Lecture 21 Handout

Data Visualization: Matplotlib and Seaborn

Transforming Data into Visual Insights

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 7 - Array-Oriented Programming (Visualization sections)
Reference: Matplotlib documentation at matplotlib.org, Seaborn documentation at seaborn.pydata.org

Learning Objectives

By the end of this lecture, you will be able to:

1. **Understand visualization principles** and choose appropriate chart types for different data and questions
2. **Create basic plots with Matplotlib** using the pyplot interface for quick visualizations
3. **Build line plots** to show trends and changes over time or continuous variables
4. **Create bar charts** to compare quantities across categorical groups
5. **Generate scatter plots** to reveal relationships and correlations between variables
6. **Produce histograms** to understand the distribution and spread of numeric data
7. **Customize plots** with titles, labels, colors, legends, and styling for clear communication
8. **Use Seaborn** for statistical visualizations with attractive default styling
9. **Save plots to files** in various formats for reports, presentations, and publications

Prerequisites Review

Building on Your Data Analysis Foundation:

From Lectures 1-11, you have Python fundamentals including loops, functions, and data structures. From Lectures 12-13, you mastered NumPy arrays with statistical operations, random number generation, and array mathematics.

From Lectures 16-19, you mastered pandas Series and DataFrames: creating and manipulating data structures, boolean indexing, statistical methods, reading and writing CSV files, and groupby aggregations.

From Lecture 20, you learned advanced aggregation: multi-column groupby, pivot tables, and crosstabs that summarize data into tables perfect for visualization.

You now have all the data preparation skills. This lecture teaches you to transform your cleaned, aggregated data into visual representations that communicate insights clearly and compellingly.

Transformation Goal: Evolve from **working with numbers in tables and DataFrames** to **creating professional visualizations that reveal patterns, trends, and relationships** in your data for effective communication.

1 Part 1: Introduction to Data Visualization

1.1 Why Visualize Data?

Humans are visual creatures - we process images 60,000 times faster than text. A table of 1,000 numbers is incomprehensible, but a chart showing those same numbers reveals patterns instantly. Consider Anscombe's Quartet: four datasets with identical statistical properties (same mean, standard deviation, correlation) that look completely different when plotted. Statistics alone can mislead; visualization reveals the truth.

Visualization serves multiple purposes in data analysis. First, exploration: when you first encounter a dataset, plotting helps you understand its structure, spot outliers, and identify patterns to investigate. Second, analysis: visualizations reveal relationships (correlations, clusters, trends) that statistics might miss. Third, communication: charts and graphs communicate findings to stakeholders who may not understand statistical tables. A well-designed visualization tells a story that numbers alone cannot tell.

The key insight is that visualization is not decoration added after analysis - it's an integral part of analysis itself. You should visualize early and often throughout your data work.

1.2 Choosing the Right Chart Type

Different chart types answer different questions. Choosing the wrong type obscures your message; choosing the right type makes it obvious. Here's a guide:

Line plots show trends over time or continuous variables. Use them when you have sequential data and want to show how values change. Examples: stock prices over months, temperature over hours, revenue over quarters.

Bar charts compare quantities across categories. Use them when you have categorical data and want to compare sizes. Examples: sales by region, grades by major, counts by category.

Scatter plots show relationships between two numeric variables. Use them to reveal correlations, clusters, or outliers. Examples: height vs weight, study hours vs test scores, advertising spend vs sales.

Histograms show the distribution of a single numeric variable. Use them to understand spread, shape, and central tendency. Examples: age distribution, income distribution, test score distribution.

Throughout this lecture, we'll create each type and discuss when to use them.

1.3 The Matplotlib Library

Matplotlib is Python's foundational plotting library, providing comprehensive tools for creating static, animated, and interactive visualizations. It's highly customizable - you can control every element of a plot - but this flexibility comes with complexity. We'll use the `pyplot` interface, which provides a MATLAB-like syntax that's simpler for common plots.

