VISUALIZATION WITH PYTHON

---

1. Introduction to Data Visualization

Data visualization is the graphical representation of information and data. It's a critical skill in data science, business intelligence, and research, enabling us to communicate complex insights through visual storytelling.

Why Visualization Matters

Exploratory Data Analysis (EDA):

- Quickly identify patterns, trends, and outliers
- Understand data distributions and relationships
- Guide hypothesis formation and testing

Communication and Presentation:

- Transform complex datasets into understandable visuals
- Support decision-making processes
- Enable stakeholder engagement

Cognitive Processing:

- Humans process visual information faster than text
- Pattern recognition through visual cues
- Memory retention through visual associations

Python's Visualization Ecosystem

Python offers a rich ecosystem of visualization libraries, each with unique strengths:

- **Matplotlib:** Maximum customization and control
- **Seaborn:** Beautiful statistical visualizations with minimal code
- **Plotly:** Interactive, web-ready plots with dashboard capabilities
- **Bokeh:** Large dataset handling with interactive web applications
- **Pandas:** Quick plotting integrated with data manipulation

---

2. Matplotlib: The Foundation Library

Matplotlib is the most fundamental plotting library in Python, providing the foundation for many other visualization tools. It offers unparalleled control over every aspect of a plot.

2.1 Library Overview

Unique Features:

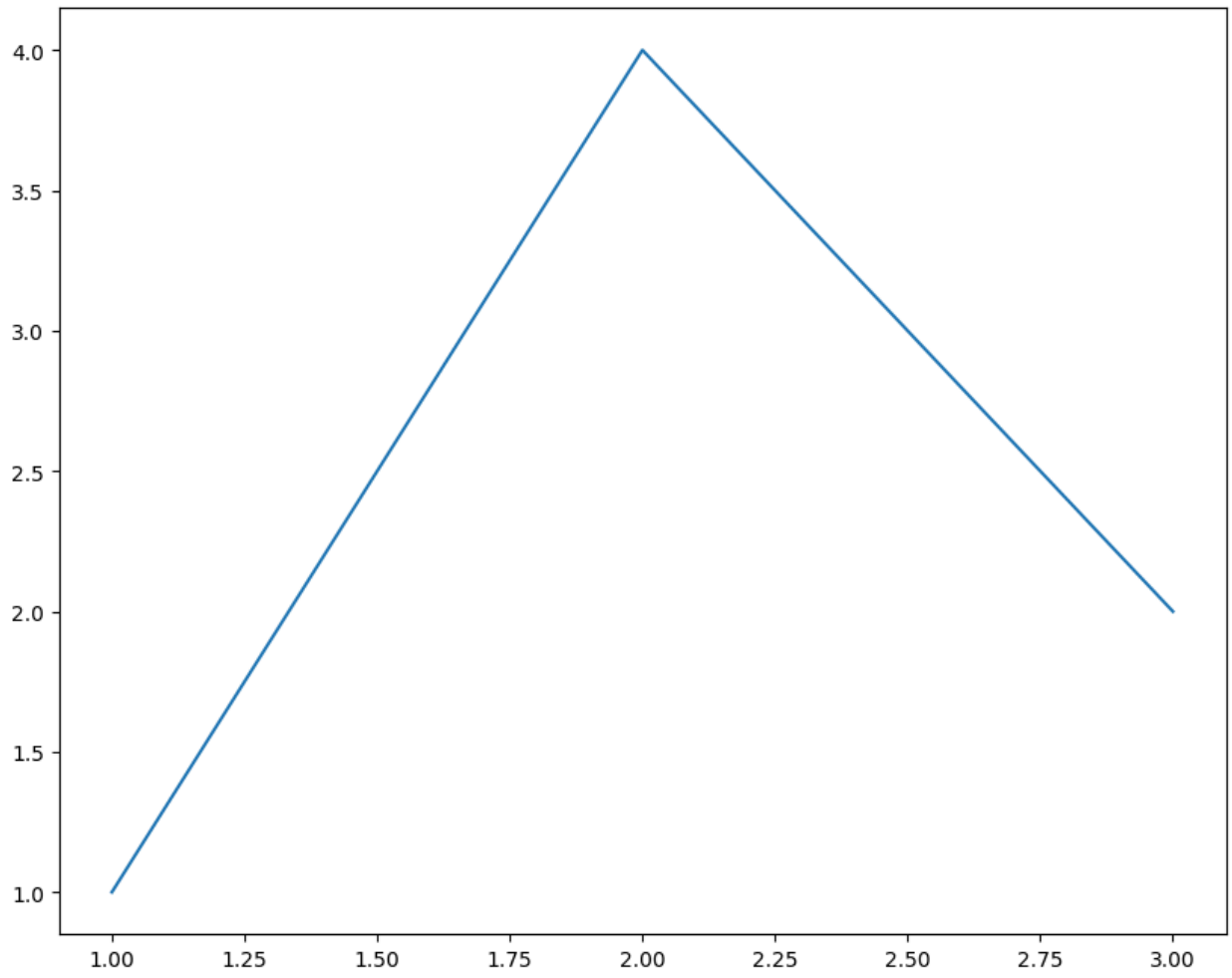
- Complete control over plot aesthetics
- Publication-quality figures
- Extensive customization options
- Object-oriented and pyplot interfaces
- Backend flexibility for different output formats

Typical Use Cases:

- Scientific publications
- Custom visualization requirements
- Educational materials
- Complex multi-panel figures

2.2 Understanding Matplotlib Architecture

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

# The matplotlib hierarchy
fig = plt.figure(figsize=(10, 8))  # Figure: the main container
ax = fig.add_subplot(111)          # Axes: the plotting area
ax.plot([1, 2, 3], [1, 4, 2])     # Artist: the actual plot elements
plt.show()
```



Key Components:

- **Figure:** The entire window or page
- **Axes:** Individual plots within the figure
- **Axis:** X and Y coordinate systems
- **Artists:** All visible elements (lines, text, etc.)

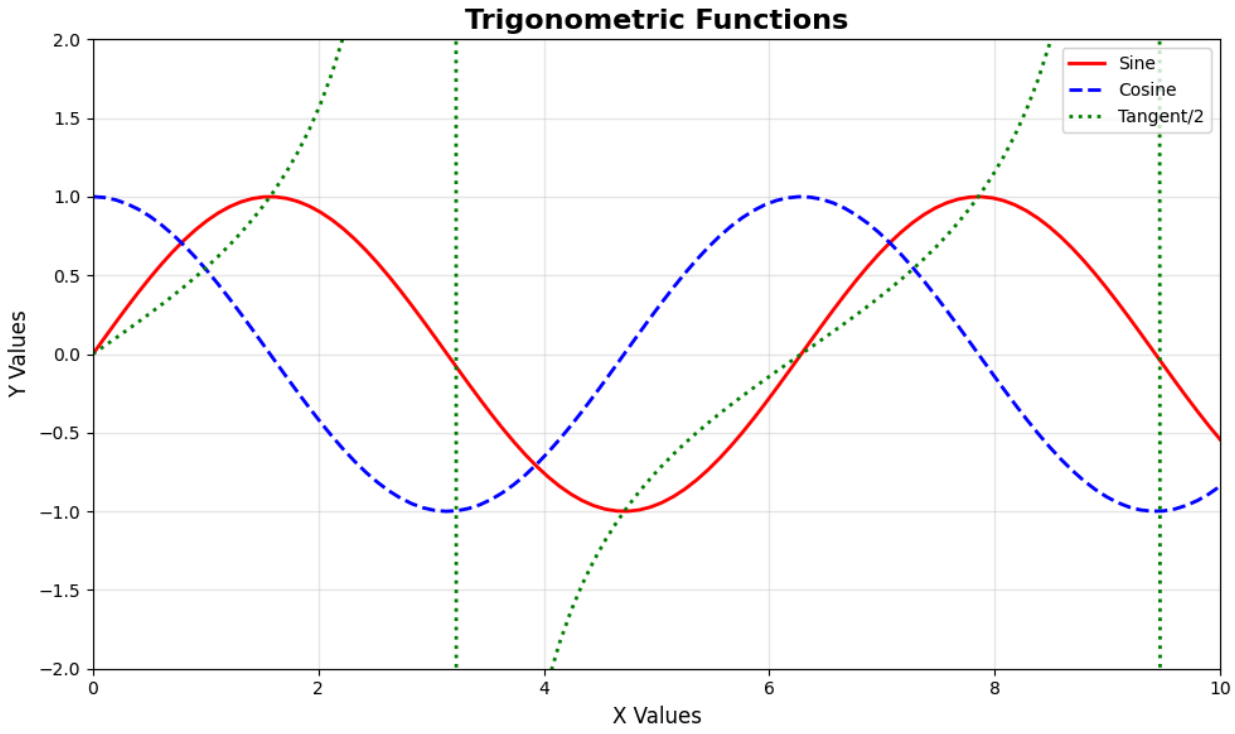
2.3 Graph Types and Examples

Line Plots

Use Case: Time series data, continuous relationships, trend analysis

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

# Sample data
x = np.linspace(0, 10, 100)
```

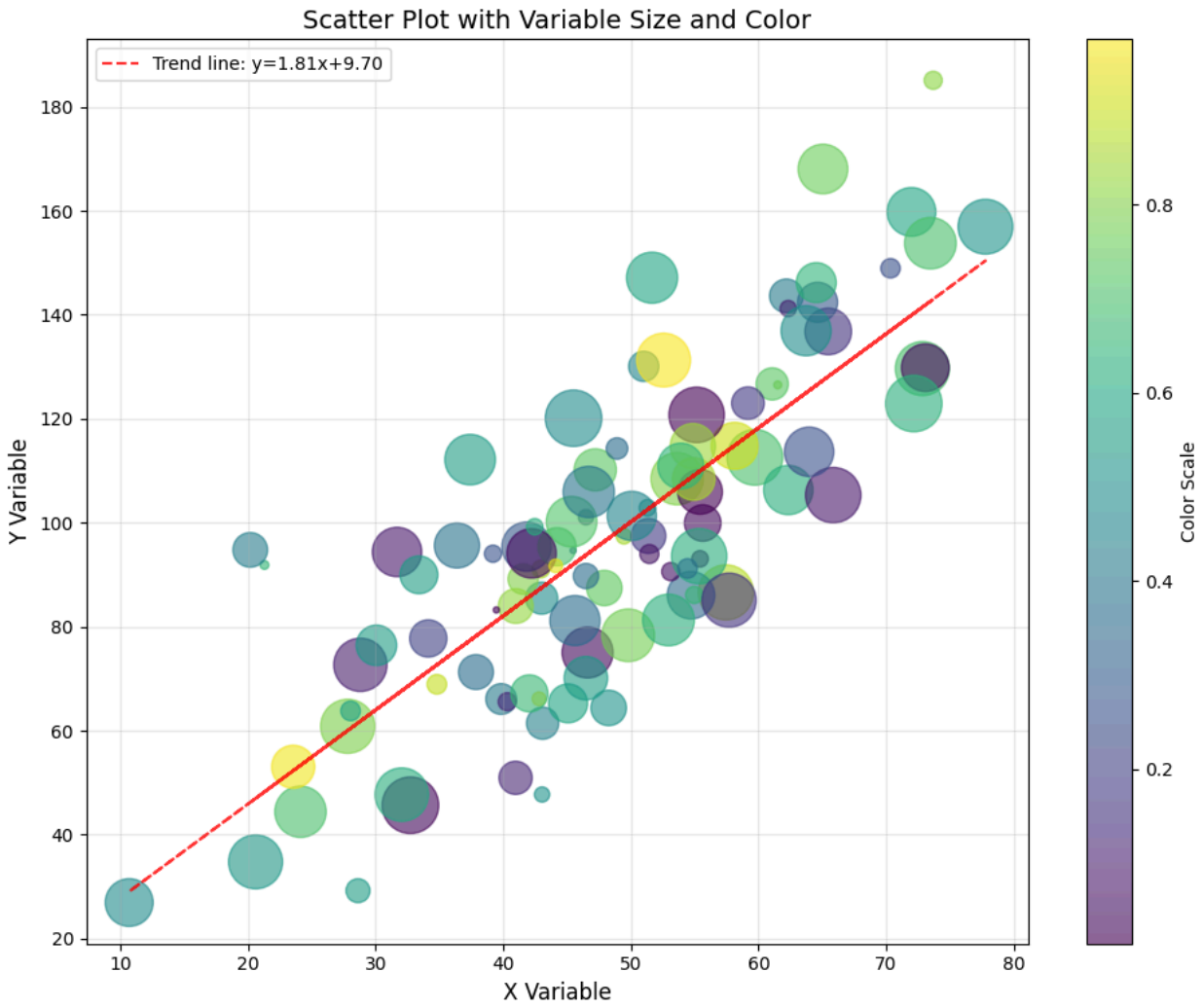
```
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x/2)

# Create line plot
plt.figure(figsize=(10, 6))
plt.plot(x, y1, label='Sine', linestyle='-', color='red', linewidth=2)
plt.plot(x, y2, label='Cosine', linestyle='--', color='blue', linewidth=2)
plt.plot(x, y3, label='Tangent/2', linestyle=':', color='green', linewidth=2)

# Customization
plt.title("Trigonometric Functions", fontsize=16, fontweight='bold')
plt.xlabel("X Values", fontsize=12)
plt.ylabel("Y Values", fontsize=12)
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.xlim(0, 10)
plt.ylim(-2, 2)

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



Scatter Plots

Use Case: Correlation analysis, pattern identification, outlier detection

