Data Science

Unit-3

Visualization

Visualization in Data Science and Its Applications

Data visualization is a crucial aspect of data science that involves representing complex datasets in a graphical or pictorial format. It helps uncover patterns, trends, and relationships that might not be apparent in raw data. Effective visualization enhances data-driven decision-making across various industries.

Key Aspects of Data Visualization

1. **Types of Data Visualizations:**
    - **Charts & Graphs:** Line charts, bar charts, pie charts, histograms, etc.
    - **Maps:** Geographic visualizations like heatmaps.
    - **Infographics:** Combining data and visuals to tell a story.
    - **Dashboards:** Interactive panels displaying real-time data.
    - **3D Visualizations:** Used in medical imaging, geospatial data, and more.
2. **Tools for Data Visualization:**
    - **Python Libraries:** Matplotlib, Seaborn, Plotly, Bokeh.
    - **R Libraries:** ggplot2, Shiny.
    - **Business Intelligence (BI) Tools:** Tableau, Power BI.
    - **Web-based Tools:** D3.js, Google Data Studio.
3. **Principles of Effective Visualization:**
    - Clarity: Avoid clutter and focus on key insights.
    - Simplicity: Use the right type of visualization for the data.
    - Accuracy: Ensure data representation is not misleading.
    - Interactivity: Use dashboards for dynamic analysis.

Applications of Data Visualization

1. **Business & Finance**
    - **Sales and Revenue Analytics:** Companies use dashboards to track sales trends.
    - **Market Analysis:** Identifying consumer behavior and preferences.
    - **Fraud Detection:** Spotting anomalies in financial transactions.
2. **Healthcare**
    - **Epidemiology:** Tracking disease outbreaks using heatmaps.
    - **Patient Monitoring:** Visualizing vital signs and medical records.
    - **Genomics:** Analyzing genetic data for medical research.
3. **Social Media & Marketing**
    - **Sentiment Analysis:** Understanding customer feedback trends.
    - **Engagement Metrics:** Analyzing likes, shares, and comments.
    - **Ad Campaign Performance:** Measuring ROI and user interactions.
4. **Science & Engineering**
    - **Climate Data Analysis:** Visualizing global temperature trends.
    - **AI and Machine Learning:** Feature importance visualization in models.
    - **Network Analysis:** Understanding complex systems like social networks.
5. **Government & Public Policy**
    - **Elections:** Visualizing polling trends and voting patterns.
    - **Public Safety:** Crime mapping for law enforcement.
    - **Census Data:** Analyzing population growth and migration.

Customizing Plots: Introduction to Matplotlib

Matplotlib is one of the most widely used Python libraries for data visualization. It provides a flexible and powerful framework for creating static, animated, and interactive visualizations. One of its key strengths is the ability to **customize** plots extensively to improve clarity and presentation.

Getting Started with Matplotlib

Installation

If you haven't installed Matplotlib yet, you can do so using:

python
pip install matplotlib

Basic Example

python
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Creating a simple line plot
plt.plot(x, y)

# Display the plot
plt.show()

Customization Techniques

1. Changing Line Styles, Colors, and Markers

Matplotlib allows customization of lines to improve readability.

python
plt.plot(x, y, color='red', linestyle='--', marker='o', linewidth=2, markersize=8)

- **color='red'** → Changes line color
- **linestyle='--'** → Dashed line style ('-' for solid, ':' for dotted)
- **marker='o'** → Circular markers ('s' for squares, '^' for triangles)
- **linewidth=2** → Adjusts line thickness
- **markersize=8** → Adjusts marker size

2. Adding Titles, Labels, and Legends

python
plt.plot(x, y, label="Sales Growth", color='blue')

# Titles and labels
plt.title("Monthly Sales Over Time", fontsize=14, fontweight='bold')

```
plt.xlabel("Month")
plt.ylabel("Sales ($)")

# Adding legend
plt.legend()

plt.show()
```

3. Adjusting Axis Limits and Grid

```
python
plt.plot(x, y, color='green')

# Adjust axis limits
plt.xlim(0, 6)
plt.ylim(5, 45)

# Adding grid lines
plt.grid(True, linestyle='--', alpha=0.7)

plt.show()
```

- **plt.xlim() & plt.ylim()** → Set custom limits for x and y axes.
- **plt.grid(True, linestyle='--', alpha=0.7)** → Adds grid with dashed lines and transparency.

4. Multiple Plots in One Figure (Subplots)

```
python
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

# First subplot
axs[0].plot(x, y, color='red')
axs[0].set_title("Sales Growth")

# Second subplot
axs[1].bar(x, y, color='blue')
axs[1].set_title("Sales Bar Chart")

plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```

- **plt.subplots(rows, columns, figsize=(width, height))** → Creates multiple plots in one figure.
- **axs[0], axs[1]** → Access individual subplots.
- **plt.tight_layout()** → Prevents overlapping elements.

5. Using Different Plot Types

Matplotlib supports various chart types:

```
python
# Bar Chart
plt.bar(x, y, color='orange')

# Scatter Plot
```

```
plt.scatter(x, y, color='purple')

# Histogram
plt.hist([10, 20, 20, 30, 30, 30, 40], bins=4, color='green', edgecolor='black')

plt.show()
```

Plots

some **basic plots** in Matplotlib along with code examples for each:

1. Line Plot

A line plot is the most common and straightforward plot used to show trends over time or continuous data.

python
```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Line plot
plt.plot(x, y, label="Sales Over Time", color='blue', linestyle='-', marker='o')

# Title and Labels
plt.title("Sales Trend")
plt.xlabel("Time (Months)")
plt.ylabel("Sales")

# Show legend
plt.legend()

# Show plot
plt.show()
```

2. Bar Plot

Bar plots are used to compare different groups or categories.

python
```
# Data
categories = ['A', 'B', 'C', 'D']
values = [25, 40, 60, 80]

# Bar plot
plt.bar(categories, values, color='orange')

# Title and Labels
plt.title("Category Comparison")
plt.xlabel("Categories")
plt.ylabel("Values")
```

```
# Show plot
plt.show()
```

3. Scatter Plot

A scatter plot is used to show the relationship between two continuous variables.

