0,2.5,5])
```

Plot

```
>>> plt.title("A Title") #Add plot title
>>> plt.ylabel("Survived") #Adjust the label of the y-axis
>>> plt.xlabel("Sex") #Adjust the label of the x-axis
>>> plt.ylim(0,100) #Adjust the limits of the y-axis
>>> plt.xlim(0,10) #Adjust the limits of the x-axis
>>> plt.setp(ax,yticks=[0,5]) #Adjust a plot property
>>> plt.tight_layout() #Adjust subplot params
```

Regression Plots

```
>>> sns.regplot(x="sepal_width", #Plot data and a linear regression model fit
                y="sepal_length",
                data=iris,
                ax=ax)
```

Distribution Plots

```
>>> plot = sns.distplot(data.y, #Plot univariate distribution
                        kde=False,
                        color="b")
```

Matrix Plots

```
>>> sns.heatmap(uniform_data,vmin=0,vmax=1) #Heatmap
```

Categorical Plots

Scatterplot

```
>>> sns.stripplot(x="species", #Scatterplot with one categorical variable
                  y="petal_length",
                  data=iris)
>>> sns.swarmplot(x="species", #Categorical scatterplot with non-overlapping points
                  y="petal_length",
                  data=iris)
```

Bar Chart

```
>>> sns.barplot(x="sex", #Show point estimates & confidence intervals with scatterplot glyphs
                y="survived",
                hue="class",
                data=titanic)
```

Count Plot

```
>>> sns.countplot(x="deck", #Show count of observations
                  data=titanic,
                  palette="Greens_d")
```

Point Plot

```
>>> sns.pointplot(x="class", #Show point estimates & confidence intervals os rectangular bars
                  y="survived",
                  hue="sex",
                  data=titanic,
                  palette={"male":"g",
                  "female":"m"},
                  markers=["^","o"],
                  linestyles=["-","--"])
```

Boxplot

```
>>> sns.boxplot(x="alive", #Boxplot
                y="age",
                hue="adult_male",
                data=titanic)
>>> sns.boxplot(data=iris,orient="h") #Boxplot with wide-form data
```

Violinplot

```
>>> sns.violinplot(x="age", #Violin plot
                   y="sex",
                   hue="survived",
                   data=titanic)
```

5 Show or Save Plot

Also see Matplotlib

```
>>> plt.show() #Show the plot
>>> plt.savefig("foo.png") #Save the plot as a figure
>>> plt.savefig("foo.png", #Save transparent figure
                transparent=True)
```

> Close & Clear

Also see Matplotlib