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

# The standard import convention
# plt is the pyplot interface to matplotlib
# We'll use this throughout the lecture

print("Matplotlib version:", plt.matplotlib.__version__)
```

The convention `import matplotlib.pyplot as plt` is universal in Python data science. When you see `plt.plot()`, `plt.bar()`, or `plt.show()`, you know it's matplotlib's pyplot interface.

1.4 Part 1 Exercise

Exercise: For each of the following scenarios, identify which chart type would be most appropriate and explain why: (1) Comparing quarterly sales across 5 regions, (2) Showing how a stock price changed over 30 days, (3) Investigating if there's a relationship between advertising spend and revenue, (4) Understanding the age distribution of your customers.

2 Part 2: Matplotlib Fundamentals

2.1 The Figure and Axes Model

Matplotlib uses a hierarchical structure: a `Figure` is the overall window or page, and `Axes` are the individual plots within that figure. A figure can contain one or many axes (subplots). Understanding this model helps you create complex multi-plot figures.

For simple single plots, pyplot handles the figure and axes automatically. For more control, you create them explicitly.

```python
# Simple approach: pyplot creates figure/axes automatically
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()

# Explicit approach: create figure and axes yourself
fig, ax = plt.subplots()  # Creates figure with one axes
ax.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()

# Multiple subplots
fig, (ax1, ax2) = plt.subplots(1, 2)  # 1 row, 2 columns
ax1.plot([1, 2, 3], [1, 2, 3])
ax2.plot([1, 2, 3], [1, 4, 9])
plt.show()
```

The simple approach is fine for quick exploration. The explicit approach gives you control over layout, sizing, and multiple plots. We'll use both throughout this lecture.

2.2 Basic Plot Anatomy

Every plot has standard components that you should understand and customize: the title (what the plot shows), axis labels (what each axis represents), tick marks and labels (the scale), legend (what different colors/styles mean), and the data itself.

```python
# Create data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create plot with all standard components
plt.figure(figsize=(8, 6))  # Set figure size (width, height in inches)

plt.plot(x, y)

plt.title('Simple Linear Relationship')  # What the plot shows
plt.xlabel('X Values')  # What x-axis represents
plt.ylabel('Y Values')  # What y-axis represents

plt.grid(True)  # Add grid lines for readability

plt.show()
```

Always include title and axis labels - a plot without labels is meaningless to anyone who wasn't watching you create it. The `figsize` parameter controls plot dimensions in inches.

2.3 Displaying and Saving Plots

In Jupyter notebooks, plots display automatically. In scripts, you need `plt.show()` to display the plot window. To save plots for reports or presentations, use `plt.savefig()`.

```python
# Create a simple plot
plt.plot([1, 2, 3], [1, 4, 9])
plt.title('Sample Plot')

# Save to file before showing
# Supports: PNG, PDF, SVG, JPG, and more
plt.savefig('my_plot.png', dpi=150, bbox_inches='tight')

# dpi: dots per inch (resolution)
# bbox_inches='tight': removes excess whitespace

plt.show()  # Display on screen

print("Plot saved to my_plot.png")
```

Always `savefig()` before `show()`, because `show()` clears the figure. Use PNG for web/screen, PDF for print quality, SVG for scalable vector graphics.

2.4 Part 2 Exercise

Exercise: Create a figure with figsize (10, 6) and plot the squares of numbers 1-10 (x = [1,2,...,10], y = [1,4,9,...,100]). Add a title "Square Numbers", label the x-axis "Number" and y-axis "Square". Add a grid and save the plot as 'squares.png' with dpi=150 before displaying it.

3 Part 3: Line Plots for Trends

3.1 Basic Line Plots

Line plots connect data points with lines, emphasizing the continuous nature of the data and making trends visible. They're ideal for time series data where you want to show how values change over time.

```python
# Monthly sales data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [12000, 15000, 18000, 22000, 25000, 28000]

plt.figure(figsize=(10, 6))
plt.plot(months, sales)

plt.title('Monthly Sales Trend - 2024')
plt.xlabel('Month')
plt.ylabel('Sales ($)')

plt.show()
```

The line clearly shows an upward trend in sales. You can immediately see that sales grew every month, with the steepest growth between March and April. This pattern would be hard to spot in a table of numbers.

3.2 Multiple Lines and Legends

To compare multiple trends, plot multiple lines on the same axes. Use different colors and a legend to distinguish them.

```python
# Compare sales across regions
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
east_sales = [12000, 15000, 18000, 22000, 25000, 28000]
west_sales = [10000, 12000, 15000, 18000, 20000, 22000]
north_sales = [8000, 9000, 11000, 13000, 15000, 17000]

plt.figure(figsize=(10, 6))

# Plot each region with label for legend
plt.plot(months, east_sales, label='East')
plt.plot(months, west_sales, label='West')
plt.plot(months, north_sales, label='North')

plt.title('Regional Sales Comparison - 2024')
plt.xlabel('Month')
plt.ylabel('Sales ($)')
plt.legend()  # Display legend using labels

plt.show()
```

The legend automatically uses the `label` parameter from each plot call. Now you can compare regions: East leads throughout, West is growing faster than North, and all three show upward trends.

3.3 Customizing Line Appearance

Control line color, style, width, and markers to make plots clearer or follow style guidelines.

