# Matplotlib Cheatsheet

**Essential operations for data visualization and plotting**

This cheatsheet provides a quick reference to fundamental Matplotlib operations, syntax, and advanced features, ideal for both beginners and experienced data scientists for creating effective data visualizations.

| Basic Plotting | Plot Customization | Multiple Plots |
|---|---|---|
| Create fundamental plots and charts | Style and format your visualizations | Create subplots and complex layouts |

| Advanced Charts | Export & Save |
|---|---|
| Specialized visualization types | Output plots in various formats |

## Basic Plotting & Chart Types

### Line Plot: `plt.plot()`

Create line charts for continuous data visualization.

```
import matplotlib.pyplot as plt
import numpy as np
# Basic line plot
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
plt.plot(x, y)
plt.show()
# Multiple lines
plt.plot(x, y, label='Line 1')
plt.plot(x, [1, 3, 5, 7, 9], label='Line 2')
plt.legend()
# Line styles and colors
plt.plot(x, y, 'r--', linewidth=2, marker='o')
```

### Scatter Plot: `plt.scatter()`

Display relationships between two variables.

```
# Basic scatter plot
plt.scatter(x, y)
# With different colors and sizes
colors = [1, 2, 3, 4, 5]
sizes = [20, 50, 100, 200, 500]
plt.scatter(x, y, c=colors, s=sizes, alpha=0.6)
plt.colorbar()  # Add color bar
```

### Bar Chart: `plt.bar()` / `plt.barh()`

Create vertical or horizontal bar charts.

```
# Vertical bars
categories = ['A', 'B', 'C', 'D']
values = [25, 35, 30, 27]
plt.bar(categories, values)
# Horizontal bars
plt.barh(categories, values)
# Grouped bars
x = np.arange(len(categories))
plt.bar(x - 0.2, values, 0.4, label='Group 1')
plt.bar(x + 0.2, [15, 25, 20, 35], 0.4, label='Group 2')
```

### Histogram: `plt.hist()`

Show distribution of continuous data.

```
# Basic histogram
data = np.random.randn(1000)
plt.hist(data, bins=30)
# Customized histogram
plt.hist(data, bins=30, alpha=0.7, color='skyblue',
edgecolor='black')
# Multiple histograms
plt.hist([data1, data2, data3], bins=30, alpha=0.7, label=['Data 1', 'Data 2'])
```

### Pie Chart: `plt.pie()`

Display proportional data as a circular chart.

```
# Basic pie chart
sizes = [25, 35, 20, 20]
labels = ['A', 'B', 'C', 'D']
plt.pie(sizes, labels=labels)
# Exploded pie with percentages
explode = (0.1, 0, 0, 0)  # explode 1st slice
plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        explode=explode, shadow=True, startangle=90)
```

### Box Plot: `plt.boxplot()`

Visualize data distribution and outliers.

```
# Single box plot
data = [np.random.randn(100) for _ in range(4)]
plt.boxplot(data)
# Customized box plot
plt.boxplot(data, labels=['Group 1', 'Group 2', 'Group 3', 'Group 4'],
            patch_artist=True, notch=True)
```

## Plot Customization & Styling

Enhance your plots with labels, titles, colors, and formatting.

### Labels & Titles: `plt.xlabel()` / `plt.title()`

Add descriptive text to your plots for clarity and context.

```
# Basic labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Chart Title')
# Formatted titles with font properties
plt.title('My Plot', fontsize=16, fontweight='bold')
plt.xlabel('Time (seconds)', fontsize=12)
# Grid for better readability
plt.grid(True, alpha=0.3)
```

### Colors & Styles: `color` / `linestyle` / `marker`

Customize the visual appearance of plot elements.

```
# Color options
plt.plot(x, y, color='red')  # Named colors
plt.plot(x, y, color='#FF5733')  # Hex colors
plt.plot(x, y, color=(0.1, 0.2, 0.5))  # RGB tuple
# Line styles
plt.plot(x, y, linestyle='-')  # Solid
plt.plot(x, y, linestyle='--')  # Dashed
plt.plot(x, y, linestyle='-.')  # Dash-dot
# Markers
plt.plot(x, y, marker='o', markersize=8, markerfacecolor='red')
```

### Legends & Annotations: `plt.legend()` / `plt.annotate()`

Add legends and annotations to explain plot elements.

```
# Basic legend
plt.plot(x, y1, label='Dataset 1')
plt.plot(x, y2, label='Dataset 2')
plt.legend()
# Customize legend position
plt.legend(loc='upper right')
# Other options: 'upper left', 'lower right', 'best', etc.
# Annotations
plt.annotate('Important Point', xy=(2, 4), xytext=(3, 5),
             arrowprops=dict(arrowstyle='->', color='red'))
```

