

## What is Data Preprocessing?

**Data preprocessing** is the process of cleaning, transforming, and organizing raw data into a format that can be easily understood and analyzed by machine learning algorithms and visualization tools.

## Example

# This is what raw data often looks like:

```
raw_data = {
    'age': [25, 35, 'unknown', 42, -5],      # Missing values, invalid numbers
    'salary': [50000, '60,000', 75000, 'N/A', 80000], # Different formats, missing
    'name': ['John', 'Mary', 'Bob', 'Alice', 'John'], # Duplicates
    'date': ['2023-01-15', '15/02/2023', 'March 5', '2023-04-20', '2023-01-15'] # Inconsistent formats
}
```

If you feed messy data to your visualization or machine learning models:

- **Incorrect results**
- **Misleading patterns**
- **Poor decisions**
- **Wasted time and resources**

## Key Steps in Data Preprocessing

### 1. Handling Missing Values

*Common Approaches:*

python

```
import pandas as pd
import numpy as np
```

```
# Sample data with missing values
data = {'age': [25, np.nan, 35, 42, np.nan],
        'salary': [50000, 60000, np.nan, 75000, 80000]}

df = pd.DataFrame(data)

# Method 1: Remove rows with missing values
df_cleaned = df.dropna()

# Method 2: Fill with mean/median/mode
df['age'].fillna(df['age'].mean(), inplace=True)
df['salary'].fillna(df['salary'].median(), inplace=True)

# Method 3: Fill with specific value
df.fillna(0, inplace=True) # or df.fillna('Unknown')
```

## 2. Handling Duplicate Data

python

```
# Remove duplicate rows
df = df.drop_duplicates()

# Remove duplicates based on specific columns
df = df.drop_duplicates(subset=['email', 'phone_number'])
```

## 3. Data Type Conversion

python

```
# Convert data types
df['age'] = df['age'].astype(int)
df['salary'] = df['salary'].astype(float)
df['date'] = pd.to_datetime(df['date'])

# Handle currency strings
df['salary'] = df['salary'].str.replace('$', '').str.replace(',', '').astype(float)
```

## 4. Handling Outliers

python

```
# Identify outliers using IQR method
Q1 = df['salary'].quantile(0.25)
Q3 = df['salary'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Remove outliers
df_clean = df[(df['salary'] >= lower_bound) & (df['salary'] <= upper_bound)]

# Or cap outliers
df['salary'] = np.where(df['salary'] > upper_bound, upper_bound,
                        np.where(df['salary'] < lower_bound, lower_bound, df['salary']))
```

## What is Data Visualization?

### Definition

**Data visualization** is the graphical representation of information and data using visual elements like charts, graphs, maps, and other visual tools.


## Why Do We Need Data Visualization?

### 1. Humans are Visual Creatures

- Our brains process images **60,000 times faster** than text
- We remember **80%** of what we see, but only **20%** of what we read

### 2. Spot Patterns Quickly

```
# Which is easier to understand?
# Raw data:
[125, 130, 118, 145, 160, 155, 172, 168, 190, 205]

# Or a visual?
#  (A line going upward showing growth)
```

### 3. Tell Stories with Data

Visualizations help explain:

- **Trends** (sales increasing/decreasing)
- **Comparisons** (Product A vs Product B)
- **Relationships** (temperature vs ice cream sales)
- **Distributions** (test scores across a class)

## Components of Data Visualization

### 1. Charts & Graphs

- **Line charts** → Trends over time
- **Bar charts** → Comparisons
- **Pie charts** → Parts of a whole
- **Scatter plots** → Relationships

### 2. Visual Elements

- **Colors** → Different categories or values
- **Shapes** → Different data groups
- **Size** → Importance or magnitude
- **Position** → Relationships and comparisons

### 3. Tools We Use

```
# Code: libraries for visualization
import matplotlib.pyplot as plt # Basic plotting
import seaborn as sns          # Statistical visuals
```

```
import plotly.express as px          # Interactive plots
import pandas as pd                 # Data manipulation
```

## Matplotlib: The Foundation

### Definition

**Matplotlib** is a comprehensive **Code: library** for creating static, animated, and interactive visualizations. It's like the "**Photoshop for data**" in Code:!

**Matplotlib** was created by **John D. Hunter** in 2003 because he was frustrated with the plotting options available for medical research data! Now it's used by millions worldwide.