```
# Generate sample data
np.random.seed(42)
n_points = 100
x = np.random.normal(50, 15, n_points)
y = 2 * x + np.random.normal(0, 20, n_points)
colors = np.random.rand(n_points)
sizes = 1000 * np.random.rand(n_points)

# Create scatter plot
plt.figure(figsize=(10, 8))
scatter = plt.scatter(x, y, c=colors, s=sizes, alpha=0.6, cmap='viridis')

# Add colorbar
plt.colorbar(scatter, label='Color Scale')

# Customization
plt.title("Scatter Plot with Variable Size and Color", fontsize=14)
plt.xlabel("X Variable", fontsize=12)
plt.ylabel("Y Variable", fontsize=12)
plt.grid(True, alpha=0.3)

# Add trend line
z = np.polyfit(x, y, 1)
p = np.poly1d(z)
plt.plot(x, p(x), "r--", alpha=0.8, label=f'Trend line: y={z[0]:.2f}x+{z[1]:.2f}')
plt.legend()

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

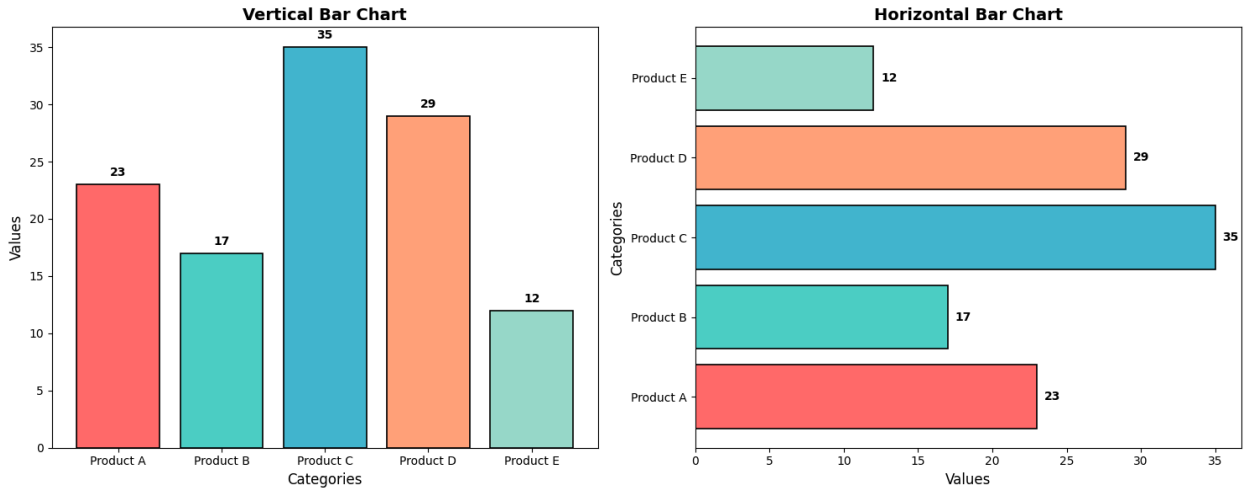


Bar Charts

Use Case: Categorical comparisons, survey results, performance metrics

```
# Sample data
categories = ['Product A', 'Product B', 'Product C', 'Product D', 'Product E']
values = [23, 17, 35, 29, 12]
colors = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#FFA07A', '#98D8C8']

# Create bar chart
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Vertical bar chart
bars1 = ax1.bar(categories, values, color=colors, edgecolor='black', linewidth=1.2)
ax1.set_title("Vertical Bar Chart", fontsize=14, fontweight='bold')
```

```
ax1.set_ylabel("Values", fontsize=12)
ax1.set_xlabel("Categories", fontsize=12)

# Add value labels on bars
for bar, value in zip(bars1, values):
    height = bar.get_height()
    ax1.text(bar.get_x() + bar.get_width()/2., height + 0.5,
             f'{value}', ha='center', va='bottom', fontweight='bold')

# Horizontal bar chart
bars2 = ax2.barh(categories, values, color=colors, edgecolor='black', linewidth=1.2)
ax2.set_title("Horizontal Bar Chart", fontsize=14, fontweight='bold')
ax2.set_xlabel("Values", fontsize=12)
ax2.set_ylabel("Categories", fontsize=12)

# Add value labels on bars
for bar, value in zip(bars2, values):
    width = bar.get_width()
    ax2.text(width + 0.5, bar.get_y() + bar.get_height()/2.,
             f'{value}', ha='left', va='center', fontweight='bold')

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

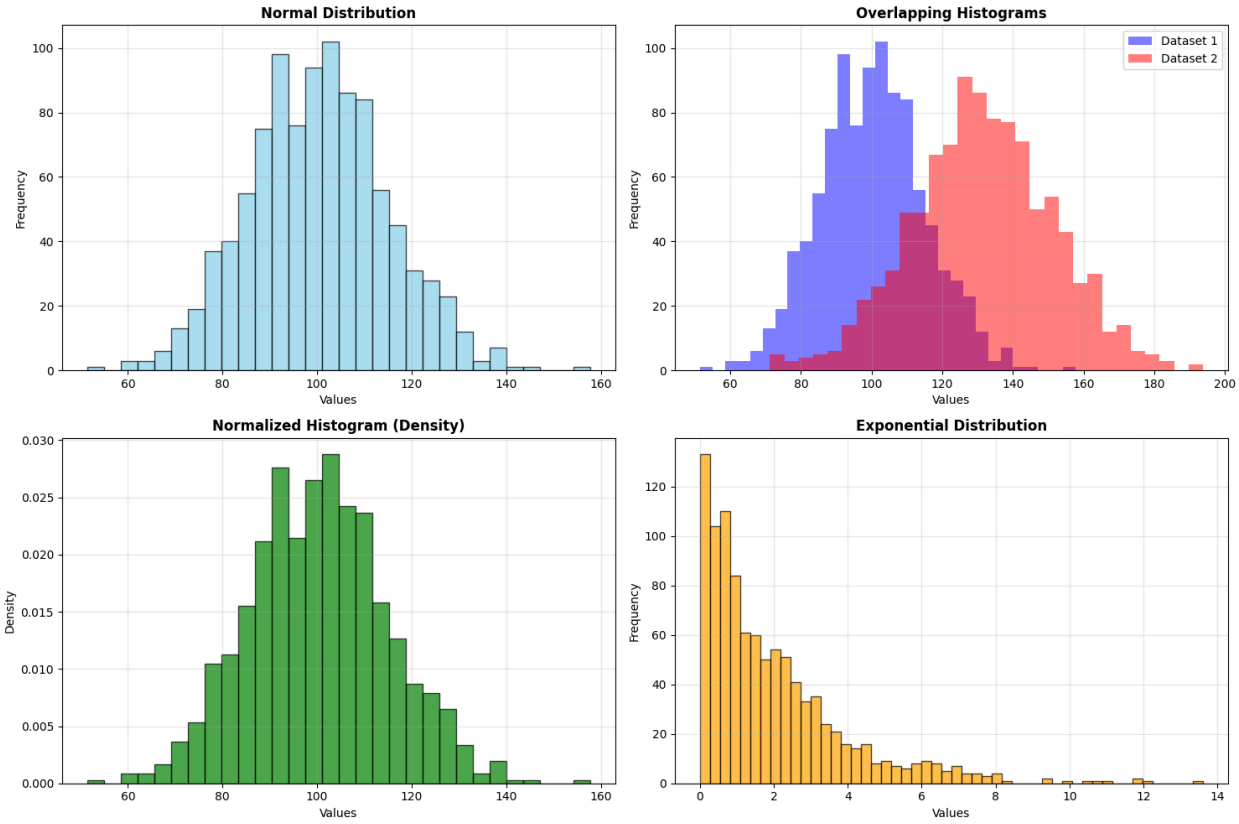


Histograms

Use Case: Distribution analysis, frequency analysis, data exploration

```
# Generate sample data
np.random.seed(42)
data1 = np.random.normal(100, 15, 1000)
```

```
data2 = np.random.normal(130, 20, 1000)
data3 = np.random.exponential(2, 1000)

# Create histogram
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Single histogram
axes[0, 0].hist(data1, bins=30, color='skyblue', edgecolor='black', alpha=0.7)
axes[0, 0].set_title("Normal Distribution", fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel("Values")
axes[0, 0].set_ylabel("Frequency")
axes[0, 0].grid(True, alpha=0.3)

# Multiple histograms
axes[0, 1].hist(data1, bins=30, alpha=0.5, label='Dataset 1', color='blue')
axes[0, 1].hist(data2, bins=30, alpha=0.5, label='Dataset 2', color='red')
axes[0, 1].set_title("Overlapping Histograms", fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel("Values")
axes[0, 1].set_ylabel("Frequency")
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Normalized histogram (density)
axes[1, 0].hist(data1, bins=30, density=True, color='green', alpha=0.7, edgecolor='black')
axes[1, 0].set_title("Normalized Histogram (Density)", fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel("Values")
axes[1, 0].set_ylabel("Density")
axes[1, 0].grid(True, alpha=0.3)

# Exponential distribution
axes[1, 1].hist(data3, bins=50, color='orange', edgecolor='black', alpha=0.7)
axes[1, 1].set_title("Exponential Distribution", fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel("Values")
axes[1, 1].set_ylabel("Frequency")
axes[1, 1].grid(True, alpha=0.3)