```
>>> plt.cla() #Clear an axis
>>> plt.clf() #Clear an entire figure
>>> plt.close() #Close a window
```

Learn Data Skills Online at www.DataCamp.com

Plotting using plotly

Using the plotly library

The plotly Python library is an interactive, open-source plotting library that supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases.
Built on top of the Plotly JavaScript library (plotly.js), plotly enables Python users to create beautiful interactive web-based visualizations that can be displayed in Jupyter notebooks, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash. The plotly Python library is sometimes referred to as "plotly.py" to differentiate it from the JavaScript library.

Plotly provides a high-level interface to create visually appealing and interactive plots, making it suitable for both exploratory analysis and production-level visualizations. Plotly supports various chart types, including line plots, scatter plots, bar charts, pie charts, and more.

conda install -c plotly plotly

import plotly

UPPSALA UNIVERSITET

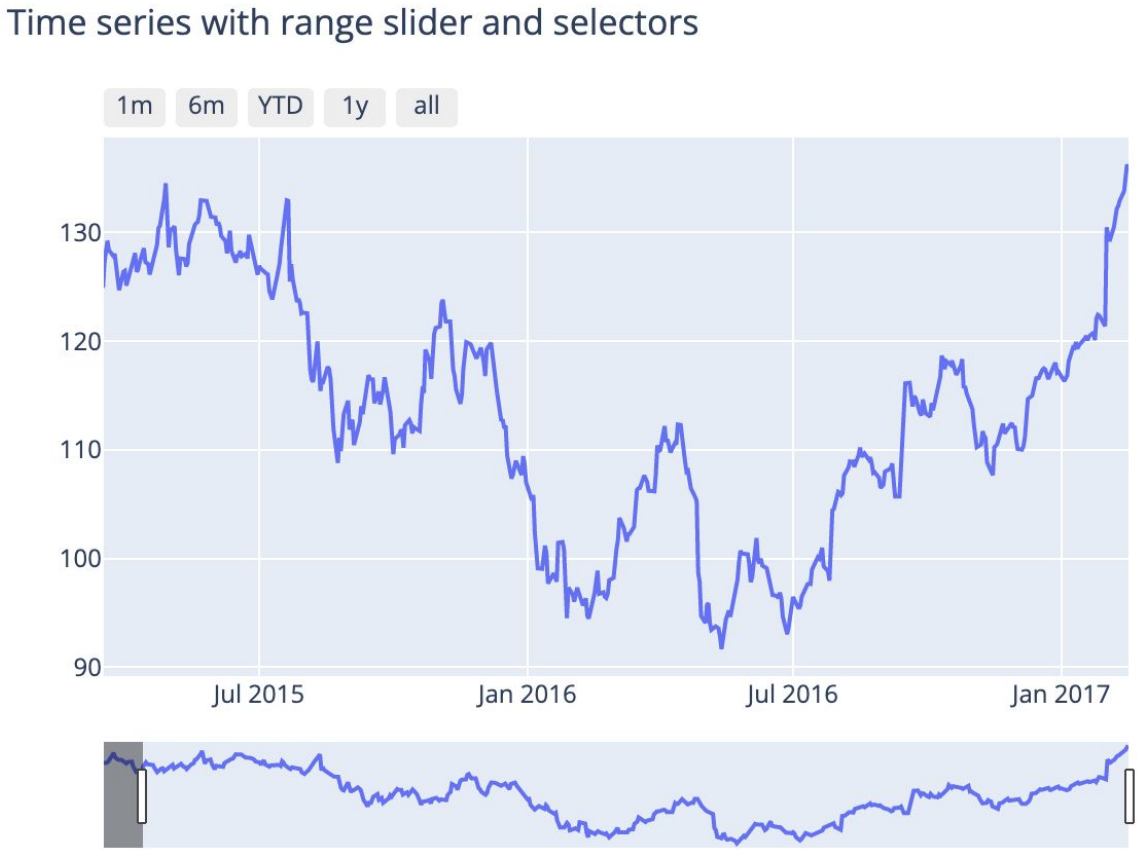
Pros of plotly

Benefits of Plotly:

1. Interactivity: Plotly allows users to create interactive plots with customizable features, providing a rich and engaging visual experience. Users can zoom in/out, pan across the plot, and hover over data points to see additional information.
2. Web-based: Plotly generates plots that can be easily shared and embedded in web applications or dashboards. The plots can be viewed and interacted with using modern web browsers, making it convenient for collaboration and sharing visualizations online.
3. Versatility: Plotly supports a wide range of chart types and visualizations, enabling users to create complex plots with multiple traces, subplots, and annotations. It also provides tools for 3D plotting, geospatial visualizations, and statistical graphics.

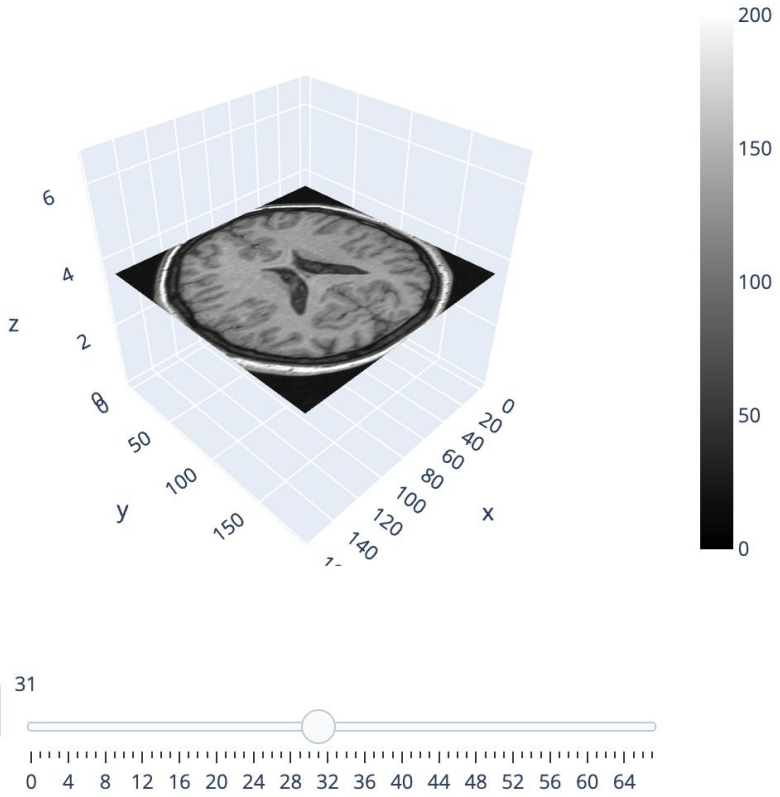
Cons of plotly