```python
plt.figure(figsize=(10, 6))

# Customize each line
plt.plot(months, east_sales,
         color='blue',          # Line color
         linestyle='-',         # Solid line
         linewidth=2,           # Line thickness
         marker='o',            # Circle markers at points
         markersize=8,          # Marker size
         label='East')

```

```python
plt.plot(months, west_sales,
         color='red',
         linestyle='--',      # Dashed line
         linewidth=2,
         marker='s',          # Square markers
         markersize=8,
         label='West')

plt.plot(months, north_sales,
         color='green',
         linestyle=':',       # Dotted line
         linewidth=2,
         marker='^',          # Triangle markers
         markersize=8,
         label='North')

plt.title('Regional Sales Comparison - 2024')
plt.xlabel('Month')
plt.ylabel('Sales ($)')
plt.legend()
plt.grid(True, alpha=0.3)  # Light grid

plt.show()
```

Common line styles: '-' (solid), '–' (dashed), ':' (dotted), '-.' (dash-dot). Common markers: 'o' (circle), 's' (square), '^' (triangle), 'D' (diamond). The `alpha` parameter for grid makes it semi-transparent.

3.4 Part 3 Exercise

Exercise: Create quarterly revenue data for two products: Product A = [45000, 52000, 48000, 61000] and Product B = [38000, 41000, 55000, 58000] for Q1-Q4. Plot both on the same figure with different colors, line styles, and markers. Add a legend, title, and axis labels. Which product shows stronger growth?

4 Part 4: Bar Charts for Comparisons

4.1 Basic Bar Charts

Bar charts display categorical data with rectangular bars whose lengths represent values. They're perfect for comparing quantities across categories - you can immediately see which category is largest.

```python
# Average grades by department
departments = ['CS', 'Math', 'Engineering', 'Physics']
avg_grades = [85.5, 82.3, 88.1, 79.8]

plt.figure(figsize=(10, 6))
plt.bar(departments, avg_grades)

plt.title('Average Grade by Department')
plt.xlabel('Department')
plt.ylabel('Average Grade')

# Add value labels on top of each bar
for i, v in enumerate(avg_grades):
    plt.text(i, v + 0.5, f'{v:.1f}', ha='center')

plt.show()
```

The bar chart immediately shows Engineering has the highest average grade and Physics the lowest. Adding value labels on bars makes exact comparisons easy without consulting the y-axis.

4.2 Horizontal Bar Charts

When category names are long, horizontal bars are more readable. Use `barh()` instead of `bar()`.

```python
# Sales by product (long names)
products = ['Premium Widget Pro', 'Basic Gadget Standard',
            'Enterprise Tool Suite', 'Starter Kit Basic']
sales = [45000, 32000, 58000, 28000]

plt.figure(figsize=(10, 6))
plt.barh(products, sales)

plt.title('Sales by Product')
plt.xlabel('Sales ($)')
plt.ylabel('Product')

# Add value labels at end of each bar
for i, v in enumerate(sales):
    plt.text(v + 500, i, f'${v:,}', va='center')

plt.show()
```

Horizontal orientation lets you read long category names easily. It's also conventional for certain chart types like ranking charts where the top position matters.

4.3 Grouped Bar Charts

To compare multiple values per category, use grouped (clustered) bars. This requires calculating bar positions manually.

```python
# Compare Q1 and Q2 sales by region
regions = ['East', 'West', 'North', 'South']
q1_sales = [45000, 38000, 32000, 41000]
q2_sales = [52000, 45000, 38000, 48000]

x = np.arange(len(regions))  # Label positions
width = 0.35  # Bar width

plt.figure(figsize=(10, 6))

# Position Q1 bars slightly left, Q2 slightly right
plt.bar(x - width/2, q1_sales, width, label='Q1')
plt.bar(x + width/2, q2_sales, width, label='Q2')

plt.title('Quarterly Sales Comparison by Region')
plt.xlabel('Region')
plt.ylabel('Sales ($)')
plt.xticks(x, regions)  # Put region names at center
plt.legend()

plt.show()
```

Grouped bars let you compare both across regions (which region sold most?) and across quarters (did sales grow?). You can see all regions improved from Q1 to Q2.

4.4 Stacked Bar Charts

Stacked bars show how parts contribute to totals. Each segment represents a component, and the total bar height shows the sum.