### Key Features

- Creates **2D and 3D** plots
- **Highly customizable** - control every detail
- Works with **multiple data formats** (lists, arrays, DataFrames)
- **Publication-quality** graphics
- **Free and open-source**

## Pyplot: The Easy-to-Use Interface

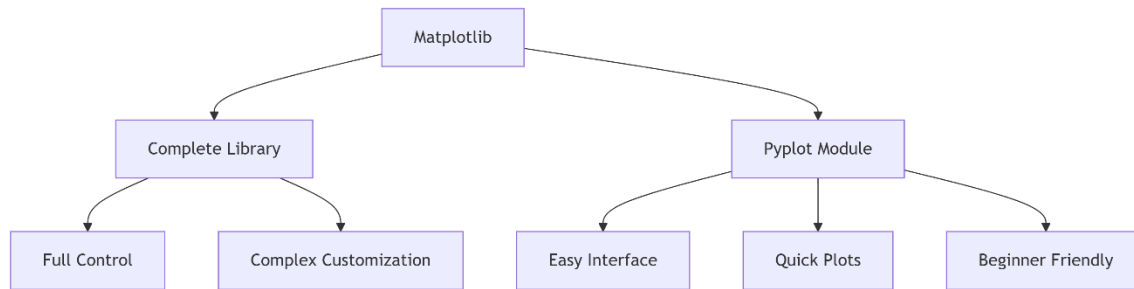
### Definition

**Pyplot** is a **module within Matplotlib** that provides a simple, interface for plotting. It's the "**easy button**" for creating visualizations!

### Key Features

- **Simplified commands** - easier to learn
- **Automatic figure and axis management**
- **Quick plotting** for everyday tasks

## Relationship Between Matplotlib and Pyplot



## Common Types of Visualizations

Visualization	Best For	Example
<b>Line Chart</b>	Trends over time	Stock prices, temperature changes
<b>Bar Chart</b>	Comparing categories	Sales by product, scores by student
<b>Pie Chart</b>	Showing proportions	Budget allocation, market share
<b>Scatter Plot</b>	Relationships between variables	Height vs weight, study time vs grades
<b>Histogram</b>	Distribution of data	Test scores, age distribution

## How They Work Together

```

# Import statement - this is the standard way
import matplotlib.pyplot as plt

# Now we can use:
plt.plot()      # ← This is Pyplot making Matplotlib easier to use!
plt.bar()
plt.scatter()
  
```

## Basic Plotting

```
plt.plot(x, y)           # Line chart
plt.scatter(x, y)        # Scatter plot
plt.bar(x, y)            # Bar chart
plt.hist(data)           # Histogram
plt.pie(sizes)           # Pie chart
```

## Customization

```
plt.title('My Chart')    # Add title
plt.xlabel('X Axis')     # X-axis label
plt.ylabel('Y Axis')     # Y-axis label
plt.legend()             # Show legend
plt.grid(True)           # Add grid lines
```

## Display & Save

```
plt.show()               # Display the plot
plt.savefig('plot.png')  # Save to file
```

# 1. plt.plot() - Line Chart

## What it Does

Creates a **line chart** that shows trends and patterns over time or ordered categories.

## Syntax

Code:

```
plt.plot(x_values, y_values, color='red', linestyle='-', marker='o', linewidth=2, label='Trend')
```

## Parameters

- **x\_values**: Data for x-axis (e.g., time, categories)
- **y\_values**: Data for y-axis (e.g., values, measurements)
- **color**: Line color ('red', 'blue', '#FF5733')
- **linestyle**: '-'(solid), '--'(dashed), ':'(dotted), '-.'(dash-dot)
- **marker**: 'o'(circle), 's'(square), '^'(triangle), 'D'(diamond)

- `linewidth`: Thickness of the line (number)
- `label`: Name for legend

## Example with Sales Data

Code:

```
import matplotlib.pyplot as plt

# Monthly profit trend
months = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
profit = [211000, 183300, 224700, 222700, 209600, 201400,
          295500, 361400, 234000, 266700, 412800, 300200]

plt.plot(months, profit, color='green', marker='o', linestyle='-', linewidth=2, label='Monthly Profit')
plt.title('Profit Trend Over Year')
plt.xlabel('Month')
plt.ylabel('Profit ($)')
plt.legend()
plt.grid(True)
plt.show()
```





## Best For

- Time series data (sales over months)
  - Trends and patterns
  - Continuous data
- 

## 2. plt.scatter() - Scatter Plot

### What it Does

Shows the **relationship between two variables** using dots. Each dot represents one data point.