1. Learning Curve: Plotly has a steeper learning curve compared to Matplotlib, especially for users who are not familiar with web technologies or JavaScript. The syntax and structure of Plotly can be more complex, requiring additional time and effort to master.
2. Limited Customization: While Plotly offers a range of customization options, it may not provide the same level of fine-grained control over plot appearance as Matplotlib. Some advanced customization features in Matplotlib may not be available or require more complex configuration in Plotly.



UPPSALA UNIVERSITET

Slices in volumetric data



datacamp plotly

Data Visualization with Plotly Express in Python

Learn Plotly online at www.DataCamp.com

> What is plotly?

Plotly Express is a high-level data visualization package that allows you to create interactive plots with very little code. It is built on top of Plotly Graph Objects, which provides a lower-level interface for developing custom visualizations.

> Interactive controls in Plotly

Plotly plots have interactive controls shown in the top-right of the plot. The controls allow you to do the following:

- **Download plot as a png:** Save your interactive plot as a static PNG.
- **Zoom:** Zoom in on a region of interest in the plot.
- **Pan:** Move around in the plot.
- **Box Select:** Select a rectangular region of the plot to be highlighted.
- **Lasso Select:** Draw a region of the plot to be highlighted.
- **Autoscale:** Zoom to a "best" scale.
- **Reset axes:** Return the plot to its original state.
- **Toggle Spike Lines:** Show or hide lines to the axes whenever you hover over data.
- **Show closest data on hover:** Show details for the nearest data point to the cursor.
- **Compare data on hover:** Show the nearest data point to the x-coordinate of the cursor.

> Plotly Express code pattern

The code pattern for creating plots is to call the plotting function, passing a data frame as the first argument. The x argument is a string naming the column to be used on the x-axis. The y argument can either be a string or a list of strings naming column(s) to be used on the y-axis.