```
python
# Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Scatter plot
plt.scatter(x, y, color='red')

# Title and Labels
plt.title("Scatter Plot Example")
plt.xlabel("X Values")
plt.ylabel("Y Values")

# Show plot
plt.show()
```

4. Histogram

Histograms are used to visualize the distribution of a dataset.

```
python
import numpy as np

# Data
data = np.random.randn(1000)  # 1000 random values from a normal distribution

# Histogram
plt.hist(data, bins=30, color='green', edgecolor='black')

# Title and Labels
plt.title("Histogram of Random Data")
plt.xlabel("Value")
plt.ylabel("Frequency")

# Show plot
plt.show()
```

5. Pie Chart

A pie chart is used to show proportions of a whole.

```
python
# Data
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
```

```
# Pie chart
plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140)

# Title
plt.title("Category Distribution")

# Show plot
plt.show()
```

Making subplots

Creating **subplots** allows you to display multiple plots in a single figure, making it easier to compare different datasets or visualize multiple aspects of data in one view. Matplotlib provides a flexible way to create subplots using the plt.subplots() function.

Basic Subplots Example

python

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 30, 40, 50]

# Creating 1x2 grid (1 row, 2 columns)
fig, axs = plt.subplots(1, 2, figsize=(10, 4))

# Plot 1: Line plot
axs[0].plot(x, y, color='blue')
axs[0].set_title("Line Plot")
axs[0].set_xlabel("X")
axs[0].set_ylabel("Y")

# Plot 2: Bar plot
axs[1].bar(x, y, color='green')
axs[1].set_title("Bar Plot")
axs[1].set_xlabel("X")
axs[1].set_ylabel("Y")

# Adjust layout to avoid overlap
plt.tight_layout()

# Show the plots
plt.show()
```

Explanation of Code:

1. **plt.subplots(1, 2, figsize=(10, 4))**: Creates a grid of 1 row and 2 columns of subplots. figsize sets the size of the entire figure.
2. **axs[0] and axs[1]**: axs is a list-like object containing the individual axes (subplots). We access them with indices like an array.
3. **plt.tight_layout()**: Automatically adjusts subplots to fit within the figure area without overlap.

Different Grid Layouts for Subplots

You can create subplots with different grid sizes, such as multiple rows and columns.

2x2 Grid (2 rows, 2 columns)
python
```
# Creating 2x2 grid
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# Plot 1: Line plot
axs[0, 0].plot(x, y, color='blue')
axs[0, 0].set_title("Line Plot")

# Plot 2: Bar plot
axs[0, 1].bar(x, y, color='orange')
axs[0, 1].set_title("Bar Plot")

# Plot 3: Scatter plot
axs[1, 0].scatter(x, y, color='red')
axs[1, 0].set_title("Scatter Plot")

# Plot 4: Histogram
import numpy as np
data = np.random.randn(1000)
axs[1, 1].hist(data, bins=30, color='green', edgecolor='black')
axs[1, 1].set_title("Histogram")

# Adjust layout
plt.tight_layout()

# Show the plots
plt.show()
```

Adjusting Plot Size and Layout

You can customize the size of the figure and adjust the layout for better spacing.

python
```
fig, axs = plt.subplots(2, 2, figsize=(12, 10))  # Adjusted figure size
plt.subplots_adjust(wspace=0.3, hspace=0.4)  # Adjust the spacing between subplots
```

- **wspace**: Adjusts the width between columns.
- **hspace**: Adjusts the height between rows.

Sharing Axes Between Subplots

To make comparisons easier, you can share axes between subplots.

python
```
fig, axs = plt.subplots(2, 1, sharex=True, sharey=True, figsize=(6, 8))

# Plot 1: Line plot
axs[0].plot(x, y, color='blue')
axs[0].set_title("Line Plot")
```

```
# Plot 2: Bar plot
axs[1].bar(x, y, color='orange')
axs[1].set_title("Bar Plot")

# Show plot
plt.show()
```

- **sharex=True**: Shares the x-axis between subplots.
- **sharey=True**: Shares the y-axis between subplots.

Controlling axes ticks

In Matplotlib, **ticks** are the markers on the axes that indicate the position of data points or intervals. You can customize these ticks to make your plots more readable or to fit specific data requirements.

Here's how to **control and customize axis ticks** in Matplotlib:

1. Controlling Major and Minor Ticks

Matplotlib provides **major** and **minor** ticks for each axis, which can be controlled separately.

Basic Example of Major and Minor Ticks

```
python
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a plot
plt.plot(x, y)

# Set major and minor ticks
plt.xticks([1, 2, 3, 4, 5], ['One', 'Two', 'Three', 'Four', 'Five'])  # Custom tick labels for x-axis
plt.yticks([10, 20, 30, 40], ['Low', 'Medium', 'High', 'Very High'])  # Custom tick labels for y-axis

# Display plot
plt.show()
```

In this example, the plt.xticks() and plt.yticks() functions are used to customize the **labels** of the ticks.

2. Manually Specifying Ticks and Labels

You can also manually specify **tick positions** and their corresponding **labels** using plt.xticks() and plt.yticks().

```
python
plt.plot(x, y)

# Manually setting x-axis and y-axis ticks
plt.xticks([1, 2, 3, 4, 5])  # Positions on the x-axis
plt.yticks([10, 15, 20, 25, 30, 35, 40])  # Positions on the y-axis
```

```
plt.show()
```

3. Customizing Tick Font Properties

You can customize the **font size**, **style**, and **rotation** of tick labels.

```
python
plt.plot(x, y)

# Customizing font size and rotation of ticks
plt.xticks(fontsize=12, rotation=45)  # 45-degree rotation for x-axis labels
plt.yticks(fontsize=10, rotation=90)  # 90-degree rotation for y-axis labels

plt.show()
```

4. Using MaxNLocator for Automatic Ticks

Matplotlib has built-in functionality like MaxNLocator for automatically adjusting ticks based on data.

```
python
from matplotlib.ticker import MaxNLocator

# Create plot
plt.plot(x, y)

# Automatically set 5 major ticks on the y-axis
plt.gca().yaxis.set_major_locator(MaxNLocator(nbins=5))

plt.show()
```

- **MaxNLocator(nbins=5)**: Sets the maximum number of ticks on the axis to 5. It will automatically place ticks with equal spacing.

5. Controlling Minor Ticks

Minor ticks are often used for more detailed control, but they are not shown by default. You can enable and control them.

```
python
from matplotlib.ticker import AutoMinorLocator

plt.plot(x, y)

# Enabling minor ticks
plt.minorticks_on()

# Customizing minor ticks
plt.gca().xaxis.set_minor_locator(AutoMinorLocator(4))  # 4 minor ticks between each major tick
plt.gca().yaxis.set_minor_locator(AutoMinorLocator(5))  # 5 minor ticks between each major tick

# Adding gridlines for minor ticks
plt.grid(which='minor', linestyle=':', linewidth=0.5)  # Dotted gridlines for minor ticks

plt.show()
```

- **AutoMinorLocator(n)**: Automatically creates n minor ticks between major ticks.
- **plt.minorticks_on()**: Enables minor ticks.
- **which='minor'**: Ensures that the gridlines and other elements affect only minor ticks.

6. Removing Ticks

If you don't want certain ticks to appear, you can remove them.