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

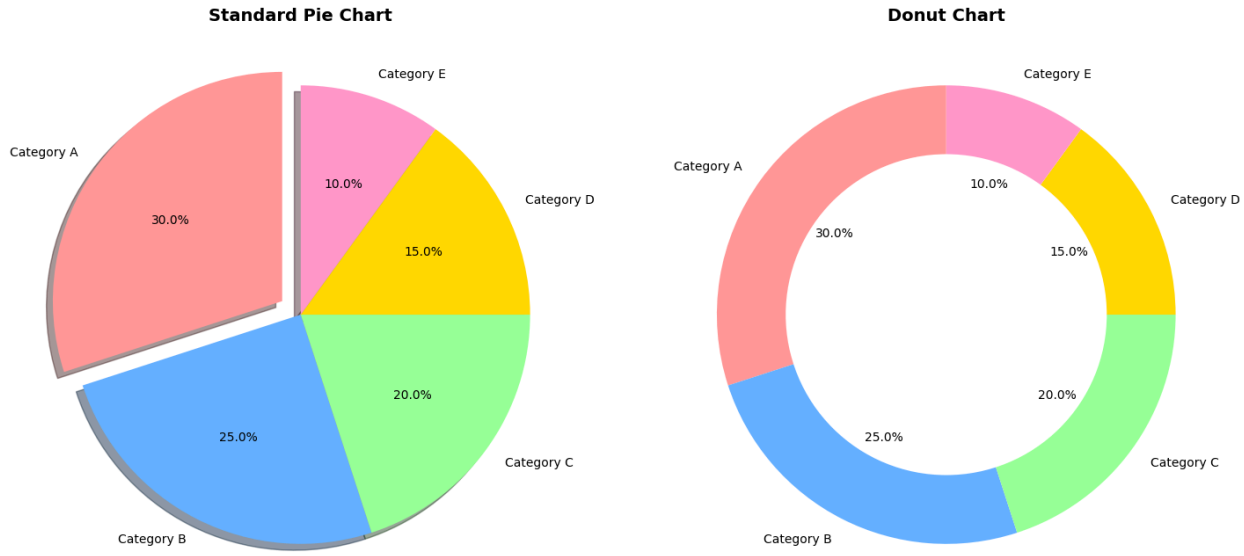


Pie Charts

Use Case: Proportional data, market share, budget allocation

```
# Sample data
sizes = [30, 25, 20, 15, 10]
labels = ['Category A', 'Category B', 'Category C', 'Category D', 'Category E']
colors = ['#FF9999', '#66B2FF', '#99FF99', '#FFD700', '#FF99CC']
explode = (0.1, 0, 0, 0, 0)  # explode first slice

# Create pie chart
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))

# Standard pie chart
ax1.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.set_title("Standard Pie Chart", fontsize=14, fontweight='bold')

# Donut chart
wedges, texts, autotexts = ax2.pie(sizes, labels=labels, colors=colors,
                                   autopct='%1.1f%%', startangle=90)
# Create donut by adding a white circle in the center
```

```
centre_circle = plt.Circle((0,0), 0.70, fc='white')
ax2.add_artist(centre_circle)
ax2.set_title("Donut Chart", fontsize=14, fontweight='bold')

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

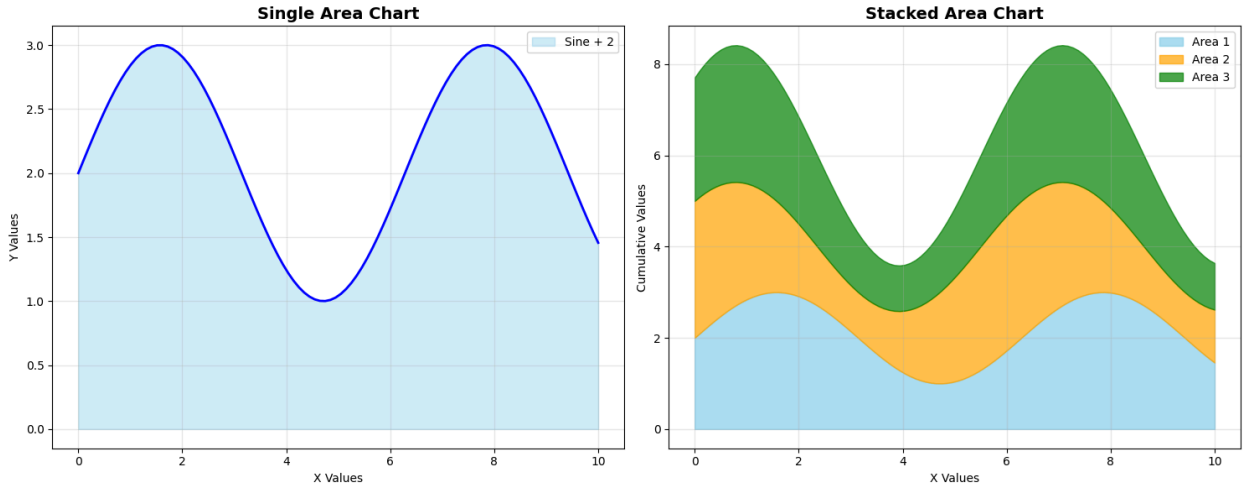


Area Charts

Use Case: Cumulative data, stacked proportions, trend emphasis

```
# Sample data
x = np.linspace(0, 10, 100)
y1 = np.sin(x) + 2
y2 = np.cos(x) + 2
y3 = np.sin(x + np.pi/4) + 2

# Create area chart
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Single area chart
ax1.fill_between(x, y1, color="skyblue", alpha=0.4, label='Sine + 2')
ax1.plot(x, y1, color="blue", linewidth=2)
ax1.set_title("Single Area Chart", fontsize=14, fontweight='bold')
ax1.set_xlabel("X Values")
ax1.set_ylabel("Y Values")
ax1.legend()
ax1.grid(True, alpha=0.3)
```

```
# Stacked area chart
ax2.fill_between(x, 0, y1, color="skyblue", alpha=0.7, label='Area 1')
ax2.fill_between(x, y1, y1+y2, color="orange", alpha=0.7, label='Area 2')
ax2.fill_between(x, y1+y2, y1+y2+y3, color="green", alpha=0.7, label='Area 3')
ax2.set_title("Stacked Area Chart", fontsize=14, fontweight='bold')
ax2.set_xlabel("X Values")
ax2.set_ylabel("Cumulative Values")
ax2.legend()
ax2.grid(True, alpha=0.3)

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

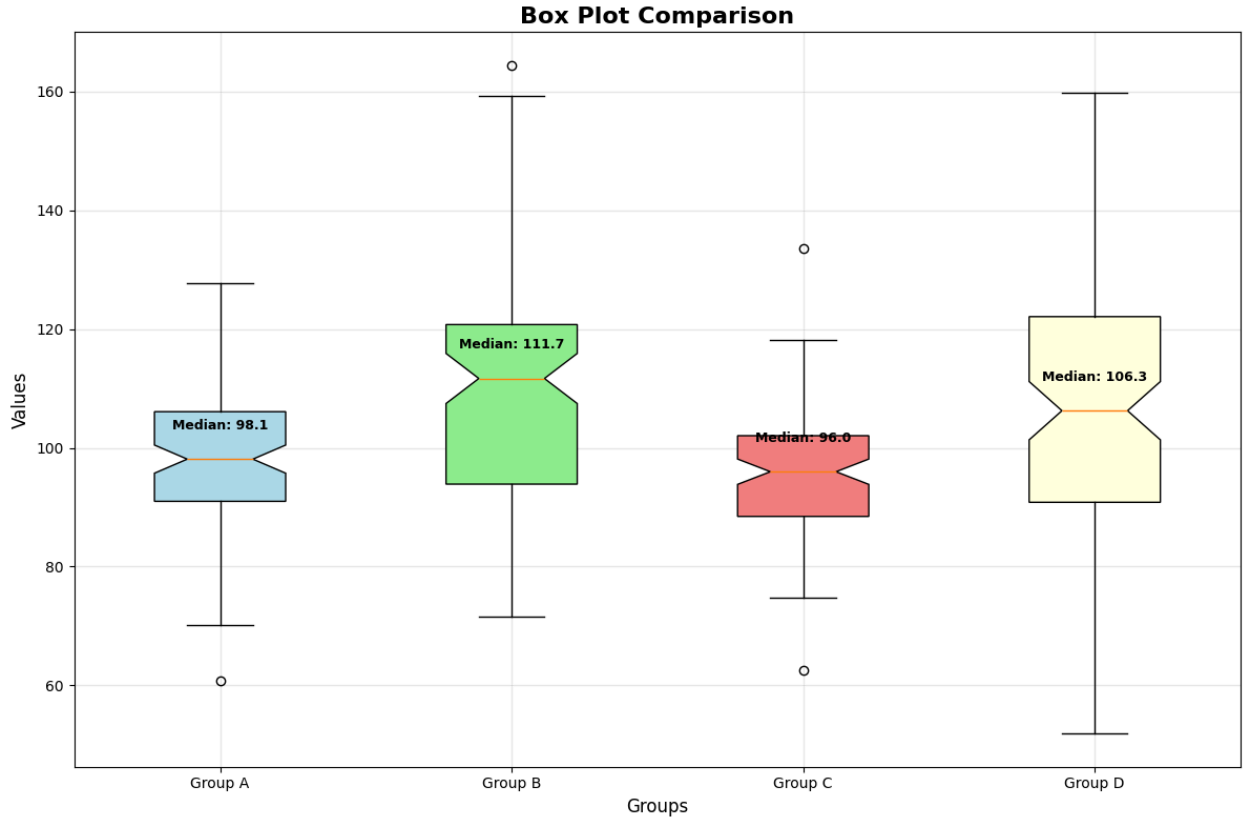


Box Plots

Use Case: Distribution comparison, outlier detection, statistical summary

```
# Generate sample data
np.random.seed(42)
data = [np.random.normal(100, 15, 100),
        np.random.normal(110, 20, 100),
        np.random.normal(95, 10, 100),
        np.random.normal(105, 25, 100)]

labels = ['Group A', 'Group B', 'Group C', 'Group D']

# Create box plot
plt.figure(figsize=(12, 8))
box_plot = plt.boxplot(data, labels=labels, patch_artist=True, notch=True)
```

```
# Customize colors
colors = ['lightblue', 'lightgreen', 'lightcoral', 'lightyellow']
for patch, color in zip(box_plot['boxes'], colors):
    patch.set_facecolor(color)

plt.title("Box Plot Comparison", fontsize=16, fontweight='bold')
plt.xlabel("Groups", fontsize=12)
plt.ylabel("Values", fontsize=12)
plt.grid(True, alpha=0.3)

# Add statistical annotations
for i, dataset in enumerate(data):
    median = np.median(dataset)
    mean = np.mean(dataset)
    plt.text(i+1, median + 5, f'Median: {median:.1f}',
             ha='center', fontsize=9, fontweight='bold')

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

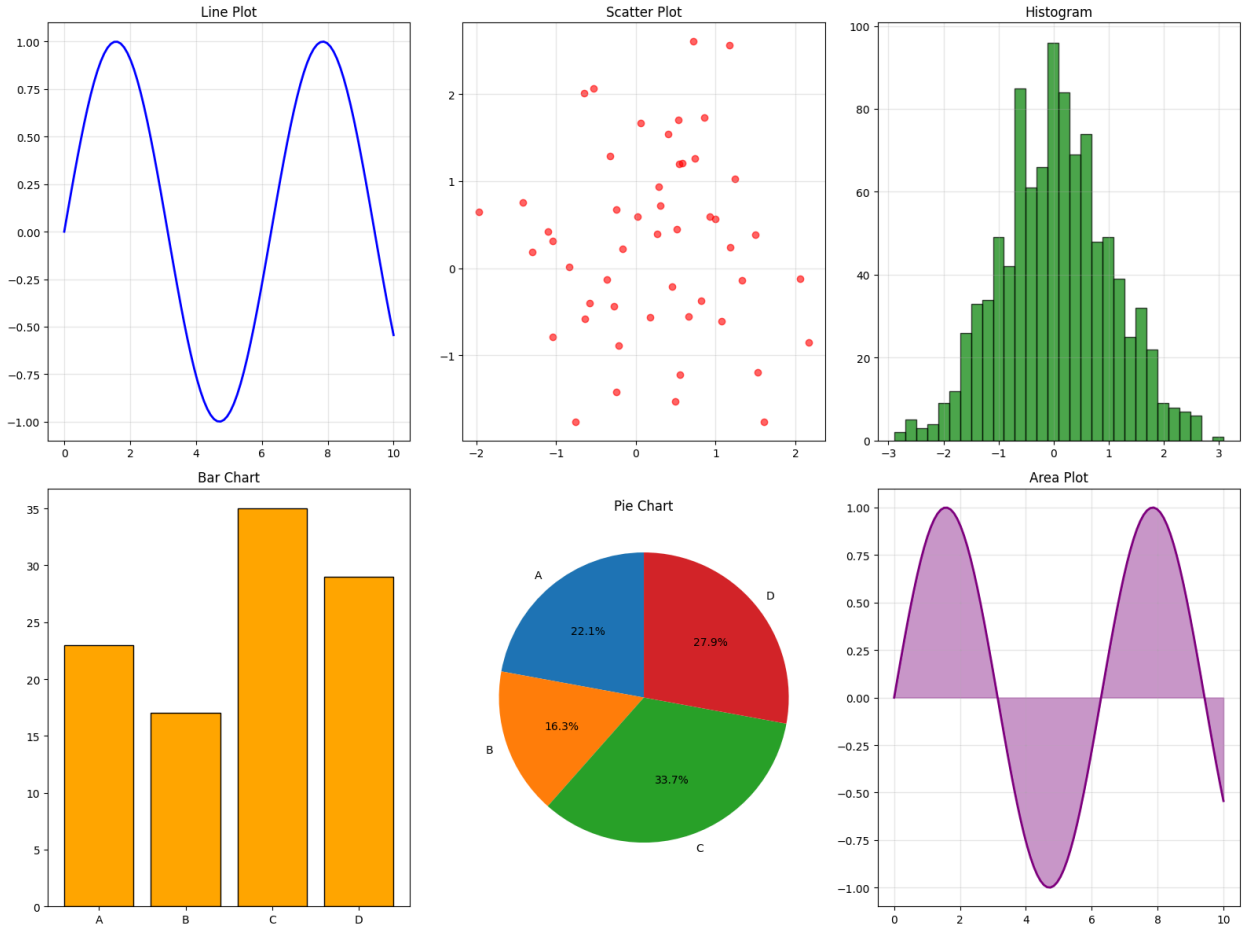


Subplots and Complex Layouts

Use Case: Multiple related visualizations, dashboard creation

```
# Create complex subplot layout
fig = plt.figure(figsize=(16, 12))

# Data preparation
x = np.linspace(0, 10, 100)
y = np.sin(x)
data = np.random.randn(1000)

# Subplot 1: Line plot
ax1 = plt.subplot(2, 3, 1)
plt.plot(x, y, 'b-', linewidth=2)
plt.title("Line Plot")
plt.grid(True, alpha=0.3)

# Subplot 2: Scatter plot
ax2 = plt.subplot(2, 3, 2)
x_scatter = np.random.randn(50)
y_scatter = np.random.randn(50)
plt.scatter(x_scatter, y_scatter, c='red', alpha=0.6)
plt.title("Scatter Plot")
plt.grid(True, alpha=0.3)

# Subplot 3: Histogram
ax3 = plt.subplot(2, 3, 3)
plt.hist(data, bins=30, color='green', alpha=0.7, edgecolor='black')
plt.title("Histogram")
plt.grid(True, alpha=0.3)

# Subplot 4: Bar chart
ax4 = plt.subplot(2, 3, 4)
categories = ['A', 'B', 'C', 'D']
values = [23, 17, 35, 29]
plt.bar(categories, values, color='orange', edgecolor='black')
plt.title("Bar Chart")

# Subplot 5: Pie chart
ax5 = plt.subplot(2, 3, 5)
plt.pie(values, labels=categories, autopct='%1.1f%%', startangle=90)
plt.title("Pie Chart")

# Subplot 6: Area plot
```

```
ax6 = plt.subplot(2, 3, 6)
plt.fill_between(x, y, color='purple', alpha=0.4)
plt.plot(x, y, color='purple', linewidth=2)
plt.title("Area Plot")
plt.grid(True, alpha=0.3)

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



3. Seaborn: Statistical Data Visualization

Seaborn is built on top of matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics.