```
px.plotting_fn(dataframe, # Dataframe being visualized
               x=["column-for-x-axis"], # Accepts a string or a list of strings
               y=["columns-for-y-axis"], # Accepts a string or a list of strings
               title="Overall plot title", # Accepts a string
               xaxis_title="X-axis title", # Accepts a string
               yaxis_title="Y-axis title", # Accepts a string
               width=width_in_pixels, # Accepts an integer
               height=height_in_pixels) # Accepts an integer
```

> Common plot types

Import plotly

```
# import plotly express as px
import plotly.express as px
```

Scatter plots

```
# Create a scatterplot on a DataFrame named clinical_data
px.scatter(clinical_data, x="experiment_1", y="experiment_2")
```

Set the `size` argument to the name of a numeric column to control the size of the points and create a bubble plot.

Line plots

```
# Create a lineplot on a DataFramed named stock_data
px.line(stock_data, x="date", y=["FB", "AMZN"])
```

Set the `line_dash` argument to the name of a categorical column to have dashes or dots for different lines.

Bar plots

```
# Create a barplot on a DataFramed named commodity_data
px.bar(commodity_data, x="nation", y=["gold", "silver", "bronze"],
       color_discrete_map={"gold": "yellow",
                           "silver": "grey",
                           "bronze": "brown"})
```

Swap the `x` and `y` arguments to draw horizontal bars.

Histograms

```
# Create a histogram on a DataFramed named bill_data
px.histogram(bill_data, x="total_bill")
```

Set the `nbins` argument to control the number of bins shown in the histogram.

Heatmaps

```
# Create a heatmap on a DataFramed named iris_data
px.imshow(iris_data.corr(numeric_only=True),
          zmin=-1, zmax=1, color_continuous_scale='rdbu')
```

Set the `text_auto` argument to `True` to display text values for each cell.

> Customizing plots in plotly

The code pattern for customizing a plot is to save the figure object returned from the plotting function, call its `.update_traces()` method, then call its `.show()` method to display it.

```
# Create a plot with plotly (can be of any type)
fig = px.some_plotting_function()
# Customize and show it with .update_traces() and .show()
fig.update_traces()
fig.show()
```

Customizing markers in Plotly

When working with visualizations like scatter plots, lineplots, and more, you can customize markers according to certain properties. These include:

- `size`: set the marker size
- `color`: set the marker color
- `opacity`: set the marker transparency
- `line`: set the width and color of a border
- `symbol`: set the shape of the marker

```
# In this example, we're updating a scatter plot named fig_sct
fig_sct.update_traces(marker={"size": 24,
                              "color": "magenta",
                              "opacity": 0.5,
                              "line": {"width": 2, "color": "cyan"},
                              "symbol": "square"})
fig_sct.show()
```

Customizing lines in Plotly

When working with visualizations that contain lines, you can customize them according to certain properties. These include:

- `color`: set the line color
- `dash`: set the dash style ("solid", "dot", "dash", "longdash", "dashdot", "longdashdot")
- `shape`: set how values are connected ("linear", "spline", "hv", "vh", "hvh","vhv")
- `width`: set the line width

```
# In this example, we're updating a scatter plot named fig_ln
fig_ln.update_traces(patch={"line": {"dash": "dot",
                                     "shape": "spline",
                                     "width": 6}})
fig_ln.show()
```

Customizing bars in Plotly

When working with barplots and histograms, you can update the bars themselves according to the following properties:

- `size`: set the marker size
- `color`: set the marker color
- `opacity`: set the marker transparency
- `line`: set the width and color of a border
- `symbol`: set the shape of the marker