Removing All Ticks from Axis:
python
```
plt.plot(x, y)

# Remove x-axis ticks
plt.xticks([])

# Remove y-axis ticks
plt.yticks([])

plt.show()
```
Removing Only Tick Labels:

You can keep the tick marks but remove the labels.

python
```
plt.plot(x, y)

# Remove labels on x-axis but keep the ticks
plt.xticks([], [])

plt.show()
```

7. Formatting Tick Labels

You can format tick labels (e.g., showing currency, percentages, or scientific notation).

python
```
from matplotlib.ticker import FuncFormatter

# Custom function to format ticks
def currency_formatter(x, pos):
  return f"${x:.2f}"  # Format as currency

plt.plot(x, y)

# Apply custom tick formatter to y-axis
plt.gca().yaxis.set_major_formatter(FuncFormatter(currency_formatter))

plt.show()
```

Labels and legends

In Matplotlib, **labels** and **legends** are important tools for making your plots more informative and easier to interpret. Labels help identify the axes, and legends explain what each element (line, bar, scatter point, etc.) represents.

Here's how you can work with **labels** and **legends** in Matplotlib:

1. Axis Labels

Axis labels describe what the x and y axes represent.

python
```
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a line plot
plt.plot(x, y, label="Sales Growth", color='blue')

# Add x and y axis labels
plt.xlabel("Time (Months)")  # x-axis label
plt.ylabel("Sales ($)")  # y-axis label

# Add a title
plt.title("Sales Growth Over Time")

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

- **plt.xlabel()**: Adds a label to the x-axis.
- **plt.ylabel()**: Adds a label to the y-axis.

2. Adding a Legend

A **legend** is used to label multiple datasets or series in the plot. You can use label inside the plot function and then call plt.legend() to display it.

python
```
# Multiple plots
y2 = [15, 25, 35, 45, 55]
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')

# Add a legend
plt.legend()

# Add labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

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

- **label="..."**: Specifies the label for each dataset.
- **plt.legend()**: Displays the legend on the plot.

Customizing the Legend

You can customize the location, font size, and other properties of the legend.

python
```
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')

# Customizing legend location and font size
plt.legend(loc='upper left', fontsize=12)

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

plt.show()
```

- **loc='upper left'**: Specifies where the legend should appear (e.g., 'upper right', 'lower left').
- **fontsize=12**: Changes the font size of the legend.

Legend with Custom Markers

You can change the appearance of the legend markers to better match the plot.

python
```
plt.plot(x, y, label="Sales Growth", color='blue', linestyle='--', marker='o')
plt.plot(x, y2, label="Predicted Sales", color='green', linestyle='-', marker='^')

# Customizing the legend markers
plt.legend(markerfirst=True)  # Put markers in front of the label

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

plt.show()
```

- **markerfirst=True**: Places markers before the label text in the legend.

3. Multiple Legends

In cases where you want to add multiple legends or control where they appear, you can use the legend function with additional parameters.

python
```
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')

# Place legend in a specific location (e.g., outside the plot)
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")
```

```
plt.show()
```

- **bbox_to_anchor=(1, 1)**: Places the legend outside the plot area. You can adjust the tuple (x, y) to fine-tune its position.

4. Adding a Title to the Legend

You can also give a title to the legend to clarify what the legend represents.

```
python
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')

# Adding title to the legend
plt.legend(title="Sales Data")

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

plt.show()
```

- **title="..."**: Adds a title to the legend.

5. Controlling Legend Appearance

You can also control the background color, edge color, and transparency of the legend.

```
python
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')

# Customizing legend appearance
plt.legend(facecolor='lightgray', edgecolor='black', shadow=True)

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

plt.show()
```

- **facecolor='lightgray'**: Changes the background color of the legend.
- **edgecolor='black'**: Sets the color of the legend's border.
- **shadow=True**: Adds a shadow effect to the legend box.

6. Legends for Specific Elements

You may want to include legends for only certain plot elements (e.g., lines but not gridlines).

```
python
# Plot with gridlines and multiple lines
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='green')
```

```
plt.grid(True)

# Only include the lines in the legend, not the grid
plt.legend()

plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales vs Predicted Sales")

plt.show()
```

Annotations and Drawing on subplots

In Matplotlib, **annotations** and **drawing shapes or lines** on subplots are powerful tools that allow you to add extra information or highlight specific points in your plot. This can be useful for emphasizing trends, marking specific points, or providing additional context.

Here's a detailed guide on how to use **annotations** and **drawing** on subplots.

1. Adding Annotations

Annotations allow you to add text or arrows to specific points in your plot. This is useful for highlighting data points or providing extra information.

Basic Annotation

You can annotate a point using the plt.annotate() function, which takes the text and the coordinates of the point to annotate.

```
python
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

plt.plot(x, y, label="Sales Growth", color='blue')

# Annotating a specific point
plt.annotate('Highest Point', xy=(5, 40), xytext=(4, 35),
        arrowprops=dict(facecolor='red', arrowstyle='->'),
        fontsize=12, color='black')

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Annotation")

plt.legend()
plt.show()
```

Explanation of plt.annotate() parameters:

- **xy=(5, 40)**: The coordinates where the annotation arrow will point.

- **xytext=(4, 35)**: The position where the annotation text will appear.
- **arrowprops=dict(facecolor='red', arrowstyle='->')**: Specifies the appearance of the arrow (red color and a simple arrow).
- **fontsize=12**: Font size of the annotation text.
- **color='black'**: Color of the text.

2. Adding Multiple Annotations

You can add multiple annotations to different points on the same plot.

```
python
plt.plot(x, y, label="Sales Growth", color='blue')

# Annotate multiple points
plt.annotate('Point 1', xy=(1, 10), xytext=(1.5, 12),
        arrowprops=dict(facecolor='green', arrowstyle='->'),
        fontsize=12)
plt.annotate('Point 2', xy=(4, 30), xytext=(3.5, 28),
        arrowprops=dict(facecolor='purple', arrowstyle='->'),
        fontsize=12)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Multiple Annotations")

plt.legend()
plt.show()
```

3. Drawing Shapes (Lines, Rectangles, Circles, etc.)

Matplotlib also allows you to **draw shapes** like lines, rectangles, circles, and more to highlight specific areas in the plot.

Drawing a Line

You can draw lines using plt.plot() or plt.axvline() / plt.axhline() for vertical and horizontal lines, respectively.

```
python
plt.plot(x, y, label="Sales Growth", color='blue')

# Draw a vertical line at x=3
plt.axvline(x=3, color='red', linestyle='--', linewidth=2)

# Draw a horizontal line at y=20
plt.axhline(y=20, color='green', linestyle='-', linewidth=2)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Lines")

plt.legend()
```

```
plt.show()
```

Drawing a Rectangle

To draw a rectangle, use plt.gca().add_patch(). You can specify the rectangle's position and size.

```
python
import matplotlib.patches as patches

# Plot the data
plt.plot(x, y, label="Sales Growth", color='blue')

# Create a rectangle
rect = patches.Rectangle((1.5, 15), 2, 15, linewidth=2, edgecolor='purple', facecolor='orange', alpha=0.3)
plt.gca().add_patch(rect)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Rectangle")

plt.legend()
plt.show()
```

- **(1.5, 15)**: The starting point of the rectangle (x, y).
- **2, 15**: The width and height of the rectangle.
- **linewidth=2**: Border width of the rectangle.
- **facecolor='orange'**: Fill color of the rectangle.
- **alpha=0.3**: Transparency of the rectangle.