## Axes & Layout Control

### Axis Limits: `plt.xlim()` / `plt.ylim()`

Control the range of values displayed on each axis.

```
# Set axis limits
plt.xlim(0, 10)
plt.ylim(-5, 15)
# Automatic limits with margin
plt.margins(x=0.1, y=0.1)
# Invert axis
plt.gca().invert_yaxis()  # Invert y-axis
```

### Ticks & Labels: `plt.xticks()` / `plt.yticks()`

Customize axis tick marks and their labels.

```
# Custom tick positions
plt.xticks([0, 2, 4, 6, 8, 10])
plt.yticks(np.arange(0, 101, 10))
# Custom tick labels
plt.xticks([1, 2, 3], ['Jan', 'Feb', 'Mar', 'Apr'])
# Rotate tick labels
plt.xticks(rotation=45)
# Remove ticks
plt.xticks([])
plt.yticks([])
```

### Aspect Ratio: `plt.axis()`

Control the aspect ratio and axis appearance.

```
# Equal aspect ratio
plt.axis('equal')
# Square plot
plt.axis('square')
# Turn off axis
plt.axis('off')
# Custom aspect ratio
plt.gca().set_aspect('equal', adjustable='box')
```

### Figure Size: `plt.figure()`

Control the overall size and resolution of your plots.

```
# Set figure size (width, height in inches)
plt.figure(figsize=(10, 6))
# High-DPI for better quality
plt.figure(figsize=(8, 6), dpi=300)
# Multiple figures
fig1 = plt.figure(1)
plt.plot(x, y1)
fig2 = plt.figure(2)
plt.plot(x, y2)
```

### Tight Layout: `plt.tight_layout()`

Automatically adjust subplot spacing for better appearance.

```
# Prevent overlapping elements
plt.tight_layout()
# Manual spacing adjustment
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)
# Padding around subplots
plt.tight_layout(pad=3.0)
```

### Style Sheets: `plt.style.use()`

Apply predefined styles for consistent plot appearance.

```
# Available styles
print(plt.style.available)
# Use built-in styles
plt.style.use('seaborn-v0_8')
plt.style.use('ggplot')
plt.style.use('dark_background')
# Reset to default
plt.style.use('default')
```

## Subplots & Multiple Plots

### Basic Subplots: `plt.subplot()` / `plt.subplots()`

Create multiple plots in a single figure.

```
# Create 2x2 subplot grid
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
# Plot on specific subplot
axes[0, 0].plot(x, y1)
axes[0, 1].scatter(x, y2)
axes[1, 0].bar(x, y3)
axes[1, 1].hist(data)
# Alternative syntax
plt.subplot(2, 2, 1)  # 2 rows, 2 cols, 1st subplot
plt.plot(x, y)
plt.subplot(2, 2, 2)  # 2nd subplot
plt.scatter(x, y)
```

### GridSpec: Advanced Layouts

Create complex subplot arrangements with varying sizes.

```
import matplotlib.gridspec as gridspec
# Create custom grid
fig = plt.figure(figsize=(12, 8))
gs = gridspec.GridSpec(3, 3)
# Different sized subplots
ax1 = fig.add_subplot(gs[0, :])  # Top row, all columns
ax2 = fig.add_subplot(gs[1:, 0:2])  # Middle row, first 2 columns
ax3 = fig.add_subplot(gs[1:, 2])  # Last column, bottom 2 rows
ax4 = fig.add_subplot(gs[2, 0])  # Bottom left
ax5 = fig.add_subplot(gs[2, 1])  # Bottom middle
```

### Subplot Spacing: `hspace` / `wspace`

Control spacing between subplots.

```
# Adjust spacing when creating subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
plt.subplots_adjust(hspace=0.4, wspace=0.3)
# Or use tight_layout for automatic adjustment
plt.tight_layout()
```

## Advanced Visualization Types

### Heatmaps: `plt.imshow()` / `plt.pcolormesh()`

Visualize 2D data as color-coded matrices.

```
# Basic heatmap
data = np.random.random((10, 10))
plt.imshow(data, cmap='viridis')
plt.colorbar()
# Pcolormesh for irregular grids
x = np.linspace(0, 10, 11)
y = np.linspace(0, 5, 6)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)
plt.pcolormesh(X, Y, Z, shading='auto')
plt.colorbar()
```