### Syntax

Code:

```
plt.scatter(x_values, y_values, s=50, c='blue', marker='o', alpha=0.6,
            label='Data Points')
```

### Parameters

- `x_values`: First variable
- `y_values`: Second variable
- `s`: Size of markers (can be single number or array)
- `c`: Color of markers
- `marker`: Shape of markers
- `alpha`: Transparency (0.0 to 1.0)

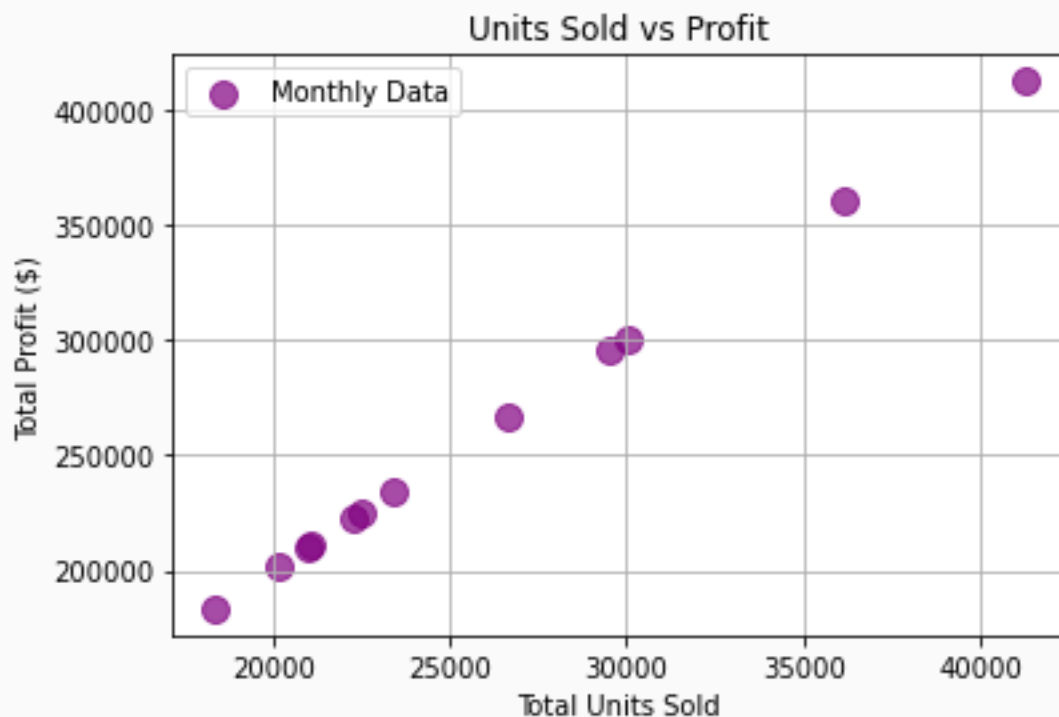
### Example with Sales Data

Code:

```
# Relationship between total units and profit
units = [21100, 18330, 22470, 22270, 20960, 20140, 29550, 36140, 23400,
         26670, 41280, 30020]
profit = [211000, 183300, 224700, 222700, 209600, 201400,
```

```
295500, 361400, 234000, 266700, 412800, 300200]
```

```
plt.scatter(units, profit, s=100, c='purple', alpha=0.7, label='Monthly Data')
plt.title('Units Sold vs Profit')
plt.xlabel('Total Units Sold')
plt.ylabel('Total Profit ($)')
plt.legend()
plt.grid(True)
plt.show()
```



### Best For

- Correlation analysis
- Outlier detection
- Relationship between two variables

## 3. plt.bar() - Bar Chart

## What it Does

Compares **categories** using rectangular bars. Height/length represents the value.

## Syntax

Code:

```
plt.bar(categories, values, color='blue', width=0.8, alpha=0.7, label='Products')
```

## Parameters

- `categories`: Names for each bar
- `values`: Heights of the bars
- `color`: Bar color(s)
- `width`: Width of bars (0-1)
- `alpha`: Transparency

## Example with Sales Data

Code:

```
# Compare average product sales
products = ['facecream', 'facewash', 'toothpaste', 'bathingsoap', 'shampoo', 'moisturizer']
avg_sales = [df[product].mean() for product in products] # Calculate averages

plt.bar(products, avg_sales, color=['lightcoral', 'lightblue', 'lightgreen', 'gold', 'plum', 'orange'])
plt.title('Average Monthly Sales by Product')
plt.xlabel('Products')
plt.ylabel('Average Sales Units')
plt.xticks(rotation=45) # Rotate labels for readability
plt.grid(axis='y') # Only horizontal grid lines
plt.show()
```



### Best For

- Comparing categories
- Discrete data
- Showing rankings

---

## 4. plt.hist() - Histogram

### What it Does

Shows the **distribution** of numerical data. Groups data into "bins" and shows frequency.