3.1 Library Overview

Unique Features:

- Beautiful default styles and color palettes
- Built-in statistical plotting functions
- Excellent integration with pandas DataFrames
- Automatic handling of statistical relationships
- Simplified syntax for complex plots

Typical Use Cases:

- Statistical analysis and reporting
- Data exploration and EDA
- Academic and research publications
- Business intelligence dashboards

3.2 Dataset Preparation

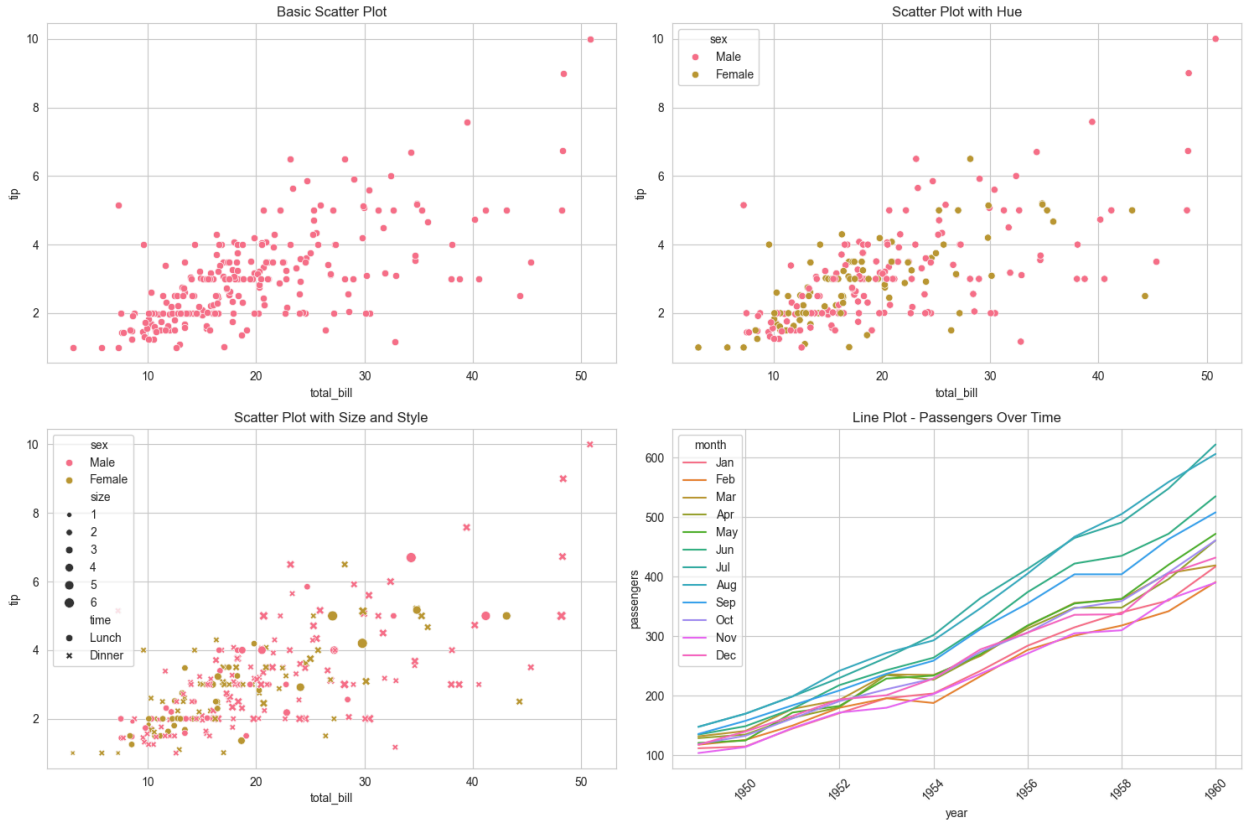
```
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Set style
sns.set_style("whitegrid")
sns.set_palette("husl")

# Load built-in datasets for examples
tips = sns.load_dataset("tips")
flights = sns.load_dataset("flights")
iris = sns.load_dataset("iris")
titanic = sns.load_dataset("titanic")
```

3.3 Graph Types and Examples

Relational Plots

Use Case: Exploring relationships between variables

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

# Load example datasets
tips = sns.load_dataset("tips")
flights = sns.load_dataset("flights")

# Scatter plot with multiple dimensions
plt.figure(figsize=(15, 10))

# Basic scatter plot
plt.subplot(2, 2, 1)
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.title("Basic Scatter Plot")

# Scatter plot with hue
plt.subplot(2, 2, 2)
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="sex")
plt.title("Scatter Plot with Hue")

# Scatter plot with size and style
plt.subplot(2, 2, 3)
sns.scatterplot(data=tips, x="total_bill", y="tip",
                hue="sex", size="size", style="time")
plt.title("Scatter Plot with Size and Style")

# Line plot for time series
plt.subplot(2, 2, 4)
flights_pivot = flights.pivot(index="month", columns="year", values="passengers")
sns.lineplot(data=flights, x="year", y="passengers", hue="month")
plt.title("Line Plot - Passengers Over Time")
plt.xticks(rotation=45)

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

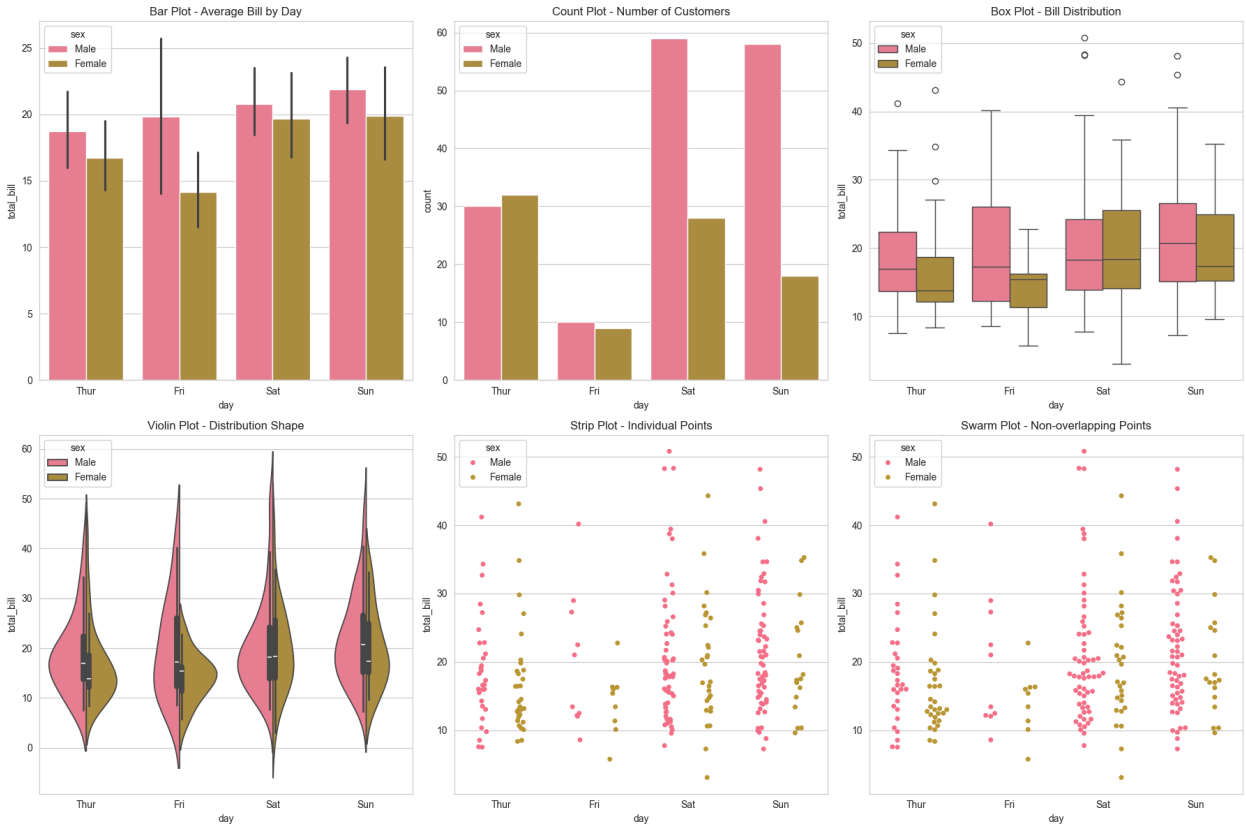


Categorical Plots

Use Case: Analyzing categorical data and group comparisons

```
plt.figure(figsize=(18, 12))

# Bar plot
plt.subplot(2, 3, 1)
sns.barplot(data=tips, x="day", y="total_bill", hue="sex")
plt.title("Bar Plot - Average Bill by Day")

# Count plot
plt.subplot(2, 3, 2)
sns.countplot(data=tips, x="day", hue="sex")
plt.title("Count Plot - Number of Customers")

# Box plot
plt.subplot(2, 3, 3)
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex")
plt.title("Box Plot - Bill Distribution")

# Violin plot
```

```
plt.subplot(2, 3, 4)
sns.violinplot(data=tips, x="day", y="total_bill", hue="sex", split=True)
plt.title("Violin Plot - Distribution Shape")

# Strip plot
plt.subplot(2, 3, 5)
sns.stripplot(data=tips, x="day", y="total_bill", hue="sex", dodge=True, jitter=True)
plt.title("Strip Plot - Individual Points")

# Swarm plot
plt.subplot(2, 3, 6)
sns.swarmplot(data=tips, x="day", y="total_bill", hue="sex", dodge=True)
plt.title("Swarm Plot - Non-overlapping Points")

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

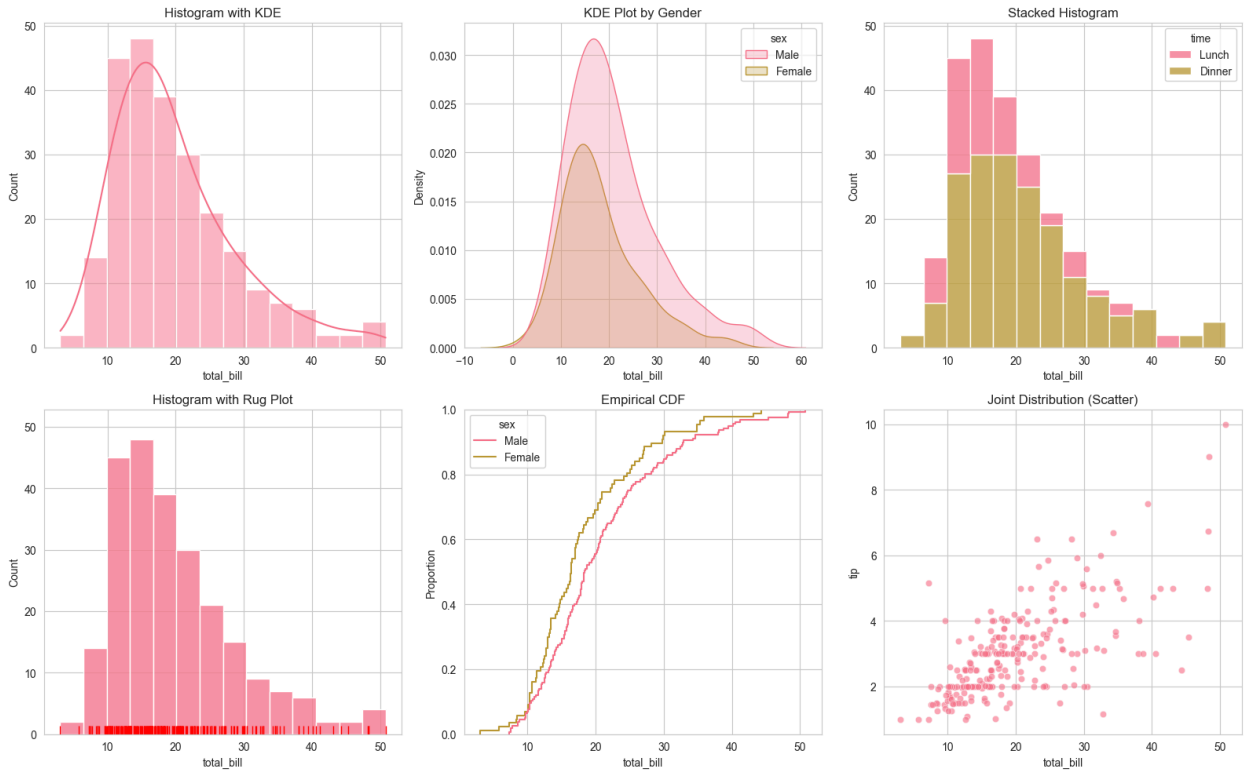


Distribution Plots

Use Case: Understanding data distributions and densities