### Error Bars: `plt.errorbar()`

Display data with uncertainty measurements.

```
# Basic error bars
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
yerr = [0.5, 0.8, 0.3, 0.7, 0.4]
plt.errorbar(x, y, yerr=yerr, fmt='o-', capsize=5)
# Asymmetric error bars
yerr_lower = [0.4, 0.6, 0.2, 0.5, 0.3]
yerr_upper = [0.6, 0.9, 0.4, 0.9, 0.5]
plt.errorbar(x, y, yerr=[yerr_lower, yerr_upper], fmt='s-')
```

### Contour Plots: `plt.contour()` / `plt.contourf()`

Show level curves and filled contour regions.

```
# Contour lines
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
Z = X**2 + Y**2
plt.contour(X, Y, Z, levels=10)
# Filled contours
plt.contourf(X, Y, Z, levels=20, cmap='RdBu')
# Filled contours
plt.colorbar()
```

### Fill Between: `plt.fill_between()`

Shade areas between curves or around lines.

```
# Fill between two curves
y1 = [1, 4, 6, 8, 10]
y2 = [1, 3, 5, 7, 9]
plt.fill_between(x, y1, y2, alpha=0.3, color='blue')
# Fill between line with error
plt.plot(x, y, 'k-', linewidth=2)
plt.fill_between(x, y-yerr, y+yerr, alpha=0.2, color='gray')
```

### 3D Plots: `mplot3d`

Create three-dimensional visualizations.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# 3D scatter
ax.scatter(x, y, z)
# 3D surface plot
ax.plot_surface(X, Y, Z, cmap='viridis')
# 3D line plot
ax.plot(x, y, z)
```

### Violin Plots: Alternative to Box Plots

Show distribution shape along with quartiles.

```
# Using pyplot
parts = plt.violinplot([data1, data2, data3])
# Customize colors
for pc in parts['bodies']:
    pc.set_facecolor('lightblue')
    pc.set_alpha(0.7)
```

## Interactive & Animation Features

Create dynamic and interactive visualizations.

### Interactive Backend: `%matplotlib widget`

Enable interactive plots in Jupyter notebooks.

```
# In Jupyter notebook
%matplotlib widget
# Or basic interactivity
%matplotlib notebook
# Interactive zoom, pan, and hover
```

### Animations: `matplotlib.animation`

Create animated plots for time-series or changing data.

```
from matplotlib.animation import FuncAnimation
fig, ax = plt.subplots()
line, = ax.plot([], [], 'r-')
ax.set_xlim(0, 10)
ax.set_ylim(-2, 2)
def animate(frame):
    x = np.linspace(0, 10, 100)
    y = np.sin(x + frame * 0.1)
    line.set_data(x, y)
    return line,
anim = FuncAnimation(fig, animate, frames=200, blit=True, interval=50)
plt.show()
# Save animation
# anim.save('animation.gif', writer='pillow')
```

### Event Handling: Mouse & Keyboard

Respond to user interactions with plots.

```
def on(click(event):
    if event.inaxes:
        print(f'Clicked at: x={event.xdata}, y={event.ydata}')
fig, ax = plt.subplots()
ax.plot(x, y)
fig.canvas.mpl_connect('button_press_event', on(click)
plt.show()
```

## Saving & Exporting Plots

Save your visualizations in various formats for different purposes.

### Save Figure: `plt.savefig()`

Export plots in high-quality files with various options.

```
# Basic save
plt.savefig('my_plot.png')
# High-quality save
plt.savefig('plot.png', dpi=300, bbox_inches='tight')
# Different formats
plt.savefig('plot.pdf')  # PDF
plt.savefig('plot.svg')  # SVG (vector)
plt.savefig('plot.eps')  # EPS
# Transparent background
plt.savefig('plot.png', transparent=True)
```

### Figure Quality: DPI & Size

Control resolution and dimensions of saved plots.

```
# High DPI for publications
plt.savefig('plot.png', dpi=600)
# Custom size (width, height in inches)
plt.figure(figsize=(12, 8))
plt.savefig('plot.png', figsize=(12, 8))
# Crop whitespace
plt.savefig('plot.png', bbox_inches='tight', pad_inches=0.1)
```

### Batch Export & Memory Management

Handle multiple plots and memory efficiently.

```
# Close figures to free memory
plt.close()  # Close current figure
plt.close('all')  # Close all figures
# Context manager for automatic cleanup
with plt.figure() as fig:
    plt.plot(x, y)
    plt.savefig('plot.png')
# Batch save multiple plots
for i, data in enumerate(datasets):
    plt.figure()
    plt.plot(data)
    plt.savefig(f'plot_{i}.png')
    plt.close()
```

## Configuration & Best Practices

Optimize Matplotlib settings and follow visualization best practices.

### RC Parameters: `plt.rcParams`

Set default styling and behavior for all plots.