```
# Revenue breakdown by product type per region
regions = ['East', 'West', 'North', 'South']
widgets = [20000, 18000, 15000, 22000]
gadgets = [15000, 12000, 10000, 14000]
tools = [10000, 8000, 7000, 12000]

plt.figure(figsize=(10, 6))

# Stack bars on top of each other
plt.bar(regions, widgets, label='Widgets')
plt.bar(regions, gadgets, bottom=widgets, label='Gadgets')

# Tools go on top of widgets+gadgets
bottom_tools = [w + g for w, g in zip(widgets, gadgets)]
plt.bar(regions, tools, bottom=bottom_tools, label='Tools')

plt.title('Revenue Composition by Region')
plt.xlabel('Region')
plt.ylabel('Revenue ($)')
plt.legend()

plt.show()
```

Stacked bars show both total revenue (bar height) and composition (segments). You can see South has highest total revenue, and Widgets are the largest component in all regions.

4.5 Part 4 Exercise

Exercise: Create a grouped bar chart comparing Morning, Afternoon, and Evening sales for Monday through Friday. Use sample data: Morning = [120, 135, 128, 142, 118], Afternoon = [95, 102, 98, 110, 105], Evening = [80, 88, 92, 85, 95]. Add value labels on top of each bar and a legend. Which time period has the most consistent sales?

5 Part 5: Scatter Plots for Relationships

5.1 Basic Scatter Plots

Scatter plots show each data point as a dot, revealing relationships between two numeric variables. They're essential for spotting correlations, clusters, and outliers.

```
# Study hours vs test scores
np.random.seed(42)
study_hours = np.random.uniform(1, 10, 50)
test_scores = 50 + 4 * study_hours + np.random.normal(0, 5, 50)

plt.figure(figsize=(10, 6))
plt.scatter(study_hours, test_scores)

plt.title('Study Hours vs Test Scores')
plt.xlabel('Study Hours')
plt.ylabel('Test Score')

plt.show()
```

The scatter plot reveals a positive relationship: more study hours correlate with higher test scores. The spread around the trend shows variability - study hours explain much of the score variation, but not all.

5.2 Customizing Scatter Plots

Control marker size, color, and transparency to show additional dimensions or handle overplotting (when points overlap).

```python
# Add a third dimension using color
# Students with different grades

# Generate sample data
n = 100
study_hours = np.random.uniform(1, 10, n)
test_scores = 50 + 4 * study_hours + np.random.normal(0, 8, n)

plt.figure(figsize=(10, 6))

# Color by score (creates a gradient)
scatter = plt.scatter(study_hours, test_scores,
                      c=test_scores,  # Color by score
                      cmap='viridis',  # Color map
                      alpha=0.6,       # Transparency
                      s=100)           # Marker size

plt.colorbar(scatter, label='Test Score')  # Add color legend

plt.title('Study Hours vs Test Scores')
plt.xlabel('Study Hours')
plt.ylabel('Test Score')

plt.show()
```

The color gradient adds a third dimension - you can see that high-scoring students (yellow) tend to study more hours, while low-scoring students (purple) study fewer hours. The alpha transparency helps when points overlap.

5.3 Scatter Plot with Regression Line

Adding a trend line helps quantify the relationship you see in the scatter.

```python
# Fit and plot a regression line
from numpy.polynomial import polynomial as P

# Fit linear regression (degree 1 polynomial)
coefficients = np.polyfit(study_hours, test_scores, 1)
poly = np.poly1d(coefficients)

# Create line points
x_line = np.linspace(study_hours.min(), study_hours.max(), 100)
y_line = poly(x_line)

plt.figure(figsize=(10, 6))

plt.scatter(study_hours, test_scores, alpha=0.6, label='Students')
plt.plot(x_line, y_line, color='red', linewidth=2,
         label=f'Trend: y = {coefficients[0]:.1f}x + {coefficients[1]:.1f}')

plt.title('Study Hours vs Test Scores with Trend Line')
plt.xlabel('Study Hours')
plt.ylabel('Test Score')
plt.legend()

plt.show()
```

The regression line shows the average relationship: for each additional study hour, test scores increase by about 4 points (the slope). The intercept (about 50) is the predicted score with zero study hours.

5.4 Part 5 Exercise

Exercise: Generate data for 75 houses: square footage (np.random.uniform(1000, 4000, 75)) and price (square_footage * 150 + np.random.normal(0, 30000, 75)). Create a scatter plot with alpha=0.6, color points by price using a colormap, and add a colorbar. Fit and plot a regression line. What does the slope tell you about price per square foot?

6 Part 6: Histograms for Distributions

6.1 Basic Histograms

Histograms show how data is distributed across a range of values. They divide data into bins (intervals) and count how many values fall into each bin. This reveals the shape of your data: is it symmetric? skewed? bimodal?