```
# In this example, we're updating a scatter plot named fig_bar
fig_bar.update_traces(marker={"color": "magenta",
                              "opacity": 0.5,
                              "line": {"width": 2, "color": "cyan"}})
fig_bar.show()
```

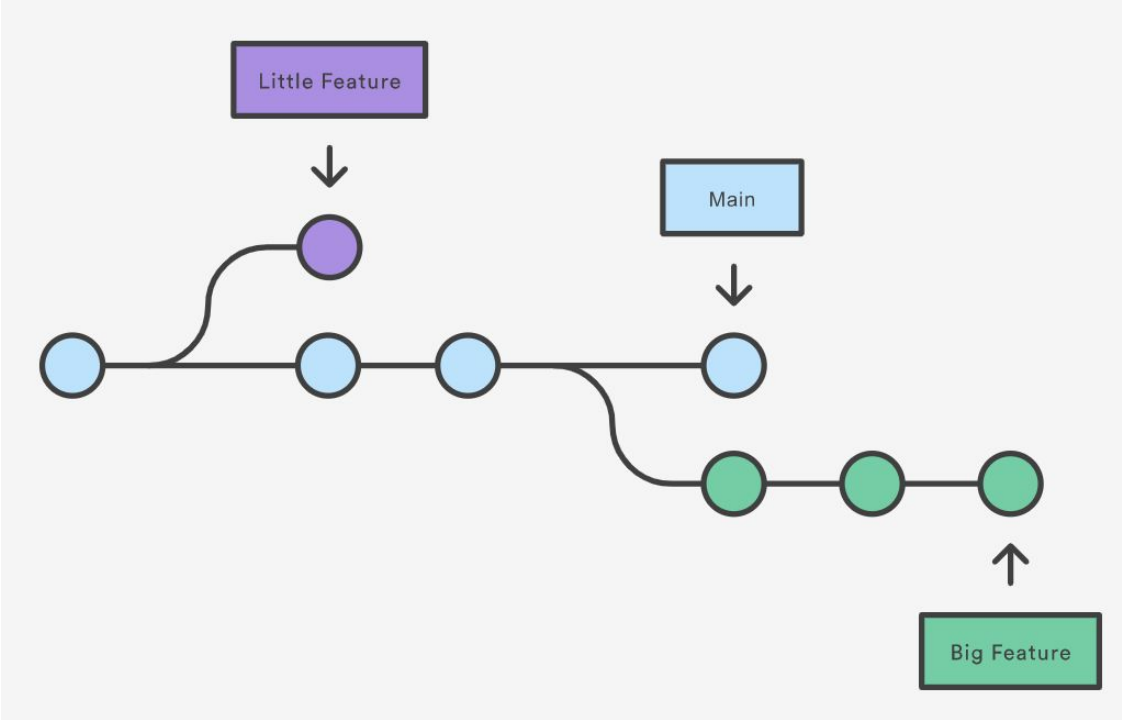
```
# In this example, we're updating a histogram named fig_hst
fig_hst.update_traces(marker={"color": "magenta",
                              "opacity": 0.5,
                              "line": {"width": 2, "color": "cyan"}})
fig_hst.show()
```

Learn Data Skills Online at
www.DataCamp.com
datacamp

Revisiting Git

Git in a nutshell

- Version control
- Revert to older version of code / models / datasets
- Annotate older versions
- Keep central copy (back-up)
- Easily share your work with others
- Limit to filesize on Github

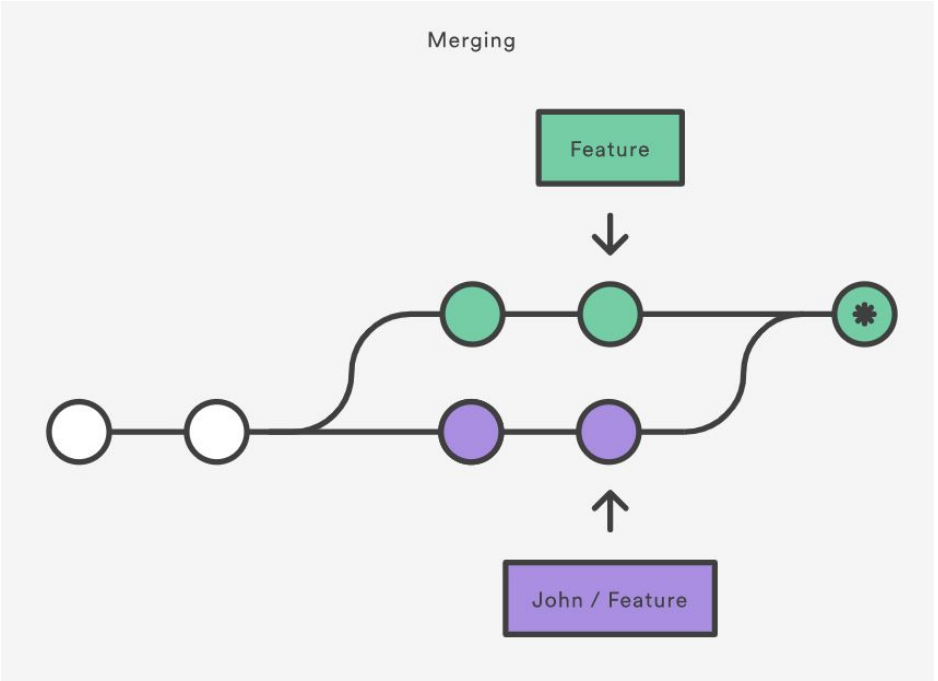


- Figure; https://www.atlassian.com/git/tutorials/using-branches

UPPSALA UNIVERSITET

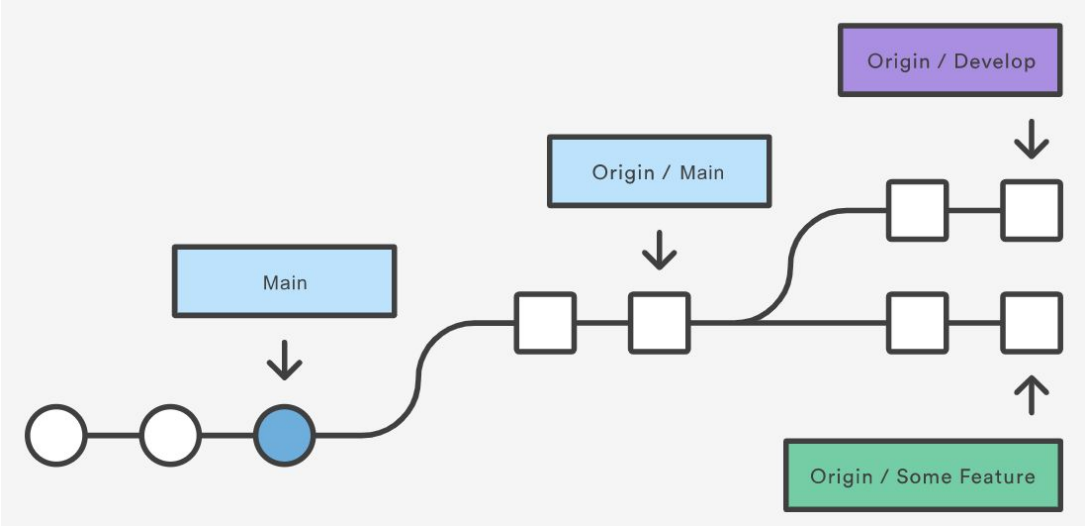
Git merge

- Brings together the code from different branches
- Typically you want to add a new feature that you developed in a separate branch into the main code



- Figure; https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Git fetch

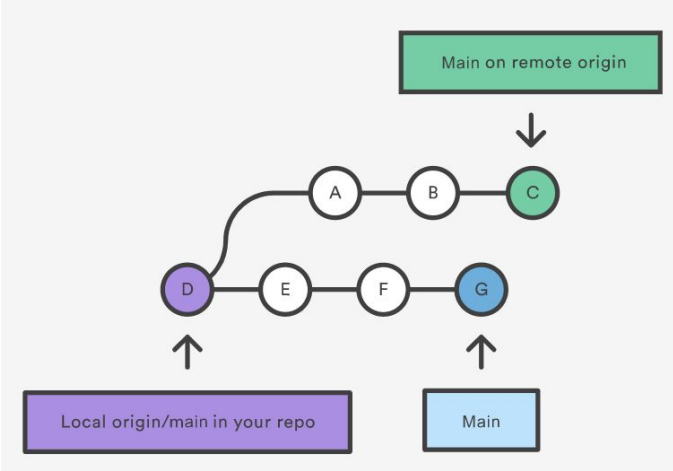


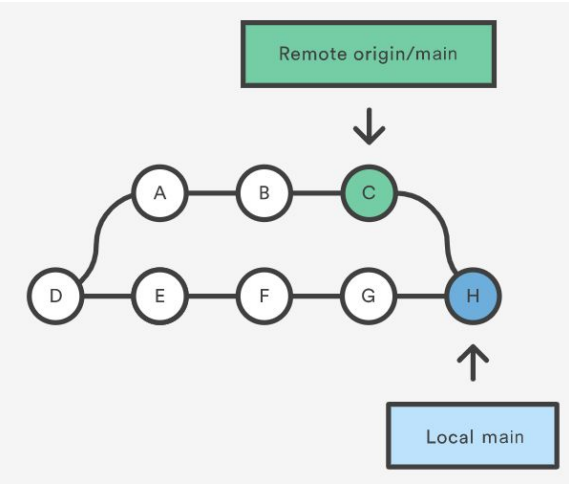
- One way of getting changes from the remote repo (the one in the cloud) to your local copy
- Does not overwrite (merge) your local files unless you actively tell it to do so
- Considered the safer option, compared to git pull, yet requires more manual work to merge the changes