```
plt.figure(figsize=(16, 10))
```

```
# Histogram with KDE
plt.subplot(2, 3, 1)
sns.histplot(data=tips, x="total_bill", kde=True)
plt.title("Histogram with KDE")

# KDE plot
plt.subplot(2, 3, 2)
sns.kdeplot(data=tips, x="total_bill", hue="sex", fill=True)
plt.title("KDE Plot by Gender")

# Distribution plot (deprecated but still useful)
plt.subplot(2, 3, 3)
sns.histplot(data=tips, x="total_bill", hue="time", multiple="stack")
plt.title("Stacked Histogram")

# Rug plot
plt.subplot(2, 3, 4)
sns.histplot(data=tips, x="total_bill")
sns.rugplot(data=tips, x="total_bill", color="red")
plt.title("Histogram with Rug Plot")

# ECDF plot
plt.subplot(2, 3, 5)
sns.ecdfplot(data=tips, x="total_bill", hue="sex")
plt.title("Empirical CDF")

# Joint distribution
plt.subplot(2, 3, 6)
# Note: jointplot creates its own figure, so we'll create a scatter instead
sns.scatterplot(data=tips, x="total_bill", y="tip", alpha=0.6)
plt.title("Joint Distribution (Scatter)")

plt.tight_layout()
plt.show()

# Separate joint plot
g = sns.jointplot(data=tips, x="total_bill", y="tip", kind="hex")
g.fig.suptitle("Joint Distribution with Hexbin", y=1.02)
plt.show()
```

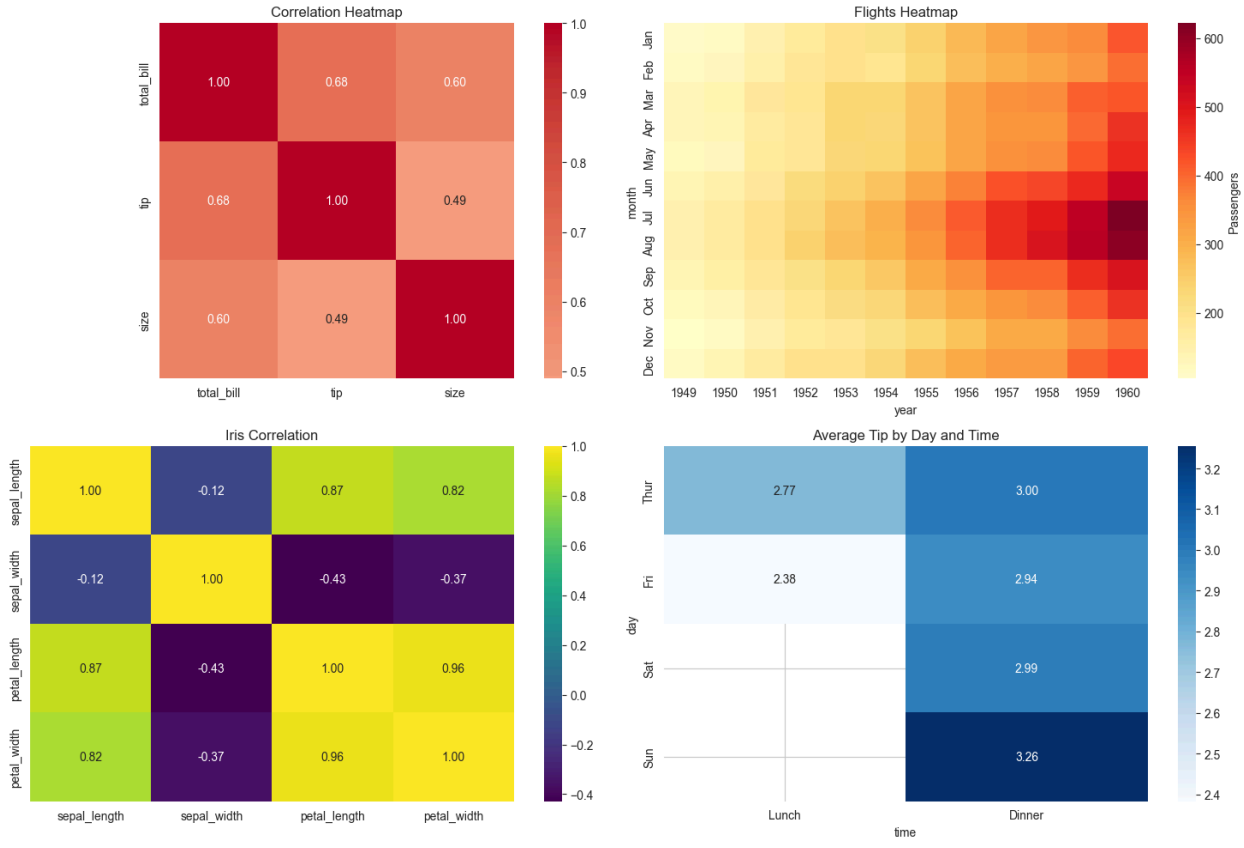


Matrix Plots

Use Case: Correlation analysis, heatmaps, pivot table visualization

```
# Prepare correlation matrix
numeric_tips = tips.select_dtypes(include=[np.number])
correlation_matrix = numeric_tips.corr()

# Create matrix plots
plt.figure(figsize=(15, 10))

# Correlation heatmap
plt.subplot(2, 2, 1)
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", center=0,
            square=True, fmt='.2f')
plt.title("Correlation Heatmap")

# Flights pivot heatmap
plt.subplot(2, 2, 2)
flights_pivot = flights.pivot("month", "year", "passengers")
sns.heatmap(flights_pivot, cmap="YlOrRd", cbar_kws={'label': 'Passengers'})
plt.title("Flights Heatmap")
```

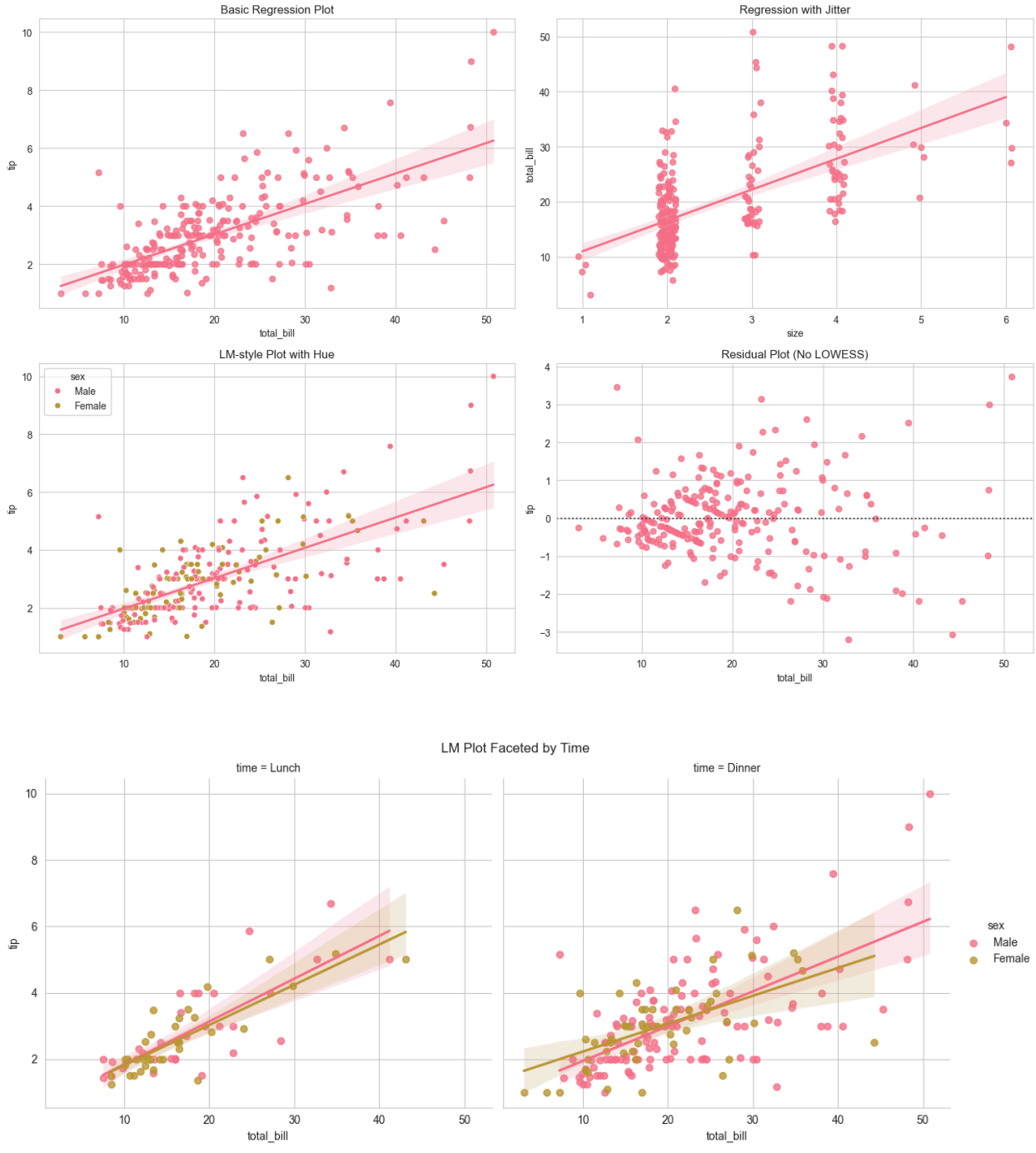
```
# Clustermap (creates its own figure)
plt.subplot(2, 2, 3)
# For demonstration, we'll create a simple heatmap here
sns.heatmap(iris.corr(), annot=True, cmap="viridis", fmt='.2f')
plt.title("Iris Correlation")

# Pivot table heatmap
plt.subplot(2, 2, 4)
tips_pivot = tips.pivot_table(values='tip', index='day', columns='time', aggfunc='mean')
sns.heatmap(tips_pivot, annot=True, cmap="Blues", fmt='.2f')
plt.title("Average Tip by Day and Time")

plt.tight_layout()
plt.show()

# Separate clustermap
g = sns.clustermap(correlation_matrix, annot=True, cmap="coolwarm", center=0)
g.fig.suptitle("Clustered Correlation Matrix", y=1.02)
plt.show()
```



Regression Plots

Use Case: Linear relationships, trend analysis, prediction intervals

```
plt.figure(figsize=(15, 10))

# Basic regression plot
plt.subplot(2, 2, 1)
sns.regplot(data=tips, x="total_bill", y="tip")
plt.title("Basic Regression Plot")

# Regression with categorical variable
plt.subplot(2, 2, 2)
sns.regplot(data=tips, x="size", y="total_bill", x_jitter=0.1)
plt.title("Regression with Jitter")

# LM plot with hue
plt.subplot(2, 2, 3)
sns.lmplot(data=tips, x="total_bill", y="tip", hue="sex", aspect=1.2)
plt.title("LM Plot with Hue")
plt.close()  # lmplot creates its own figure