```python
# Student test scores
np.random.seed(42)
scores = np.random.normal(75, 10, 200)  # Mean 75, std 10

plt.figure(figsize=(10, 6))
plt.hist(scores, bins=20)  # 20 bins

plt.title('Distribution of Test Scores')
plt.xlabel('Score')
plt.ylabel('Number of Students')

plt.show()
```

The histogram shows scores are roughly normally distributed around 75, with most students scoring between 60 and 90, and fewer at the extremes. This is more informative than just knowing the mean and standard deviation.

6.2 Customizing Histograms

Control bin count, colors, and add statistical overlays to enhance understanding.

```python
plt.figure(figsize=(10, 6))

# Customize histogram appearance
n, bins, patches = plt.hist(scores,
                            bins=25,
                            edgecolor='black',
                            color='steelblue',
                            alpha=0.7)

# Add vertical lines for mean and median
mean_score = np.mean(scores)
median_score = np.median(scores)

plt.axvline(mean_score, color='red', linestyle='--',
            linewidth=2, label=f'Mean: {mean_score:.1f}')
```

```python
plt.axvline(median_score, color='green', linestyle=':',
            linewidth=2, label=f'Median: {median_score:.1f}')

plt.title('Distribution of Test Scores')
plt.xlabel('Score')
plt.ylabel('Number of Students')
plt.legend()

plt.show()
```

The vertical lines show the mean (average) and median (middle value). When mean and median are close, the distribution is symmetric. When they differ, it's skewed.

6.3 Comparing Distributions

Use multiple histograms to compare distributions across groups. Set alpha for transparency so overlapping areas are visible.

```python
# Compare score distributions across majors
np.random.seed(42)
cs_scores = np.random.normal(78, 8, 100)
math_scores = np.random.normal(75, 12, 100)
eng_scores = np.random.normal(82, 6, 100)

plt.figure(figsize=(10, 6))

plt.hist(cs_scores, bins=20, alpha=0.5, label='CS')
plt.hist(math_scores, bins=20, alpha=0.5, label='Math')
plt.hist(eng_scores, bins=20, alpha=0.5, label='Engineering')

plt.title('Score Distributions by Major')
plt.xlabel('Score')
plt.ylabel('Number of Students')
plt.legend()

plt.show()
```

You can compare distributions: Engineering has higher scores on average and less variability (narrow distribution). Math has lower average and more spread (wider distribution). CS is in between.

6.4 Part 6 Exercise

Exercise: Generate wait times for two restaurants: Restaurant A (np.random.exponential(15, 150) - right-skewed) and Restaurant B (np.random.normal(20, 3, 150) - symmetric). Create overlapping histograms with 25 bins and alpha=0.5. Add vertical lines for the mean and median of each. Which restaurant has more predictable wait times? How do mean and median compare for skewed vs symmetric distributions?

7 Part 7: Plot Customization

7.1 Colors, Styles, and Themes

Matplotlib offers many ways to customize appearance. You can use named colors, hex codes, or RGB tuples. Style sheets change the overall look.

```python
# Available style sheets
print("Available styles:", plt.style.available)

```

```python
# Use a style sheet
plt.style.use('seaborn-v0_8-whitegrid')

# Create a styled plot
x = np.linspace(0, 10, 100)

plt.figure(figsize=(10, 6))
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.title('Trigonometric Functions')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()

# Reset to default
plt.style.use('default')
```

Style sheets provide consistent theming for all your plots. Popular ones include 'seaborn', 'ggplot', 'bmh', and 'fivethirtyeight'. Try several to find one that suits your needs.

7.2 Annotations and Text

Add text annotations to highlight specific data points or provide context.