- Figure; https://www.atlassian.com/git/tutorials/syncing/git-fetch

Git pull

- Similar to git fetch, but remote changes will automatically be merged into your local copy (though some options can be set)
- Is quicker than git fetch, but leaves you with less control
- Is fine in many cases though



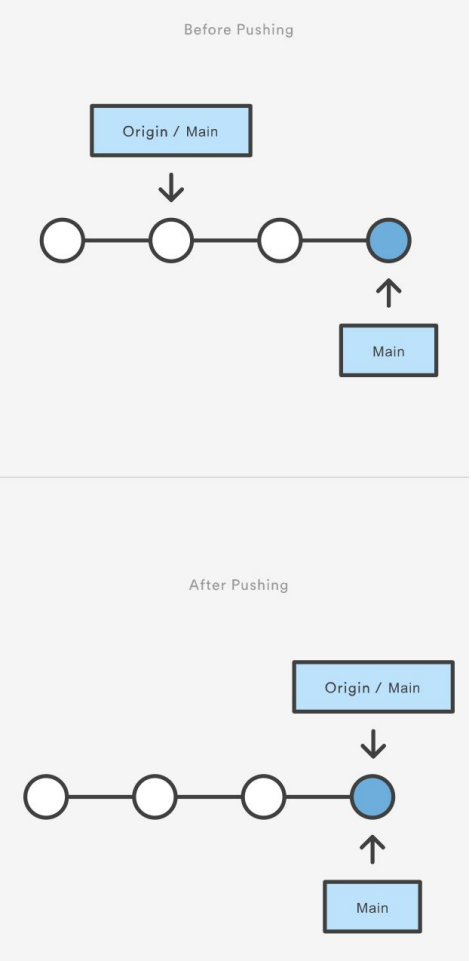


- Figure; https://www.atlassian.com/git/tutorials/syncing/git-pull

Git push

- The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. Remote branches are configured using the git remote command. Pushing has the potential to overwrite changes, caution should be taken when pushing.

- Figure; https://www.atlassian.com/git/tutorials/syncing/git-push



UPPSALA UNIVERSITET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history

`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker

`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository

`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit

`git add [file]`
add a file as it looks now to your next commit (stage)

`git reset [file]`
unstage a file while retaining the changes in working directory

`git diff`
diff of what is changed but not staged

`git diff --staged`
diff of what is staged but not yet committed

`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch

`git branch [branch-name]`
create a new branch at the current commit

`git checkout`
switch to another branch and check it out into your working directory

`git merge [branch]`
merge the specified branch's history into the current one

`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`
show the commit history for the currently active branch

`git log branchB..branchA`
show the commits on branchA that are not on branchB

`git log --follow [file]`
show the commits that changed file, even across renames

`git diff branchB...branchA`
show the diff of what is in branchA that is not in branchB

`git show [SHA]`
show any object in Git in human-readable format

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`
delete the file from project and stage the removal for commit

`git mv [existing-path] [new-path]`
change an existing file path and stage the move

`git log --stat -M`
show all commit logs with indication of any paths that moved

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`
system wide ignore pattern for all local repositories

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`
add a git URL as an alias

`git fetch [alias]`
fetch down all the branches from that Git remote

`git merge [alias]/[branch]`
merge a remote branch into your current branch to bring it up to date

`git push [alias] [branch]`
Transmit local branch commits to the remote repository branch

`git pull`
fetch and merge any commits from the tracking remote branch

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`
apply any commits of current branch ahead of specified one

`git reset --hard [commit]`
clear staging area, rewrite working tree from specified commit

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`
Save modified and staged changes

`git stash list`
list stack-order of stashed file changes

`git stash pop`
write working from top of stash stack

`git stash drop`
discard the changes from top of stash stack