# Residual plot
plt.subplot(2, 2, 4)
sns.residplot(data=tips, x="total_bill", y="tip", lowess=True)
plt.title("Residual Plot")

plt.tight_layout()
plt.show()

# Separate LM plot
g = sns.lmplot(data=tips, x="total_bill", y="tip", col="time", hue="sex")
g.fig.suptitle("LM Plot Faceted by Time", y=1.02)
plt.show()
```

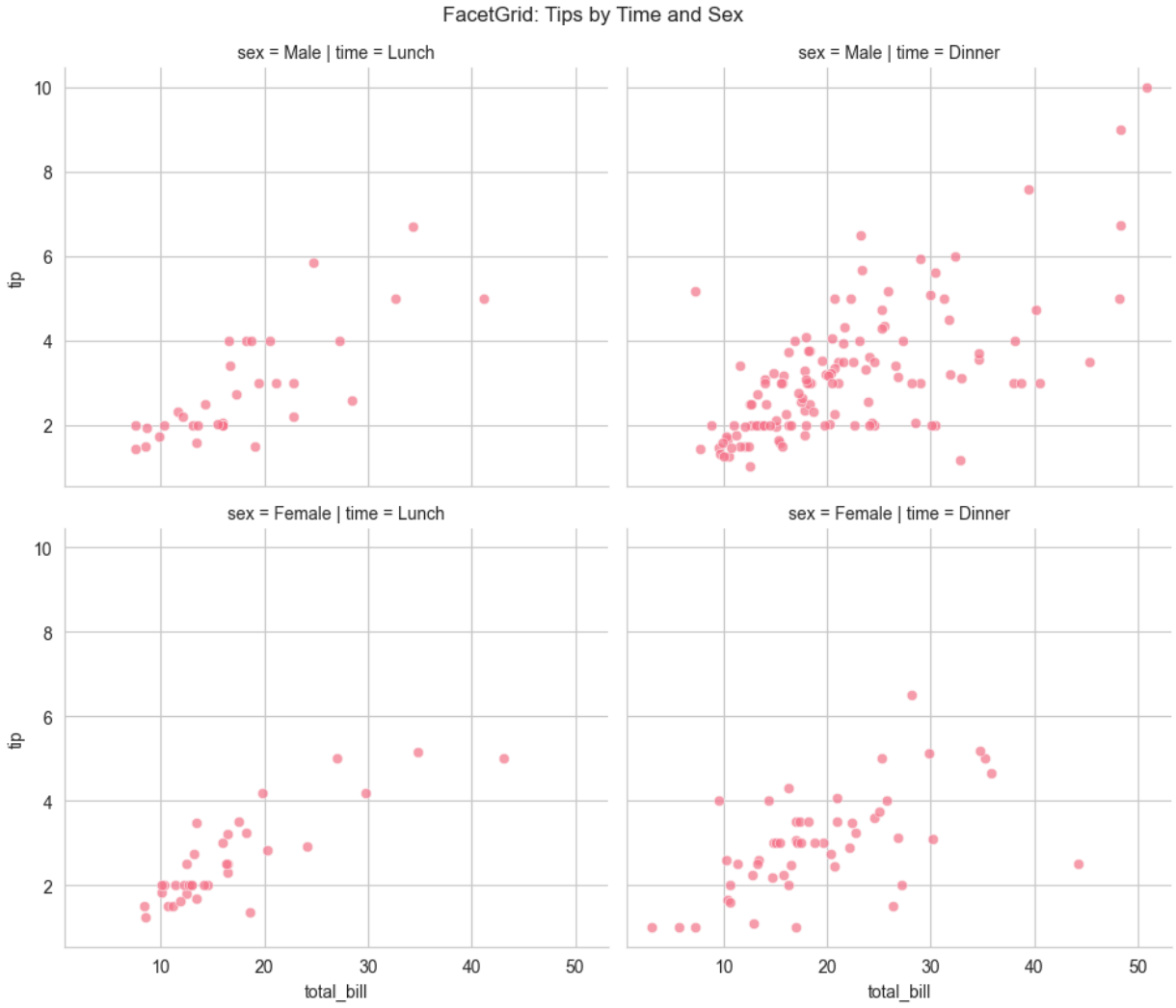


Multi-plot Grids

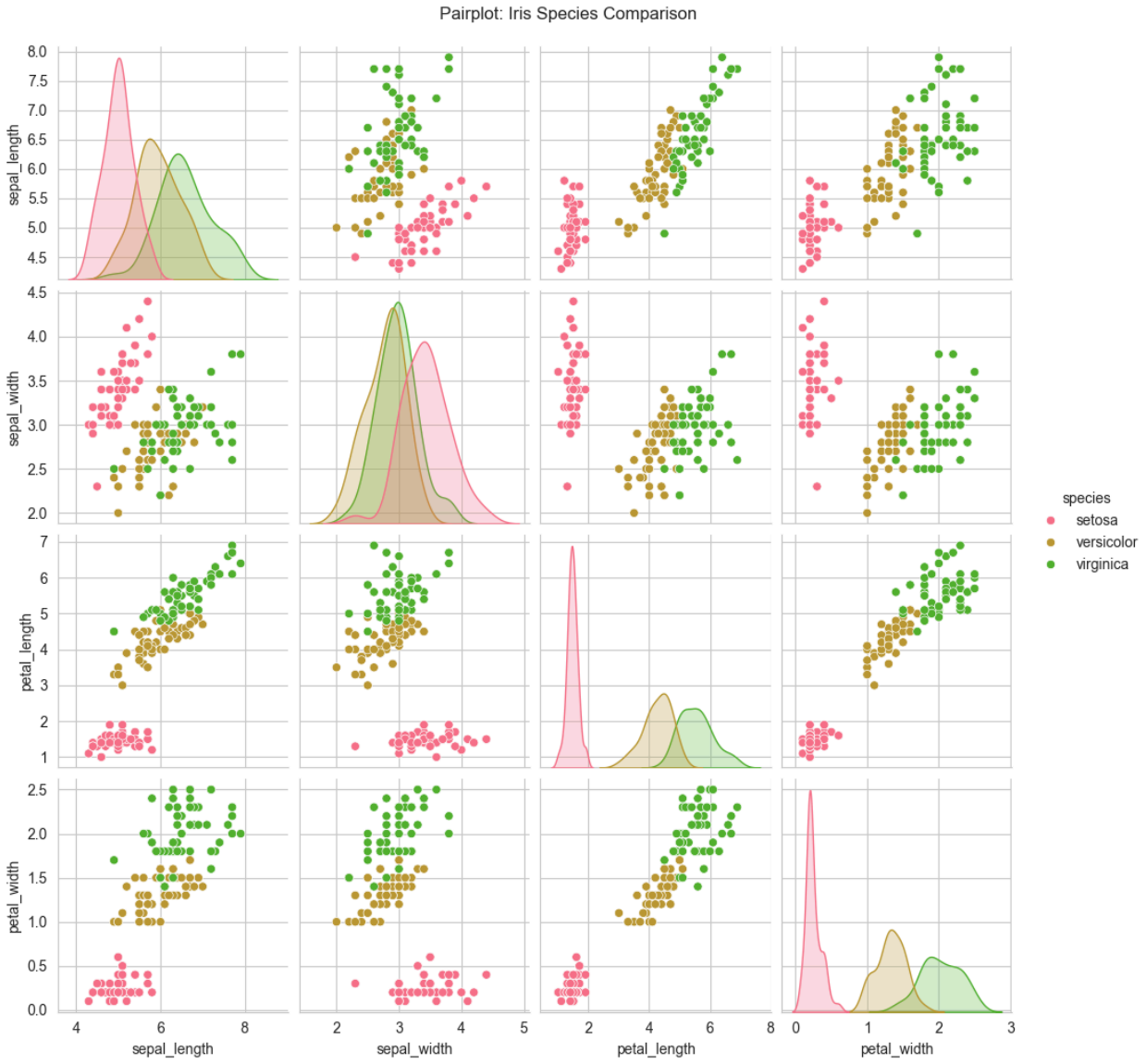
Use Case: Complex multi-dimensional analysis, faceted plots

```
# FacetGrid
g = sns.FacetGrid(tips, col="time", row="sex", height=4, aspect=1.2)
g.map(sns.scatterplot, "total_bill", "tip", alpha=0.7)
g.add_legend()
```

```
g.fig.suptitle("FacetGrid: Tips by Time and Sex", y=1.02)
plt.show()

# PairGrid
g = sns.PairGrid(iris, hue="species", height=3)
g.map_upper(sns.scatterplot)
g.map_lower(sns.kdeplot)
g.map_diag(sns.histplot)
g.add_legend()
g.fig.suptitle("PairGrid: Iris Dataset Analysis", y=1.02)
plt.show()