```python
# Annotate important points
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [12000, 15000, 18000, 22000, 19000, 28000]

plt.figure(figsize=(10, 6))
plt.plot(months, sales, marker='o', markersize=10)

# Annotate the peak
max_idx = sales.index(max(sales))
plt.annotate('Peak Sales!',
             xy=(max_idx, sales[max_idx]),  # Point to annotate
             xytext=(max_idx - 1, sales[max_idx] + 2000),  # Text position
             arrowprops=dict(arrowstyle='->', color='red'),
             fontsize=12,
             color='red')

# Annotate the dip
plt.annotate('Sales dip',
             xy=(4, 19000),
             xytext=(3, 16000),
             arrowprops=dict(arrowstyle='->', color='orange'),
             fontsize=10,
             color='orange')

plt.title('Monthly Sales with Annotations')
plt.xlabel('Month')
plt.ylabel('Sales ($)')

plt.show()
```

Annotations draw attention to important features and provide narrative context. Use them sparingly - too many annotations create clutter.

7.3 Subplots for Multiple Views

Sometimes you need multiple plots together to tell a complete story. Use `subplots()` to create grids of plots.

```python
# Create 2x2 grid of plots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Generate sample data
np.random.seed(42)
data = np.random.normal(50, 15, 200)
x = np.linspace(0, 10, 50)

# Top-left: Line plot
axes[0, 0].plot(x, np.sin(x) * 20 + 50)
axes[0, 0].set_title('Trend Over Time')
axes[0, 0].set_xlabel('Time')
axes[0, 0].set_ylabel('Value')

# Top-right: Histogram
axes[0, 1].hist(data, bins=20, edgecolor='black')
axes[0, 1].set_title('Distribution')
axes[0, 1].set_xlabel('Value')
axes[0, 1].set_ylabel('Count')

# Bottom-left: Bar chart
categories = ['A', 'B', 'C', 'D']
values = [23, 45, 56, 34]
axes[1, 0].bar(categories, values)
axes[1, 0].set_title('Comparison')
axes[1, 0].set_xlabel('Category')
axes[1, 0].set_ylabel('Value')

# Bottom-right: Scatter
axes[1, 1].scatter(x, x*2 + np.random.normal(0, 3, 50))
axes[1, 1].set_title('Relationship')
axes[1, 1].set_xlabel('X')
axes[1, 1].set_ylabel('Y')

plt.tight_layout()  # Adjust spacing to prevent overlap
plt.show()
```

Each subplot is accessed by its position in the grid (row, column). The `tight_layout()` function automatically adjusts spacing. This is useful for dashboards showing multiple perspectives on the same data.

7.4 Part 7 Exercise

Exercise: Create a 1x3 subplot figure showing the same monthly sales data three ways: (1) line plot with markers showing trend, (2) bar chart showing magnitude comparison, (3) scatter plot of month number vs sales. Use a consistent color scheme across all three. Add annotations to highlight the peak month in the line plot. Apply a style sheet of your choice.

8 Part 8: Introduction to Seaborn

8.1 Why Seaborn?

Seaborn is built on matplotlib and provides a high-level interface for statistical graphics. It has attractive default styles, works seamlessly with pandas DataFrames, and creates complex statistical visualizations with simple function calls. Use matplotlib for control and customization; use seaborn for quick, beautiful statistical plots.

```python
import seaborn as sns

```

```python
# Set seaborn style
sns.set_style('whitegrid')

# Seaborn automatically uses attractive colors and styling
# Let's create a simple plot to see the difference

plt.figure(figsize=(10, 6))
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.title('Plot with Seaborn Styling')
plt.show()
```

Just importing seaborn and setting its style improves matplotlib plots. But seaborn's real power is its specialized statistical plotting functions.

8.2 Seaborn with DataFrames

Seaborn integrates beautifully with pandas. You pass the DataFrame and column names, and seaborn handles the rest.

```python
# Create a sample DataFrame
np.random.seed(42)
df = pd.DataFrame({
    'Department': np.random.choice(['CS', 'Math', 'Eng'], 100),
    'Experience': np.random.randint(1, 20, 100),
    'Salary': np.random.normal(70000, 15000, 100)
})

# Scatter plot colored by department
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Experience', y='Salary', hue='Department')
plt.title('Salary vs Experience by Department')
plt.show()
```

The `hue` parameter automatically colors points by category and adds a legend. This would take multiple lines in pure matplotlib.

8.3 Statistical Plot Types

Seaborn excels at statistical visualizations that would be complex in matplotlib.