Drawing a Circle

To draw a circle, you can use patches.Circle.

```
python
# Plot the data
plt.plot(x, y, label="Sales Growth", color='blue')

# Create a circle
circle = patches.Circle((3, 25), 0.8, color='yellow', edgecolor='black', lw=2, alpha=0.4)
plt.gca().add_patch(circle)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Circle")

plt.legend()
plt.show()
```

- **(3, 25)**: The center of the circle (x, y).
- **0.8**: Radius of the circle.
- **color='yellow'**: Fill color of the circle.
- **lw=2**: Line width of the circle's border.
- **alpha=0.4**: Transparency of the circle.

4. Drawing Polygons

You can also draw polygons using patches.Polygon.

python
```
# Plot the data
plt.plot(x, y, label="Sales Growth", color='blue')

# Create a polygon (triangle in this case)
polygon = patches.Polygon(((2, 15), (3, 30), (4, 25)), closed=True, color='pink', edgecolor='black',
alpha=0.5)
plt.gca().add_patch(polygon)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Polygon")

plt.legend()
plt.show()
```

- **((2, 15), (3, 30), (4, 25))**: List of points for the polygon (vertices).
- **closed=True**: Ensures the polygon is closed (i.e., the last vertex connects to the first).
- **color='pink'**: Fill color of the polygon.

5. Combining Annotations and Drawing Shapes

You can combine annotations with shapes to further emphasize points or areas of interest.

python
```
# Plot the data
plt.plot(x, y, label="Sales Growth", color='blue')

# Annotate a point
plt.annotate('Highest Point', xy=(5, 40), xytext=(4, 35),
        arrowprops=dict(facecolor='red', arrowstyle='->'),
        fontsize=12)

# Draw a rectangle to highlight a region
rect = patches.Rectangle((1.5, 15), 2, 15, linewidth=2, edgecolor='purple', facecolor='orange', alpha=0.3)
plt.gca().add_patch(rect)

# Labels and title
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.title("Sales Growth with Annotations and Shapes")

plt.legend()
plt.show()
```

Saving plots to files : In Matplotlib, you can save your plots to a variety of file formats (such as PNG, PDF, SVG, JPG, etc.) using the plt.savefig() function. This is useful when you want to share your plots, include them in reports, or save them for future use.

Here's how you can save plots to files in different formats and with different options:

1. Basic Plot Saving

To save a plot, you simply call plt.savefig() after creating the plot.

python
```
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a plot
plt.plot(x, y)

# Save the plot as a PNG file
plt.savefig('sales_growth.png')

# Optionally, display the plot
plt.show()
```

- **plt.savefig('sales_growth.png')**: Saves the plot as a PNG file named sales_growth.png. You can change the file extension to save it in other formats (e.g., .pdf, .svg, .jpg).

2. Saving with Specific DPI (Resolution)

You can control the **resolution** of the saved plot using the dpi parameter. Higher DPI values result in higher-quality images.

python
```
# Save the plot with a DPI of 300 (high resolution)
plt.savefig('sales_growth_high_res.png', dpi=300)
```

- **dpi=300**: Saves the image with a resolution of 300 dots per inch (useful for high-quality prints).

3. Saving with Transparent Background

You can save plots with a **transparent background** by setting transparent=True.

python
```
# Save the plot with a transparent background
plt.savefig('sales_growth_transparent.png', transparent=True)
```

- **transparent=True**: Makes the background transparent (ideal for overlaying on other images).

4. Saving in Different File Formats

You can save your plot in various formats by simply changing the file extension:

- **PDF**: Great for vector graphics and printing.
- **SVG**: Scalable vector graphics format, ideal for high-quality prints.
- **JPG**: Compressed raster format, ideal for sharing on the web.

python
# Save as a PDF
plt.savefig('sales_growth.pdf')

# Save as an SVG
plt.savefig('sales_growth.svg')

# Save as a JPG
plt.savefig('sales_growth.jpg', quality=95) # Set image quality (1-100)

5. Saving with Specific Figure Size

You can also specify the figure size when saving the plot to ensure that it fits within certain dimensions.

python
Create a figure with a specific size
plt.figure(figsize=(8, 6)) # 8 inches wide by 6 inches tall

# Plot the data
plt.plot(x, y)

# Save the plot with the specified size
plt.savefig('sales_growth_resized.png', dpi=200)

- **figsize=(8, 6)**: Sets the figure size to 8 inches by 6 inches.
- **dpi=200**: Sets the resolution for saving.

6. Saving Only the Plot Area (Without Borders)

You can use bbox_inches='tight' to remove any unnecessary whitespace around the plot.

python
# Save the plot without extra borders or margins
plt.savefig('sales_growth_tight.png', bbox_inches='tight')

- **bbox_inches='tight'**: Automatically adjusts the bounding box to tightly fit the plot, removing extra whitespace around it.

7. Saving Multiple Subplots to a Single File

If you have multiple subplots, you can save the entire figure containing all subplots.

python
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

Plot on the first subplot
axes[0].plot(x, y)
axes[0].set_title('Sales Growth')

Plot on the second subplot
axes[1].plot(x, [i * 1.5 for i in y])
axes[1].set_title('Adjusted Sales Growth')

Save the figure with multiple subplots

```
plt.savefig('sales_growth_subplots.png')
```

- **fig, axes = plt.subplots(...)**: Creates multiple subplots.
- **plt.savefig()**: Saves the entire figure with subplots.

8. Saving with a Specific Quality (for JPG format)

For raster image formats like JPG, you can adjust the quality of the saved image.

```
python
# Save the plot as a JPG with high quality
plt.savefig('sales_growth_high_quality.jpg', quality=95)  # Quality scale from 1 to 100
```

9. Saving to a Specific Directory

You can specify the **path** and the **filename** where the plot should be saved.

```
python
# Save to a specific directory
plt.savefig('/path/to/directory/sales_growth.png')
```

- Replace **/path/to/directory/** with the actual directory path where you want the plot to be saved.

10. Closing the Plot After Saving

Once you've saved the plot, it's a good practice to close the figure if you're working with multiple plots to free up memory.

```
python
plt.savefig('sales_growth.png')
plt.close()  # Close the plot to free memory
```

Matplotlib Configuration using different plot styles

Matplotlib provides a range of **plot styles** to easily change the appearance of your plots, helping you create attractive, publication-quality visualizations without needing to manually adjust the properties of individual elements like lines, markers, or background colors. You can choose from predefined styles or create custom ones to suit your needs.

Here's an overview of how to configure **Matplotlib using different plot styles**.

1. Available Predefined Plot Styles

Matplotlib includes a variety of built-in **stylesheets** that allow you to quickly change the look of your plots. You can view all available styles by using:

```
python
import matplotlib.pyplot as plt

# List all available styles
print(plt.style.available)
```

Some of the common styles include:

- **'seaborn-white'**: A minimalist style with white background.
- **'seaborn-darkgrid'**: A grid with dark background (good for statistical plots).
- **'ggplot'**: Inspired by ggplot2 (R-based plotting library), a dark style with gridlines.
- **'bmh'**: Another minimalist style with muted colors.
- **'classic'**: The old Matplotlib style, closer to the default behavior before version 2.0.