# Pairplot (simpler version of PairGrid)
g = sns.pairplot(iris, hue="species", diag_kind="kde", height=2.5)
g.fig.suptitle("Pairplot: Iris Species Comparison", y=1.02)
plt.show()
```



PairGrid: Iris Dataset Analysis



---

4. Library Comparison

4.1 Ease of Use

Library	Learning Curve	Syntax Complexity	Documentation
Matplotlib	Steep	High	Excellent
Seaborn	Moderate	Low	Good

Matplotlib:

- **Pros:** Complete control, extensive customization
- **Cons:** Verbose syntax, requires more code for complex plots
- **Best for:** Users who need precise control over every aspect

Seaborn:

- **Pros:** Minimal code for statistical plots, beautiful defaults
- **Cons:** Less customization than matplotlib, limited interactivity
- **Best for:** Statistical analysis, quick exploratory data analysis

4.2 Customization Options

Matplotlib:

- Unlimited customization possibilities
- Control over every pixel
- Custom themes and styles
- Publication-quality output

Seaborn:

- Built-in beautiful themes
- Easy color palette management
- Statistical plot specialization
- Good integration with matplotlib for further customization

4.3 Interactivity

Matplotlib:

- Limited built-in interactivity
- Requires additional libraries (widgets) for interaction
- Static output by default

Seaborn:

- Inherits matplotlib's interactivity limitations
- Primarily static visualizations
- Focus on statistical insight rather than interaction

4.4 Performance with Large Datasets

Matplotlib:

Good performance for static plots

- Memory efficient for large datasets
- Can handle millions of points with optimization

Seaborn:

- Performance depends on underlying matplotlib
- Some statistical computations can be slow

4.5 Output Formats

Matplotlib:

- PNG, PDF, SVG, EPS
- Print-ready quality

Seaborn:

- Inherits matplotlib formats
- Excellent for academic publications

4.6 Use Case Recommendations

Use Case	Best Library	Alternative
Academic Publications	Matplotlib	Seaborn
Quick Data Exploration	Seaborn	Plotly Express
Interactive Dashboards	Plotly	Bokeh
Web Applications	Plotly	Bokeh
Statistical Analysis	Seaborn	Matplotlib
Custom Visualizations	Matplotlib	Plotly
Real-time Data	Plotly	Bokeh
Large Datasets	Matplotlib	Bokeh

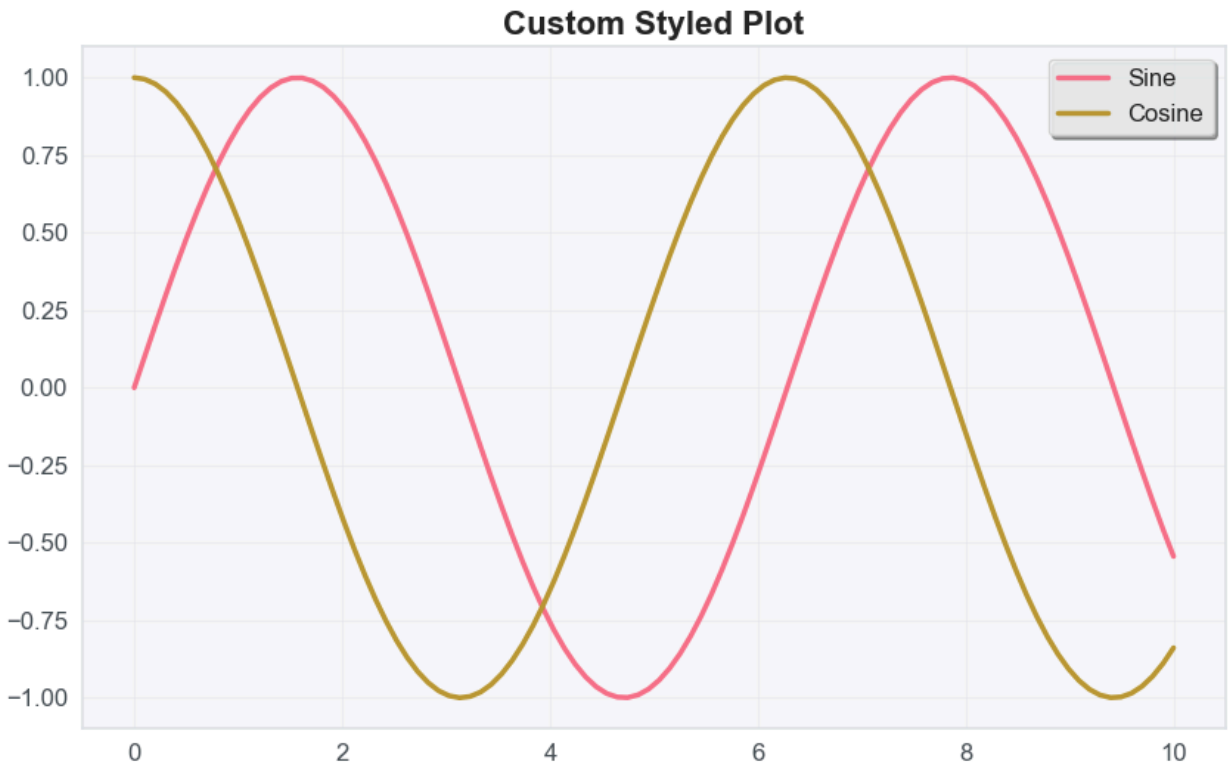
---

5. Advanced Techniques

5.1 Custom Themes and Styling

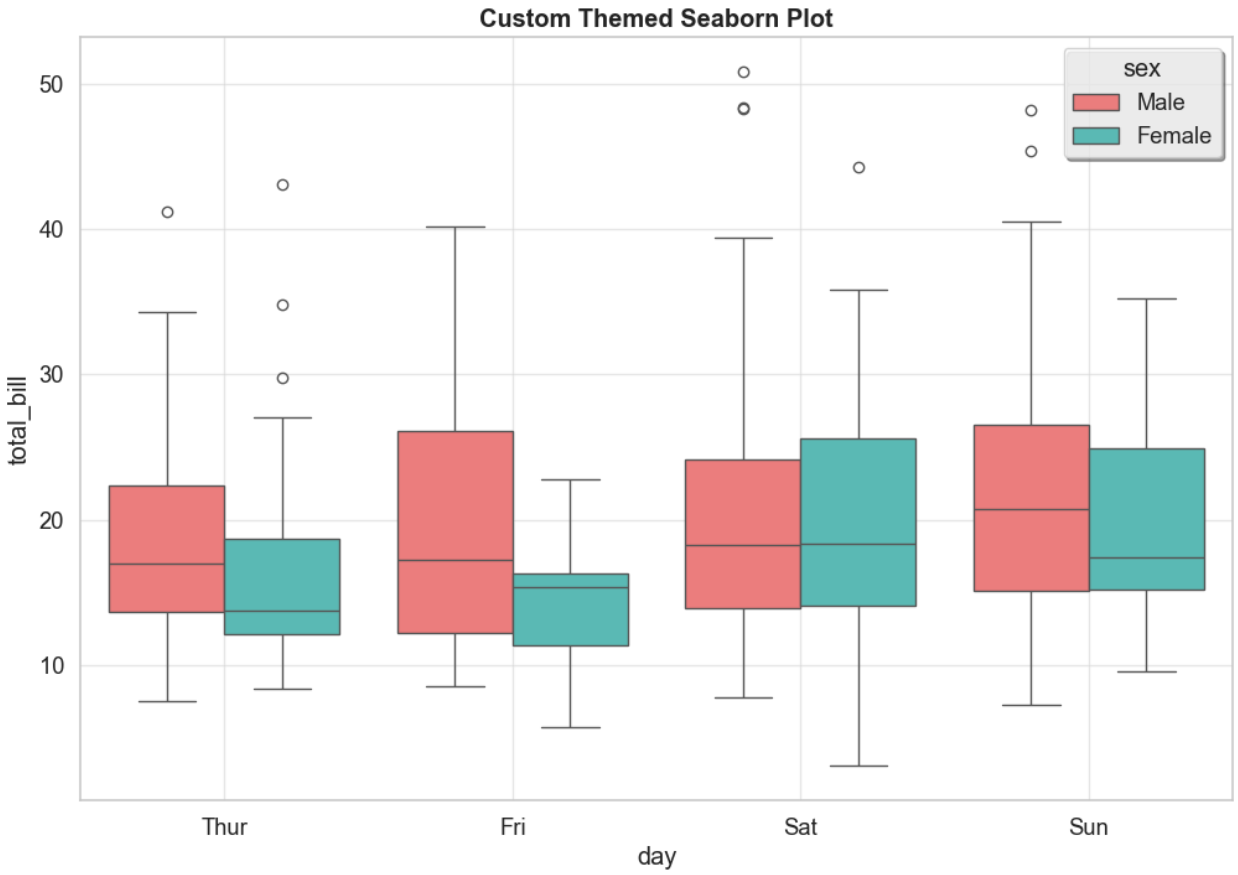
Matplotlib Custom Styling:

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

# Create custom style
custom_style = {
    'figure.facecolor': 'white',
    'axes.facecolor': '#f8f9fa',
    'axes.edgecolor': '#dee2e6',
    'axes.linewidth': 1.2,
    'axes.grid': True,
    'axes.grid.alpha': 0.4,
    'grid.linewidth': 0.8,
    'grid.color': '#dee2e6',
    'xtick.color': '#495057',
    'ytick.color': '#495057',
    'axes.labelcolor': '#495057',
    'axes.titlesize': 16,
    'axes.titleweight': 'bold',
    'font.size': 12,
    'legend.frameon': True,
    'legend.fancybox': True,
    'legend.shadow': True
}

# Apply custom style
plt.rcParams.update(custom_style)

# Example plot with custom style
fig, ax = plt.subplots(figsize=(10, 6))
x = np.linspace(0, 10, 100)
ax.plot(x, np.sin(x), label='Sine', linewidth=2.5)
ax.plot(x, np.cos(x), label='Cosine', linewidth=2.5)
ax.set_title('Custom Styled Plot')
ax.legend()
plt.show()
```



Seaborn Custom Themes:

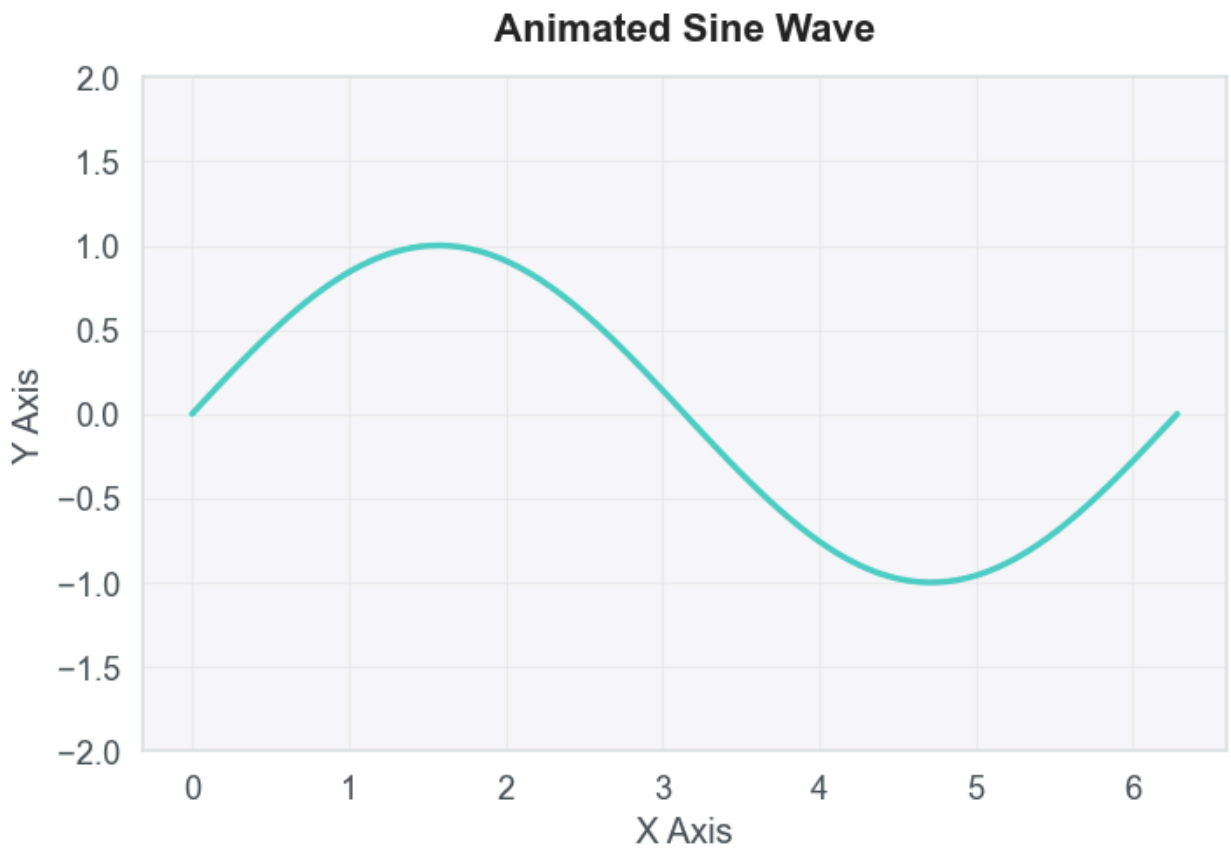
```
import seaborn as sns

# Create custom palette
custom_palette = ["#FF6B6B", "#4ECDC4", "#45B7D1", "#FFA07A", "#98D8C8"]
sns.set_palette(custom_palette)

# Custom theme
sns.set_theme(
    style="whitegrid",
    palette=custom_palette,
    font_scale=1.2,
    rc={"figure.figsize": (10, 6)}
)

# Example with custom theme
tips = sns.load_dataset("tips")
plt.figure(figsize=(12, 8))
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex")
plt.title("Custom Themed Seaborn Plot")
plt.show()
```



5.2 Animation and Interactive Features

Matplotlib Animation:

```
import matplotlib.animation as animation

# Create animated sine wave
fig, ax = plt.subplots()
x = np.linspace(0, 2*np.pi, 100)
line, = ax.plot(x, np.sin(x))
ax.set_ylim(-2, 2)

def animate(frame):
    line.set_ydata(np.sin(x + frame/10))
    return line,

anim = animation.FuncAnimation(fig, animate, frames=200, interval=50, blit=True)
# anim.save('sine_wave.gif', writer='pillow')  # Uncomment to save
plt.show()
```