```python
# Box plot: shows distribution by category
plt.figure(figsize=(10, 6))
sns.boxplot(data=df, x='Department', y='Salary')
plt.title('Salary Distribution by Department')
plt.show()

# Violin plot: like box plot but shows full distribution
plt.figure(figsize=(10, 6))
sns.violinplot(data=df, x='Department', y='Salary')
plt.title('Salary Distribution by Department (Violin)')
plt.show()
```

Box plots show median, quartiles, and outliers. Violin plots add the full distribution shape. These statistical summaries help you compare groups beyond just means.

8.4 Pair Plots and Heatmaps

Seaborn can create complex multi-variable visualizations with single function calls.

```python
# Create a DataFrame with multiple numeric columns
df_numeric = pd.DataFrame({
    'Var1': np.random.normal(0, 1, 100),
    'Var2': np.random.normal(0, 1, 100),
    'Var3': np.random.normal(0, 1, 100)
})
df_numeric['Var2'] = df_numeric['Var1'] * 0.8 + np.random.normal(0, 0.5, 100)

# Pair plot: scatter plots for all pairs of variables
sns.pairplot(df_numeric)
plt.suptitle('Pair Plot of All Variables', y=1.02)
plt.show()

# Correlation heatmap
plt.figure(figsize=(8, 6))
correlation = df_numeric.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Heatmap')
plt.show()
```

Pair plots show relationships between all pairs of variables - perfect for exploring datasets with multiple numeric columns. Heatmaps visualize matrices of values, commonly used for correlation matrices where you want to spot which variables are related.

8.5 Part 8 Exercise

Exercise: Create a DataFrame with 80 employees: Department (random choice of 'Sales', 'IT', 'HR'), Years_Experience (randint 1-15), and Salary (50000 + Years * 3000 + random noise). Using seaborn: (1) Create a scatter plot colored by Department, (2) Create a box plot of Salary by Department, (3) Create a violin plot of Years_Experience by Department. Which department has the highest salary variability?

9 Part 9: Complete Visualization Workflow

9.1 From Data to Insight: A Complete Example

Let's create a complete visualization workflow that demonstrates how to combine the techniques you've learned. We'll analyze sales data and create a multi-plot report.

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

# Generate comprehensive sales data
np.random.seed(42)
n = 500

sales_data = pd.DataFrame({
    'Date': pd.date_range('2024-01-01', periods=n, freq='D'),
    'Region': np.random.choice(['East', 'West', 'North', 'South'], n),
    'Product': np.random.choice(['Widget', 'Gadget', 'Tool'], n),
    'Units': np.random.randint(10, 100, n),
    'Revenue': np.random.uniform(500, 5000, n)
})

# Add month for aggregation
sales_data['Month'] = sales_data['Date'].dt.month

print("Sales data sample:")
print(sales_data.head())
print(f"\nTotal records: {len(sales_data)}")
```

9.2 Creating a Dashboard

```python
# Create a 2x2 dashboard
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle('Sales Performance Dashboard - 2024', fontsize=16)

# Plot 1: Monthly Revenue Trend (Line)
monthly_revenue = sales_data.groupby('Month')['Revenue'].sum()
axes[0, 0].plot(monthly_revenue.index, monthly_revenue.values,
                marker='o', linewidth=2)
axes[0, 0].set_title('Monthly Revenue Trend')
axes[0, 0].set_xlabel('Month')
axes[0, 0].set_ylabel('Total Revenue ($)')
axes[0, 0].grid(True, alpha=0.3)

# Plot 2: Revenue by Region (Bar)
regional_revenue = sales_data.groupby('Region')['Revenue'].sum().sort_values()
axes[0, 1].barh(regional_revenue.index, regional_revenue.values)
axes[0, 1].set_title('Total Revenue by Region')
axes[0, 1].set_xlabel('Revenue ($)')

# Plot 3: Revenue Distribution (Histogram)
axes[1, 0].hist(sales_data['Revenue'], bins=30, edgecolor='black', alpha=0.7)
axes[1, 0].axvline(sales_data['Revenue'].mean(), color='red',
                   linestyle='--', label=f"Mean: ${sales_data['Revenue'].mean()
                       :.0f}")
axes[1, 0].set_title('Revenue Distribution')
axes[1, 0].set_xlabel('Revenue ($)')
axes[1, 0].set_ylabel('Frequency')
axes[1, 0].legend()

# Plot 4: Units vs Revenue (Scatter)
scatter = axes[1, 1].scatter(sales_data['Units'], sales_data['Revenue'],
                              alpha=0.5, c=sales_data['Month'], cmap='viridis')
axes[1, 1].set_title('Units Sold vs Revenue')
axes[1, 1].set_xlabel('Units')
axes[1, 1].set_ylabel('Revenue ($)')
plt.colorbar(scatter, ax=axes[1, 1], label='Month')

plt.tight_layout()
plt.savefig('sales_dashboard.png', dpi=150, bbox_inches='tight')
plt.show()

print("Dashboard saved to sales_dashboard.png")
```