2. Using Built-in Styles

To apply a style, you use plt.style.use() and pass the name of the style as an argument.

python
```
import matplotlib.pyplot as plt

# Apply a specific style
plt.style.use('seaborn-darkgrid')

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a plot with the selected style
plt.plot(x, y, label="Sales Growth", color='blue')
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.legend()

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

By changing the style, the appearance of the plot will automatically adjust to match the chosen style (e.g., gridlines, background color, font size, etc.).

3. Combining Multiple Styles

You can also combine multiple styles. For example, you can combine the **seaborn** style with a **dark grid**:

python
```
# Apply multiple styles
plt.style.use(['seaborn', 'dark_background'])

# Sample plot
plt.plot(x, y, label="Sales Growth", color='cyan')
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.legend()

plt.show()
```

This combines the "seaborn" style with a dark background. You can also chain more styles in the list.

4. Creating Custom Styles

If you want to create your own style or modify an existing one, you can define a custom style by creating a .mplstyle file. You can specify properties such as font size, line width, colors, etc., and save them in a file with a .mplstyle extension.

Here's an example of how you can create a custom style file (e.g., my_style.mplstyle):

text
```
# my_style.mplstyle
axes.titlesize : 14
axes.labelsize : 12
axes.labelweight : bold
lines.linewidth : 2
lines.markersize : 8
axes.grid : True
grid.alpha : 0.3
axes.facecolor : 'lightgray'
xtick.labelsize : 10
ytick.labelsize : 10
```

Then you can load this custom style using:

python
```
plt.style.use('path/to/your/my_style.mplstyle')

# Plotting with the custom style
plt.plot(x, y)
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.show()
```

Seaborn library

Seaborn is a powerful **Python visualization library** built on top of Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn makes it easy to generate complex plots with fewer lines of code, and it is especially well-suited for working with data frames, particularly those from **Pandas**.

It simplifies many tasks, such as handling categorical data, automatically calculating and visualizing statistical relationships, and applying attractive themes to plots.

Key Features of Seaborn:

1. **Attractive Default Styles**: Seaborn comes with aesthetically pleasing default themes, colors, and options for easy and beautiful plots.
2. **Built-in Themes**: Offers several built-in themes for creating publication-ready plots.
3. **High-level functions**: Functions for plotting regression, distribution, and relationship data, among other things.
4. **Integration with Pandas**: Directly works with Pandas DataFrames and Series.
5. **Statistical Plots**: Provides tools for creating plots like histograms, box plots, scatter plots, and even regression plots with confidence intervals.

1. Installing Seaborn

If you don't have Seaborn installed, you can install it via pip:

bash
pip install seaborn

Or using conda:

bash
CopyEdit
conda install seaborn

2. Basic Plotting with Seaborn

Seaborn simplifies the process of creating various types of plots, such as:

- **Line plots**
- **Scatter plots**
- **Box plots**
- **Histograms**
- **Bar plots**
- **Pair plots**
- **Heatmaps**

Some of the common types of plots.

2.1. Line Plot

The sns.lineplot() function creates line plots.

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

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create a line plot
sns.lineplot(x=x, y=y)

# Title and labels
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")

plt.show()
```

2.2. Scatter Plot

The sns.scatterplot() function creates scatter plots.

```
# Create a scatter plot
```

```
sns.scatterplot(x=x, y=y, color='red')

# Title and labels
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")

plt.show()
```

2.3. Bar Plot

Bar plots are useful for comparing categorical data. Use sns.barplot().

```
python
# Sample categorical data
categories = ['Category A', 'Category B', 'Category C']
values = [10, 20, 15]

# Create a bar plot
sns.barplot(x=categories, y=values)

# Title and labels
plt.title("Category Sales")
plt.xlabel("Categories")
plt.ylabel("Sales ($)")

plt.show()
```

2.4. Box Plot

A box plot (or box-and-whisker plot) is useful for visualizing the distribution of data.

```
python
# Sample data
data = [10, 20, 25, 30, 40, 15, 35, 45]

# Create a box plot
sns.boxplot(data=data)

# Title and labels
plt.title("Sales Distribution")
plt.ylabel("Sales ($)")

plt.show()
```

2.5. Histogram

Histograms help to visualize the distribution of a dataset. Use sns.histplot().

```
python
# Sample data
data = [10, 20, 20, 25, 25, 30, 30, 30, 35, 40]

# Create a histogram
sns.histplot(data, kde=True)  # 'kde=True' adds a kernel density estimate

# Title and labels
```

```
plt.title("Sales Distribution")
plt.xlabel("Sales ($)")
plt.ylabel("Frequency")

plt.show()
```

2.6. Heatmap

Seaborn also makes it easy to visualize a **correlation matrix** or other 2D data with heatmaps using sns.heatmap().

```
python
import numpy as np

# Create a correlation matrix
data = np.random.rand(10, 12)
sns.heatmap(data, annot=True, cmap='coolwarm', cbar=True)

# Title
plt.title("Heatmap Example")

plt.show()
```

3. Pair Plot

The sns.pairplot() function automatically generates a grid of scatter plots for every pair of columns in a DataFrame, making it a great tool for visualizing pairwise relationships in datasets.

```
python
import seaborn as sns
import pandas as pd

# Load the built-in 'iris' dataset from Seaborn
iris = sns.load_dataset('iris')

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

plt.show()
```

- **hue='species'**: Colors the points based on the 'species' column, helping to distinguish different categories in the data.

4. Customizing Seaborn Plots

Seaborn comes with **aesthetic settings** out of the box, but you can also customize the plots further using matplotlib functions (like changing labels, colors, and other features).

4.1. Set Plot Styles

You can change the overall style of Seaborn plots using sns.set_style() and sns.set_palette().

```
python
# Set the plot style and color palette
sns.set_style("whitegrid")
```

```
sns.set_palette("muted")

# Create a plot
sns.lineplot(x=x, y=y)

plt.title("Sales Growth Over Time")
plt.show()
```

- **sns.set_style()**: Available styles include "darkgrid", "whitegrid", "dark", "white", and "ticks".
- **sns.set_palette()**: Sets the color palette (e.g., "deep", "muted", "bright", etc.).