```
# Common rc parameters
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 12
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['axes.grid'] = True
# Save and restore settings
original_params = plt.rcParams.copy()
# ... make changes ...
plt.rcParams.update(original_params)  # Restore
```

### Performance Optimization

Improve plotting performance for large datasets.

```
# Use blitting for animations
anim = FuncAnimation(fig, animate, blit=True)
# Rasterize complex plots
plt.plot(x, y, rasterized=True)
# Reduce data points for display
# Downsample data before plotting
indices = np.arange(0, len(large_data), step=10)
plt.plot(large_data[indices])
```

### Color Management: Colormaps & Palettes

Work effectively with colors and colormaps.

```
# List available colormaps
print(plt.colormaps())
# Use colormap for multiple lines
colors = plt.cm.viridis(np.linspace(0, 1, len(datasets)))
for i, (data, color) in enumerate(zip(datasets, colors)):
    plt.plot(data, color=color, label=f'Dataset {i+1}')
# Create custom colormap
from matplotlib.colors import LinearSegmentedColormap
custom_cmap = LinearSegmentedColormap.from_list('custom', ['red', 'yellow', 'blue'])
```

### Memory Usage: Efficient Plotting

Manage memory when creating many plots or large visualizations.

```
# Clear axes instead of creating new figures
fig, ax = plt.subplots()
for data in datasets:
    ax.clear()  # Clear previous plot
    ax.plot(data)
    plt.savefig(f'plot_{i}.png')
# Use generators for large datasets
def data_generator():
    for i in range(1000):
        yield np.random.randn(100)
for i, data in enumerate(data_generator()):
    if i > 10:  # Limit number of plots
        break
```

## Integration with Data Libraries

### Pandas Integration: Direct Plotting

Use Matplotlib through Pandas DataFrame methods.

```
import pandas as pd
# DataFrame plotting (uses matplotlib backend)
df = pd.read_csv('data.csv')
df.plot.scatter(x='x_col', y='y_col')
df.plot.hist(bins=30)
df.plot.box()
# Access underlying matplotlib objects
ax = df.plot(kind='line')
ax.set_title('Custom Title')
plt.show()
# Statistical distributions
data = np.random.normal(0, 1, 1000)
plt.hist(data, bins=50, density=True, alpha=0.7)
```

### Seaborn Integration: Enhanced Styling

Combine Matplotlib with Seaborn for better default aesthetics.

```
import seaborn as sns
# Use seaborn styling with matplotlib
sns.set_style('whitegrid')
plt.plot(x, y)
plt.show()
# Mix Seaborn and matplotlib
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
sns.scatterplot(data=df, x='x', y='y', ax=axes[0,0])
plt.plot(x, y, ax=axes[1,1])  # Pure matplotlib
```

### Jupyter Integration: Inline Plotting

Optimize Matplotlib for Jupyter notebook environments.

```
# Magic commands for Jupyter
%matplotlib inline  # Static plots
%matplotlib widget  # Interactive plots
# High-DPI displays
%config InlineBackend.figure_format = 'retina'
# Automatic figure sizing
%matplotlib inline
plt.rcParams['figure.dpi'] = 100
```

## Installation & Environment Setup

Install and configure Matplotlib for various platforms.

### Pip: `pip install matplotlib`

Standard Python package installer for Matplotlib.

```
# Install Matplotlib
pip install matplotlib
# Upgrade to latest version
pip install matplotlib --upgrade
# Install with additional backends
pip install matplotlib[all]
# Show package information
pip show matplotlib
```

### Conda: `conda install matplotlib`

Package manager for Anaconda/Miniconda environments.

```
# Install in current environment
conda install matplotlib
# Update matplotlib
conda update matplotlib
# Create environment with matplotlib
conda create -n dataviz matplotlib numpy pandas
# List matplotlib info
conda list matplotlib
```

### Backend Configuration

Set up display backends for different environments.

```
# Check available backends
import matplotlib
print(matplotlib.get_backend())
# Set backend programmatically
matplotlib.use('TkAgg')  # For Tkinter
matplotlib.use('Qt5Agg')  # For PyQt5
# For headless servers
matplotlib.use('Agg')
# Import after setting backend
import matplotlib.pyplot as plt
```

**References:** This cheatsheet covers essential Matplotlib commands and modern practices for creating effective data visualizations in data science workflows.