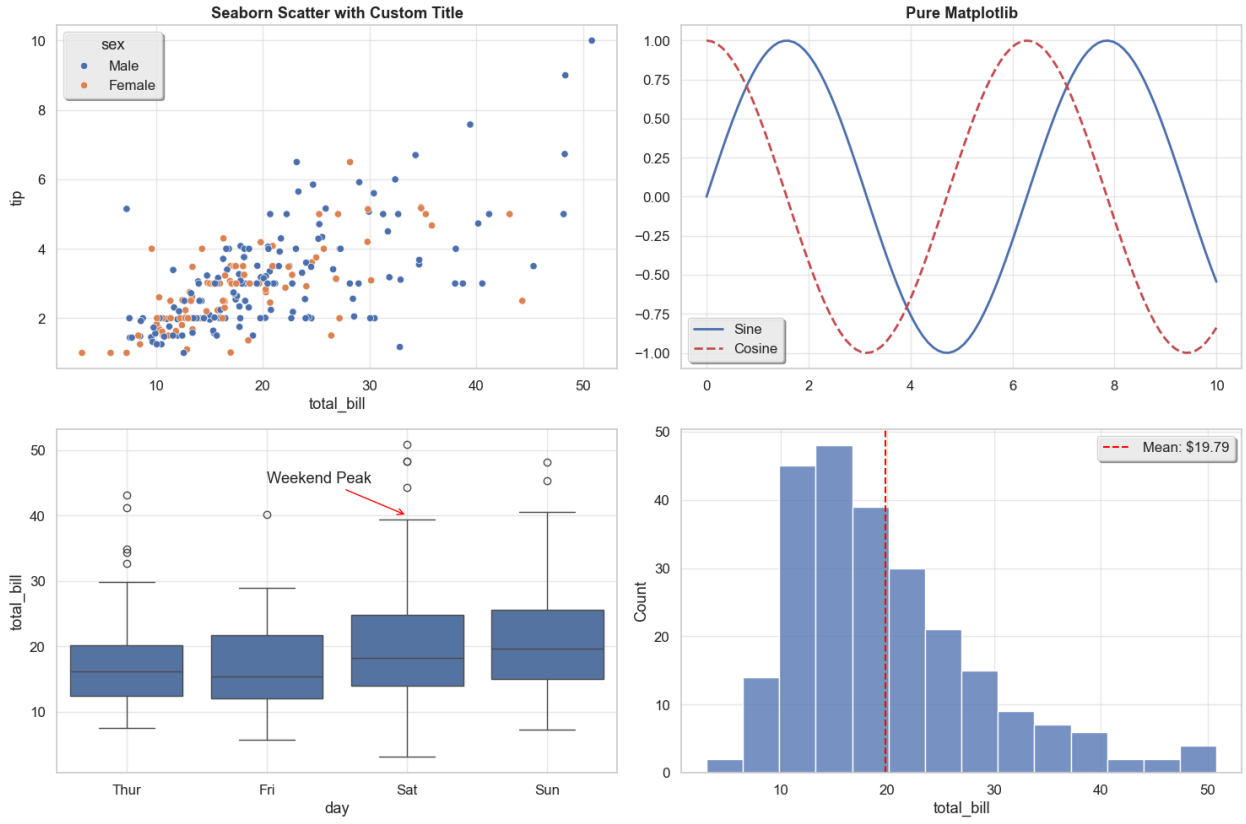


5.3 Integration Techniques

Combining Libraries:

```
# Using matplotlib for precise control with seaborn data
import matplotlib.pyplot as plt
import seaborn as sns

# Create figure with matplotlib
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Use seaborn for statistical plots
tips = sns.load_dataset("tips")

# Subplot 1: Seaborn plot with matplotlib customization
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="sex", ax=axes[0,0])
axes[0,0].set_title("Seaborn Scatter with Custom Title", fontsize=14, fontweight='bold')

# Subplot 2: Custom matplotlib plot
x = np.linspace(0, 10, 100)
axes[0,1].plot(x, np.sin(x), 'b-', linewidth=2, label='Sine')
```

```
axes[0,1].plot(x, np.cos(x), 'r--', linewidth=2, label='Cosine')
axes[0,1].set_title("Pure Matplotlib")
axes[0,1].legend()

# Subplot 3: Seaborn with matplotlib annotations
sns.boxplot(data=tips, x="day", y="total_bill", ax=axes[1,0])
axes[1,0].annotate('Weekend Peak', xy=('Sat', 40), xytext=(1, 45),
                   arrowprops=dict(arrowstyle='->', color='red'))

# Subplot 4: Combined statistical and custom elements
sns.histplot(data=tips, x="total_bill", ax=axes[1,1])
mean_bill = tips["total_bill"].mean()
axes[1,1].axvline(mean_bill, color='red', linestyle='--',
                  label=f'Mean: ${mean_bill:.2f}')
axes[1,1].legend()

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



6. Best Practices

6.1 Design Principles

Choose the Right Chart Type:

- **Line plots:** Time series, continuous data
- **Bar charts:** Categorical comparisons
- **Scatter plots:** Relationships between variables
- **Histograms:** Data distributions
- **Box plots:** Statistical summaries and outliers
- **Heatmaps:** Correlation and matrix data

Color Guidelines:

```
# Good color practices
# 1. Use colorblind-friendly palettes
colorblind_palette = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd']

# 2. Maintain contrast
high_contrast = ['#000000', '#E69F00', '#56B4E9', '#009E73', '#F0E442']

# 3. Use semantic colors
semantic_colors = {
    'positive': '#2E8B57',    # Sea Green
    'negative': '#DC143C',    # Crimson
    'neutral': '#708090',     # Slate Gray
    'warning': '#FF8C00'      # Dark Orange
}
```

Typography and Layout:

```
# Font size hierarchy
title_size = 16
subtitle_size = 14
axis_label_size = 12
tick_label_size = 10
annotation_size = 9

# Consistent spacing
plt.subplots_adjust(
```

```
    left=0.1,    # Left margin
    bottom=0.1,  # Bottom margin
    right=0.9,   # Right margin
    top=0.9,     # Top margin
    wspace=0.2,  # Width spacing between subplots
    hspace=0.3   # Height spacing between subplots
)
```

6.2 Performance Optimization

Matplotlib Performance Tips:

```
# 1. Use appropriate backends
import matplotlib
matplotlib.use('Agg')  # For non-interactive, faster rendering

# 2. Reduce data points for large datasets
def downsample_data(x, y, max_points=1000):
    if len(x) > max_points:
        indices = np.linspace(0, len(x)-1, max_points, dtype=int)
        return x[indices], y[indices]
    return x, y

# 3. Use rasterization for complex plots
plt.plot(x, y, rasterized=True)  # Rasterize complex elements

# 4. Batch updates
plt.ioff()  # Turn off interactive mode
# ... create multiple plots ...
plt.ion()   # Turn back on
plt.show()
```

Memory Management:

```
# Close figures to free memory
plt.close('all')  # Close all figures
plt.close(fig)    # Close specific figure

# Use context managers
with plt.style.context('seaborn'):
    plt.plot(x, y)
    plt.show()
```

6.3 Accessibility

Making Plots Accessible:

```
# 1. Add alternative text descriptions
plt.title("Sales Data: Steady growth from Q1 to Q4")
plt.xlabel("Quarter")
plt.ylabel("Sales (in thousands)")

# 2. Use patterns in addition to colors
plt.plot(x, y1, 'o-', label='Series 1')
plt.plot(x, y2, 's--', label='Series 2')
plt.plot(x, y3, '^:', label='Series 3')

# 3. Ensure sufficient contrast
# Test your colors at https://webaim.org/resources/contrastchecker/

# 4. Provide data tables for screen readers
def create_accessible_plot(x, y, title="Plot"):
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.set_title(title)

    # Create data summary
    summary = f"""
    Data Summary for {title}:
    - X range: {x.min():.2f} to {x.max():.2f}
    - Y range: {y.min():.2f} to {y.max():.2f}
    - Data points: {len(x)}
    """
    print(summary)

    return fig, ax
```

6.4 Code Organization

Modular Plotting Functions:

```
def create_comparison_plot(data, x_col, y_col, group_col,
                           plot_type='scatter', figsize=(10, 6)):
    """
    Create a comparison plot with consistent styling.

    Parameters:
```

```
    ----------
    data : DataFrame
        The data to plot
    x_col : str
        Column name for x-axis
    y_col : str
        Column name for y-axis
    group_col : str
        Column name for grouping
    plot_type : str
        Type of plot ('scatter', 'line', 'bar')
    figsize : tuple
        Figure size

    Returns:
    --------
    fig, ax : matplotlib figure and axes objects
    """
    fig, ax = plt.subplots(figsize=figsize)

    # Apply consistent styling
    plt.style.use('seaborn-v0_8-whitegrid')

    if plot_type == 'scatter':
        for group in data[group_col].unique():
            group_data = data[data[group_col] == group]
            ax.scatter(group_data[x_col], group_data[y_col],
                       label=group, alpha=0.7, s=60)

    elif plot_type == 'line':
        for group in data[group_col].unique():
            group_data = data[data[group_col] == group]
            ax.plot(group_data[x_col], group_data[y_col],
                    label=group, linewidth=2)

    # Consistent formatting
    ax.set_xlabel(x_col.replace('_', ' ').title())
    ax.set_ylabel(y_col.replace('_', ' ').title())
    ax.legend()
    ax.grid(True, alpha=0.3)

    plt.tight_layout()
    return fig, ax
```

```
# Usage example
# fig, ax = create_comparison_plot(tips, 'total_bill', 'tip', 'sex')
```

Configuration Management:

```
# plotting_config.py
class PlottingConfig:
    # Color schemes
    COLORS = {
        'primary': '#1f77b4',
        'secondary': '#ff7f0e',
        'success': '#2ca02c',
        'danger': '#d62728',
        'warning': '#ff7f0e',
        'info': '#17a2b8'
    }

    # Font sizes
    FONT_SIZES = {
        'title': 16,
        'subtitle': 14,
        'label': 12,
        'tick': 10,
        'annotation': 9
    }

    # Default figure size
    FIGURE_SIZE = (10, 6)

    # Style parameters
    STYLE_PARAMS = {
        'axes.grid': True,
        'grid.alpha': 0.3,
        'axes.spines.top': False,
        'axes.spines.right': False,
        'figure.autolayout': True
    }

# Apply configuration
config = PlottingConfig()
plt.rcParams.update(config.STYLE_PARAMS)
```

---

7. Resources

7.1 Official Documentation

Matplotlib:

- Official Documentation: https://matplotlib.org/stable/users/explain/quick_start.html
- Gallery: https://matplotlib.org/stable/gallery/index.html
- Tutorials: https://matplotlib.org/stable/tutorials/index.html

Seaborn:

- Official Documentation: https://seaborn.pydata.org/tutorial/introduction.html
- Gallery: https://seaborn.pydata.org/examples/index.html
- API Reference: https://seaborn.pydata.org/api.html

7.2 Additional Libraries Worth Exploring

Plotly:

- Documentation: https://plotly.com/python/
- Best for: Interactive and publication-quality plots, dashboards, and data apps
- Strengths: Built-in interactivity, wide range of chart types (3D, maps, animations)

Bokeh:

- Documentation: https://docs.bokeh.org/en/latest/docs/user_guide/basic.html
- Best for: Large datasets, interactive web applications
- Strengths: Server applications, real-time streaming

Pandas Plotting:

- Documentation: https://pandas.pydata.org/docs/user_guide/index.html
- Best for: Quick exploratory plots integrated with data analysis
- Strengths: Direct DataFrame integration, minimal syntax

Altair:

- Grammar of graphics approach
- Declarative statistical visualization
- Excellent for data exploration

7.3 Learning Path Recommendations

Beginner:

1. Start with pandas plotting for basic exploration
2. Learn matplotlib fundamentals
3. Move to seaborn for statistical plots
4. Experiment with plotly for interactivity

Intermediate:

1. Master matplotlib customization
2. Create complex seaborn visualizations
3. Build interactive dashboards with plotly
4. Learn performance optimization techniques

Advanced:

1. Create custom visualization libraries
2. Integrate multiple libraries effectively
3. Build real-time visualization systems
4. Develop accessibility-compliant visualizations

7.4 Common Troubleshooting

Installation Issues:

```
# Complete installation
pip install matplotlib seaborn plotly pandas numpy

# For Jupyter notebooks
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension

# For plotly in Jupyter
pip install "notebook>=5.3" "ipywidgets>=7.2"
```

Common Errors and Solutions:

1. **"Figure is too large" warning:**

```
# Reduce figure size or increase memory
plt.figure(figsize=(8, 6))  # Smaller size
# Or increase matplotlib memory limit
plt.rcParams['figure.max_open_warning'] = 50
```

od
fig.show()

2. **Seaborn plot not showing:**

```
# Always call plt.show() after seaborn plots
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.show()
```

---

8. Conclusion

This comprehensive guide has explored the fundamental concepts and practical techniques of data visualization in Python, focusing on two of the most widely used and powerful libraries: Matplotlib and Seaborn. Each of these libraries plays a unique role in the visualization workflow, and understanding when and how to use them is key to effective data communication.

- **Matplotlib** excels in offering maximum flexibility and control over every element of a visualization. It is particularly well-suited for creating publication-quality, highly customized static visualizations. Whether you need to fine-tune figure sizes, axis ticks, labels, or create multi-panel plots, Matplotlib allows for complete customization. Its versatility makes it a foundational tool for Python data visualization, especially for detailed, professional reporting.

- **Seaborn**, on the other hand, is designed to simplify the process of creating beautiful, statistically rich plots with minimal code. It is built on top of Matplotlib but offers a higher-level interface, making it ideal for exploratory data analysis (EDA) and quick pattern discovery. With built-in support for complex statistical visualizations, Seaborn enables analysts and data scientists to move from raw data to insights much faster.

In practice, many analysts use these libraries together — leveraging Seaborn for quick, elegant statistical plots and Matplotlib for precise customization and fine-tuning. Mastering both tools empowers you to create visualizations that are not only aesthetically pleasing but also informative, accurate, and tailored to the needs of your audience.

Ultimately, data visualization is more than just creating attractive charts — it is about telling a story with data. The right visualization bridges the gap between raw numbers and meaningful insights, enabling better decision-making and clearer communication. By combining the strengths of Matplotlib and Seaborn, you have the tools to effectively present complex datasets in a way that is both visually engaging and analytically powerful.