4.2. Titles, Labels, and Legends

You can easily customize titles, axis labels, and legends in Seaborn plots using plt.title(), plt.xlabel(), plt.ylabel(), and plt.legend() from matplotlib.

```
python
sns.scatterplot(x=x, y=y, color='blue')
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")
plt.legend(["Sales Growth"], loc="upper left")

plt.show()
```

5. Using Seaborn with Pandas DataFrames

Seaborn works seamlessly with Pandas DataFrames, allowing you to plot entire columns directly.

```
python
import seaborn as sns
import pandas as pd

# Load the built-in 'tips' dataset
tips = sns.load_dataset('tips')

# Create a scatter plot using DataFrame columns
sns.scatterplot(x='total_bill', y='tip', data=tips, hue='sex')

plt.title("Tip vs Total Bill")
plt.show()
```

6. Regression Plots

Seaborn also provides built-in tools to plot **regression lines** and confidence intervals using sns.regplot().

```
python
# Create a regression plot
sns.regplot(x='total_bill', y='tip', data=tips)

plt.title("Regression Line: Tip vs Total Bill")
plt.show()
```

Making sense of data through advanced visualization: Controlling line properties of chart

When visualizing data, especially through **line charts**, it's essential to be able to control the appearance of the lines to highlight specific trends, patterns, or relationships. This can be done using **Matplotlib** and **Seaborn** by modifying line properties such as color, style, width, markers, and transparency.

Here's a guide on how to **control line properties** in **Matplotlib** to customize the look of your charts and make your visualizations clearer and more insightful.

1. Controlling Line Properties in Matplotlib

In **Matplotlib**, you can control various properties of the line, such as its **color**, **line style**, **line width**, **markers**, **markersize**, and more, to better represent the data.

1.1. Line Color

The color of the line can be set using the color parameter. You can use various representations like color names, RGB values, hex color codes, or even predefined color styles.

python
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 30, 40, 50]

# Create a plot with a red line
plt.plot(x, y, color='red')  # You can also use hex, e.g., '#FF5733'
plt.title("Line with Red Color")
plt.show()
```

1.2. Line Style

You can control the **style** of the line by using the linestyle parameter. Some common line styles include solid ('-'), dashed ('--'), dotted (':'), and dash-dot ('-.').

python
```
# Dashed line
plt.plot(x, y, color='blue', linestyle='--')
plt.title("Dashed Line")
plt.show()

# Dotted line
plt.plot(x, y, color='green', linestyle=':')
plt.title("Dotted Line")
plt.show()
```

1.3. Line Width

You can change the **thickness** of the line by using the linewidth parameter.

python
```
# Thick line
```

```
plt.plot(x, y, color='purple', linewidth=4)
plt.title("Thick Line")
plt.show()

# Thin line
plt.plot(x, y, color='orange', linewidth=1)
plt.title("Thin Line")
plt.show()
```

1.4. Line Markers

Markers are symbols placed at each data point to make it easier to see individual points on the line. The marker parameter controls the marker style, and the markersize parameter controls its size.

Some common marker styles:

- 'o': Circle
- 's': Square
- '^': Triangle
- 'D': Diamond
- '*': Star
- x: Cross

python

```
# Line with circular markers
plt.plot(x, y, color='blue', marker='o', markersize=8)
plt.title("Line with Circular Markers")
plt.show()

# Line with square markers
plt.plot(x, y, color='brown', marker='s', markersize=6)
plt.title("Line with Square Markers")
plt.show()

# Line with stars as markers
plt.plot(x, y, color='cyan', marker='*', markersize=10)
plt.title("Line with Star Markers")
plt.show()
```

1.5. Combining Line Styles and Markers

You can also combine line styles, markers, and colors for more customization.

python

```
# Line with dashed style, red color, and circle markers
plt.plot(x, y, color='red', linestyle='--', marker='o', markersize=8, linewidth=2)
plt.title("Dashed Line with Circle Markers")
plt.show()
```

1.6. Transparency (Alpha Channel)

Transparency can be controlled with the alpha parameter, which accepts values between 0 (fully transparent) and 1 (fully opaque).

python
```
# Line with 50% transparency
plt.plot(x, y, color='blue', alpha=0.5)
plt.title("Line with Transparency")
plt.show()
```

2. Customizing Multiple Lines in a Plot

In many cases, you'll want to compare multiple lines within the same plot. You can customize each line separately by adding multiple plt.plot() calls with different parameters.

python
```
# Plot multiple lines with different properties
y2 = [5, 15, 25, 35, 45]

plt.plot(x, y, color='green', linestyle='-', marker='o', label='Line 1')
plt.plot(x, y2, color='red', linestyle='--', marker='s', label='Line 2')

# Add title, labels, and legend
plt.title("Multiple Lines with Custom Properties")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()

plt.show()
```

3. Advanced Line Customization in Seaborn

In Seaborn, many of these properties can be controlled through the sns.lineplot() function, which provides a simple interface for customizing line properties, especially in relation to Pandas DataFrames.

3.1. Basic Line Plot in Seaborn

Seaborn's sns.lineplot() automatically handles several customization options.

python
```
import seaborn as sns

# Sample data
data = sns.load_dataset('tips')

# Simple line plot
sns.lineplot(x='total_bill', y='tip', data=data)
plt.title("Tip vs Total Bill")
plt.show()
```
3.2. Customizing Line Style and Color in Seaborn

Seaborn allows you to customize line styles, colors, and markers directly through its API.

python
```
sns.lineplot(x='total_bill', y='tip', data=data, color='purple', linestyle='--', marker='o')
plt.title("Tip vs Total Bill with Customization")
plt.show()
```

3.3. Using Hue for Different Lines

You can plot multiple lines for different groups in the data using the hue parameter to color the lines based on a categorical variable.

python
```
# Use 'hue' to color the lines based on the 'sex' category
sns.lineplot(x='total_bill', y='tip', data=data, hue='sex')
plt.title("Tip vs Total Bill (Colored by Gender)")
plt.show()
```

4. Advanced Customization Example

Here's an example where we control almost every property of the lines and markers in a plot:

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

# Data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create a plot with customized properties for two lines
plt.plot(x, y1, color='magenta', linestyle='-', marker='o', markersize=5, linewidth=2, label='sin(x)')
plt.plot(x, y2, color='green', linestyle='--', marker='s', markersize=7, linewidth=3, label='cos(x)')

# Title, labels, and legend
plt.title("Sine and Cosine Waves")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()

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

Playing with text

Text is a powerful tool when it comes to adding **annotations**, **labels**, and **titles** to your plots. In both **Matplotlib** and **Seaborn**, you can customize text in various ways to enhance the clarity and readability of your visualizations.

Here's a guide to **playing with text** in your visualizations:

1. Adding Titles, Axis Labels, and Legends in Matplotlib

Before diving into advanced text features, let's start with the basics of adding text elements such as titles, axis labels, and legends to your plot.

1.1. Adding Titles

You can use plt.title() to add a title to the plot. It also accepts various parameters to control text properties like size, color, and font.

python
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Create the plot
plt.plot(x, y)

# Add title
plt.title("Sales Growth Over Time", fontsize=16, color='blue', fontweight='bold')

# Show plot
plt.show()
```
1.2. Adding Axis Labels

Use plt.xlabel() and plt.ylabel() to add labels to the x-axis and y-axis, respectively.

python
```
plt.plot(x, y)

# Add axis labels
plt.xlabel("Time (Months)", fontsize=12, color='green')
plt.ylabel("Sales ($)", fontsize=12, color='green')

# Show plot
plt.show()
```
1.3. Adding Legends

Legends help to identify different data series on a plot. You can add a legend by using plt.legend(). Pass labels for each plot series to the label argument.

python
```
# Create another dataset
y2 = [5, 15, 25, 35, 45]

# Plot both datasets
plt.plot(x, y, label="Sales Growth", color='blue')
plt.plot(x, y2, label="Predicted Sales", color='red')

# Add legend
plt.legend(title="Sales Data", loc="upper left", fontsize=10)

# Show plot
plt.show()
```

2. Annotating Plots with Text

Annotations can help provide additional context to a plot, such as highlighting specific points or explaining a feature of the graph.