This dashboard provides four complementary views: trend over time (line), comparison across categories (bar), distribution shape (histogram), and variable relationship (scatter). Together they tell a complete story about sales performance.

9.3 Creating a Seaborn Report

```python
# Use seaborn for statistical visualizations
sns.set_style('whitegrid')

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle('Statistical Analysis Report', fontsize=16)

# Plot 1: Box plot of Revenue by Region
sns.boxplot(data=sales_data, x='Region', y='Revenue', ax=axes[0, 0])
axes[0, 0].set_title('Revenue Distribution by Region')

# Plot 2: Bar plot of mean Revenue by Product
```

```
product_revenue = sales_data.groupby('Product')['Revenue'].mean().reset_index()
sns.barplot(data=product_revenue, x='Product', y='Revenue', ax=axes[0, 1])
axes[0, 1].set_title('Average Revenue by Product')

# Plot 3: Violin plot of Units by Region
sns.violinplot(data=sales_data, x='Region', y='Units', ax=axes[1, 0])
axes[1, 0].set_title('Units Distribution by Region')

# Plot 4: Scatter with regression line
sns.regplot(data=sales_data, x='Units', y='Revenue',
            scatter_kws={'alpha': 0.3}, ax=axes[1, 1])
axes[1, 1].set_title('Units vs Revenue with Trend')

plt.tight_layout()
plt.savefig('statistical_report.png', dpi=150, bbox_inches='tight')
plt.show()

print("Statistical report saved to statistical_report.png")
```

Seaborn's box plots, violin plots, and regression plots add statistical depth to your analysis. The regression line in the scatter plot shows the relationship trend and its confidence interval.

9.4 Part 9 Exercise

Exercise: Using the complete sales_data from this section, create a comprehensive 2x3 dashboard that includes: (1) Line plot of daily revenue (aggregate by day), (2) Horizontal bar chart of total revenue by Region sorted by value, (3) Stacked bar chart showing revenue by Product for each Region, (4) Histogram of Revenue with mean line, (5) Scatter plot of Units vs Revenue with regression line, (6) Seaborn heatmap of correlations between numeric columns. Save the complete dashboard as 'comprehensive_dashboard.png' with dpi=200.

Summary and Best Practices

Key Takeaways:

1. Choose chart types based on what you want to show: trends (line), comparisons (bar), relationships (scatter), distributions (histogram)
2. Always include titles and axis labels - unlabeled plots are meaningless
3. Use color purposefully: to distinguish categories, show values, or highlight points
4. Consider your audience: exploratory plots can be rough; presentation plots need polish
5. Seaborn provides beautiful statistical plots with minimal code
6. Use subplots to show multiple related views in one figure
7. Save plots before showing them; use appropriate resolution for intended use
8. Start simple and add complexity only when it improves understanding

Common Pitfalls to Avoid

- Missing axis labels and titles - the most common visualization mistake
- Using pie charts (hard to compare) when bar charts would be clearer

- Too many colors, annotations, or chart elements creating visual clutter
- Misleading scales (truncated y-axes, non-zero baselines for bars)
- Using 3D effects that distort data perception
- Plotting too many lines or categories on one chart
- Forgetting plt.tight_layout() causing overlapping labels in subplots
- Not saving the figure before plt.show() (which clears it)
- Using seaborn functions without importing seaborn first

Next Steps

You now have foundational visualization skills with matplotlib and seaborn. Advanced topics to explore include:

- Interactive visualizations with Plotly or Bokeh
- Geographic visualizations with mapping libraries
- Animation and dynamic plots
- Custom styling and corporate themes
- Dashboard creation with Streamlit or Dash
- Publication-quality figures for academic papers

Visualization is both science and art. The science is choosing appropriate charts and accurate representations. The art is making them clear, beautiful, and compelling. Practice by visualizing datasets you care about, and study exemplary visualizations to develop your aesthetic sense. Remember: the best visualization is one that makes your insight obvious to your audience.