2.1. Basic Annotation

You can use plt.text() to place text at any location on the plot.

python
```
# Create the plot
plt.plot(x, y)

# Annotate a specific point
plt.text(3, 25, "Peak", fontsize=12, color='red', ha='center')

plt.show()
```

- **ha='center'**: Horizontal alignment of the text (options include 'left', 'right', and 'center').
- You can also use va for vertical alignment.

2.2. Annotating with Arrows

You can add arrows to point to specific regions on the plot using plt.annotate().

python
```
# Create the plot
plt.plot(x, y)

# Annotate with an arrow
plt.annotate("Peak Point", xy=(3, 25), xytext=(2, 30),
        arrowprops=dict(facecolor='blue', shrink=0.05),
        fontsize=12, color='purple')

plt.show()
```

- **xy**: The location of the point you're annotating (the point on the plot).
- **xytext**: The position where the text will appear.
- **arrowprops**: Customize the arrow (e.g., color, line width, shrink ratio).

2.3. Adding Multiple Annotations

You can add multiple annotations to different points in the same plot.

python
```
# Create the plot
plt.plot(x, y)

# Annotate multiple points
plt.annotate("Start", xy=(1, 10), xytext=(2, 15), arrowprops=dict(facecolor='green'))
plt.annotate("End", xy=(5, 40), xytext=(4, 45), arrowprops=dict(facecolor='red'))

plt.show()
```

3. Customizing Text with Font Properties

Matplotlib provides many ways to control text appearance, such as font size, weight, family, and style.

3.1. Controlling Font Properties

You can specify font properties using fontdict or through individual parameters.

python
```
# Customize font properties
font_properties = {'family': 'serif', 'color': 'darkred', 'weight': 'bold', 'size': 14}

# Title with custom font
plt.title("Sales Over Time", fontdict=font_properties)

# Show plot
plt.show()
```
3.2. Using rcParams to Set Default Text Properties

You can set the default text properties for all plots in a session using plt.rcParams.

python
```
# Set global font properties
plt.rcParams.update({'font.size': 14, 'font.family': 'serif', 'axes.labelweight': 'bold'})

# Plot with default font properties
plt.plot(x, y)
plt.title("Sales Growth Over Time")
plt.xlabel("Time (Months)")
plt.ylabel("Sales ($)")

plt.show()
```
4. Advanced Text Customizations

4.1. Rotating Text

You can rotate text elements like titles and labels for better readability.

python
```
plt.plot(x, y)

# Rotate x-axis labels
plt.xticks(rotation=45)

# Rotate title
plt.title("Sales Growth Over Time", rotation=45)

plt.show()
```
4.2. Using Mathematical Expressions in Text

Matplotlib allows the use of **LaTeX-style** mathematical expressions in text.

python
```
# Plot a sine curve
x = np.linspace(0, 10, 100)
```

```
y = np.sin(x)

plt.plot(x, y)

# Add a mathematical expression as a title
plt.title(r"y = \sin(x)", fontsize=16)

plt.show()
```

- The r before the string makes it a raw string (necessary for LaTeX-style formatting).
- The text will be rendered as y = sin(x) with the correct math symbols.

4.3. Text on Polar Plots

You can also add text to **polar plots**, where the theta and r parameters are used to position the text.

```
python
# Create a polar plot
theta = np.linspace(0, 2*np.pi, 100)
r = np.abs(np.sin(theta))

plt.subplot(projection='polar')
plt.plot(theta, r)

# Annotate in polar coordinates
plt.text(np.pi/4, 0.5, "Peak", color='red', fontsize=15, ha='center')

plt.show()
```

5. Interactive Text with matplotlib

Interactive plots allow you to dynamically add and move text elements, making the exploration of your data more engaging.

```
python
import matplotlib.pyplot as plt

# Create interactive text at a certain position
fig, ax = plt.subplots()

line, = ax.plot(x, y)
text = ax.text(3, 25, "Drag me!", fontsize=15, ha='center', color='blue')

def on_move(event):
   # Update text position with mouse coordinates
   if event.inaxes:
      text.set_position((event.xdata, event.ydata))
      fig.canvas.draw()

fig.canvas.mpl_connect('motion_notify_event', on_move)

plt.show()
```

This allows you to drag the annotation on the plot using the mouse.

Styling your plot

Styling your plots is a key aspect of making your visualizations more effective and visually appealing. Matplotlib and Seaborn provide several ways to style your plots, from simple color changes to more advanced customizations like themes, grids, and specific element styling.

Here's a guide to **styling your plots** in **Matplotlib** and **Seaborn**.

1. Styling Plots in Matplotlib

Matplotlib provides a lot of flexibility when it comes to customizing and styling your plots. Here are some key styling features:

1.1. Using Matplotlib's Predefined Styles

Matplotlib includes a variety of predefined styles, which you can use to quickly change the appearance of your plots. To apply a style, use plt.style.use().

python
```
import matplotlib.pyplot as plt

# Apply a predefined style
plt.style.use('seaborn-darkgrid')

# Create the plot
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y)

# Show the plot
plt.title("Styled Plot with Seaborn Dark Grid")
plt.show()
```

Common styles include:

- 'seaborn-darkgrid'
- 'seaborn-notebook'
- 'seaborn-white'
- 'fivethirtyeight'
- 'ggplot'

1.2. Customizing Line Properties

Control the **color**, **line width**, **style**, and **markers** of your plot for more fine-grained styling.

python
```
# Create a plot with custom line properties
plt.plot(x, y, color='red', linewidth=3, linestyle='--', marker='o', markersize=8)
plt.title("Styled Plot with Custom Line Properties")
plt.show()
```

1.3. Customizing the Grid

Grids help readers track values on the plot. You can control whether to display the grid and how it appears.

python
```
# Plot with grid
plt.plot(x, y)
plt.grid(True)  # Display the grid

# Customize grid lines
plt.grid(which='both', color='gray', linestyle='--', linewidth=0.5)

plt.title("Plot with Customized Grid Lines")
plt.show()
```

- **which**: 'both', 'major', or 'minor' to specify which grid lines to display.
- **color**, **linestyle**, and **linewidth** control the appearance of grid lines.

1.4. Controlling Ticks

You can style the **tick marks** on both axes using plt.xticks() and plt.yticks() to adjust the font size, color, and rotation.

python
```
# Customize ticks
plt.plot(x, y)
plt.xticks(fontsize=12, color='blue', rotation=45)
plt.yticks(fontsize=12, color='red')

plt.title("Plot with Styled Ticks")
plt.show()
```
1.5. Using rcParams for Global Customization

You can use **rcParams** to apply global styling settings across all plots.

python
```
# Customize default styles globally
plt.rcParams.update({
   'axes.titleweight': 'bold',  # Bold titles
   'axes.labelweight': 'bold',  # Bold labels
   'axes.grid': True,          # Show grid by default
   'grid.alpha': 0.5,          # Set grid transparency
   'grid.color': 'black',      # Grid line color
   'font.size': 12             # Set default font size
})

# Create a plot to reflect the global styles
plt.plot(x, y)
plt.title("Plot with Global Style Customizations")
plt.show()
```

2. Styling Plots in Seaborn

Seaborn is built on top of Matplotlib and provides an even more straightforward way to style plots. It also includes several built-in themes that allow you to customize the overall appearance of your plots with minimal code.

2.1. Using Seaborn's Built-in Themes

Seaborn has three primary themes that you can apply to your plots:

- **darkgrid** (default)
- **whitegrid**
- **dark**
- **white**
- **ticks**

To set a theme, you can use the sns.set_style() function.

python
```
import seaborn as sns

# Set the theme for the plot
sns.set_style("whitegrid")

# Create a simple line plot
sns.lineplot(x=x, y=y)
plt.title("Line Plot with Whitegrid Style")
plt.show()
```

2.2. Customizing the Color Palette

Seaborn allows you to easily change the color palette of your plots using sns.set_palette(). You can choose from predefined palettes or create your own.

python
```
# Set a custom color palette
sns.set_palette("husl")  # Or use other palettes like "coolwarm", "deep", etc.

# Create a plot
sns.lineplot(x=x, y=y)
plt.title("Line Plot with Custom Palette")
plt.show()
```

You can also define a custom list of colors, for example:

python
```
sns.set_palette(["#1f77b4", "#ff7f0e", "#2ca02c"])
```

2.3. Combining Styles and Palettes

You can apply both themes and palettes at the same time for even more control.

python
```
# Set the theme and palette
sns.set_style("ticks")
sns.set_palette("pastel")

# Create a plot
sns.lineplot(x=x, y=y)
plt.title("Line Plot with Ticks Style and Pastel Palette")
plt.show()
```

2.4. Plotting with Seaborn's Grid Function

Seaborn provides functions like sns.FacetGrid() and sns.pairplot() to create multi-panel plots that are automatically styled.

python
```
# Example using FacetGrid for a multi-plot grid layout
tips = sns.load_dataset("tips")
g = sns.FacetGrid(tips, col="time", row="sex")
g.map(sns.scatterplot, "total_bill", "tip")
plt.show()
```

2.5. Customizing Plot Elements in Seaborn

You can also customize individual elements of the plot, such as axes labels, titles, and legends.

python
```
# Create a plot with custom elements
sns.lineplot(x=x, y=y)
plt.title("Customized Plot", fontsize=16, color='darkblue')
plt.xlabel("Time (Months)", fontsize=14, color='darkgreen')
plt.ylabel("Sales ($)", fontsize=14, color='darkgreen')

# Show plot
plt.show()
```

3. Advanced Styling Options

3.1. Adding Annotations

Annotations can highlight key points or trends in your plot. You can use plt.annotate() for more customized annotations.

python
```
# Add annotation to a plot
plt.plot(x, y)
plt.annotate("Peak Value", xy=(3, 25), xytext=(4, 35),
        arrowprops=dict(facecolor='green', shrink=0.05),
        fontsize=12, color='red')
plt.show()
```

3.2. Adding Subplots with Different Styles

You can use **subplots** with different styles to compare various data representations.

python
```
# Create subplots with different styles
fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# Apply different styles to each subplot
plt.style.use('seaborn-darkgrid')
axs[0].plot(x, y)
axs[0].set_title("Dark Grid Style")
```

```
plt.style.use('seaborn-whitegrid')
axs[1].plot(x, y)
axs[1].set_title("White Grid Style")

plt.show()
```

3d plot of surface

Creating 3D plots in Matplotlib is a great way to visualize data that has three variables. You can plot 3D surfaces, scatter plots, and wireframes to represent your data in a 3D space. For surface plots, the Axes3D module in Matplotlib is used.

Here's a step-by-step guide to creating a **3D surface plot** using Matplotlib.

1. Importing Necessary Libraries

First, import the necessary libraries and modules. You need to import Axes3D from mpl_toolkits.mplot3d to create 3D plots.

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

2. Generating Data for 3D Plot

For a 3D surface plot, you'll typically generate X, Y, and Z data. Here's an example using a simple mathematical function, such as Z=X2+Y2Z = X^2 + Y^2Z=X2+Y2, to create a 3D surface.

python
```
# Create meshgrid for X and Y values
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)

# Define Z values using a mathematical function
z = x**2 + y**2
```

3. Creating the 3D Plot

Now, use Axes3D to plot the surface. We will use the plot_surface method for the surface plot.

python
```
# Create a figure
fig = plt.figure(figsize=(10, 7))

# Add 3D subplot
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(x, y, z, cmap='viridis')
```

```
# Add labels
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')

# Add a title
ax.set_title('3D Surface Plot')

# Show the plot
plt.colorbar(surf)  # Show color bar for the surface
plt.show()
```

4. Customizing the 3D Plot

You can customize your 3D surface plot with various options:

- **Changing the colormap**: You can change the colormap by modifying the cmap argument in plot_surface(). For example, use 'plasma', 'inferno', or 'coolwarm'.
- **Adding a color bar**: The plt.colorbar() function adds a color bar to indicate the range of values in the surface.
- **Adjusting the angle and rotation**: You can rotate the plot interactively in a Jupyter notebook or use ax.view_init() to set the viewing angle programmatically.

Example of changing the viewing angle:

python
```
# Set the view angle
ax.view_init(elev=30, azim=45)
```

5. Adding Wireframe to 3D Plot

Sometimes, you may want to add a wireframe along with the surface plot to better understand the structure of the data. This can be done using plot_wireframe.

python
```
# Plot the wireframe
ax.plot_wireframe(x, y, z, color='black', linewidth=0.5)

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

6. Example: Complete Code

Here is the complete code for generating a 3D surface plot with a wireframe overlay:

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

# Create meshgrid for X and Y values
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
```

```
x, y = np.meshgrid(x, y)

# Define Z values using a mathematical function
z = x**2 + y**2

# Create a figure
fig = plt.figure(figsize=(10, 7))

# Add 3D subplot
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(x, y, z, cmap='viridis')

# Plot the wireframe
ax.plot_wireframe(x, y, z, color='black', linewidth=0.5)

# Add labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
ax.set_title('3D Surface Plot with Wireframe')

# Show color bar
plt.colorbar(surf)

# Set view angle
ax.view_init(elev=30, azim=45)

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