### Syntax

Code:

```
plt.hist(data, bins=10, color='blue', edgecolor='black', alpha=0.7, label='Distribution')
```

## Parameters

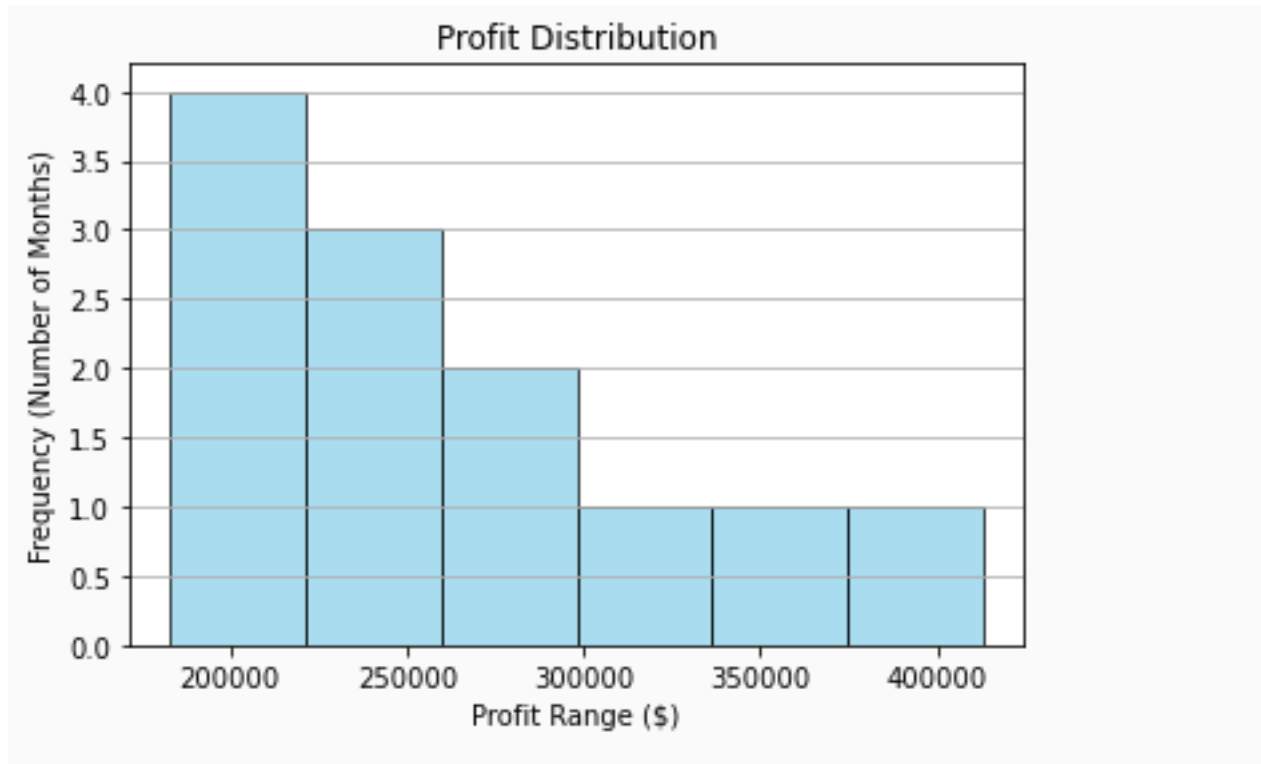
- `data`: Numerical values to analyze
- `bins`: Number of groups/intervals (or specific bin edges)
- `edgecolor`: Color of bin edges
- `alpha`: Transparency

## Example with Sales Data

Code:

```
# Distribution of monthly profits
profit_data = [211000, 183300, 224700, 222700, 209600, 201400,
               295500, 361400, 234000, 266700, 412800, 300200]

plt.hist(profit_data, bins=6, color='skyblue', edgecolor='black', alpha=0.7)
plt.title('Profit Distribution')
plt.xlabel('Profit Range ($)')
plt.ylabel('Frequency (Number of Months)')
plt.grid(axis='y')
plt.show()
```



## Best For

- Understanding data distribution
- Identifying patterns (normal, skewed)
- Outlier detection

## 5. plt.pie() - Pie Chart

### What it Does

Shows **parts of a whole** as slices of a pie. Each slice represents a proportion.

### Syntax

Code:

```
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors, startangle=90, explode=explode)
```

## Parameters

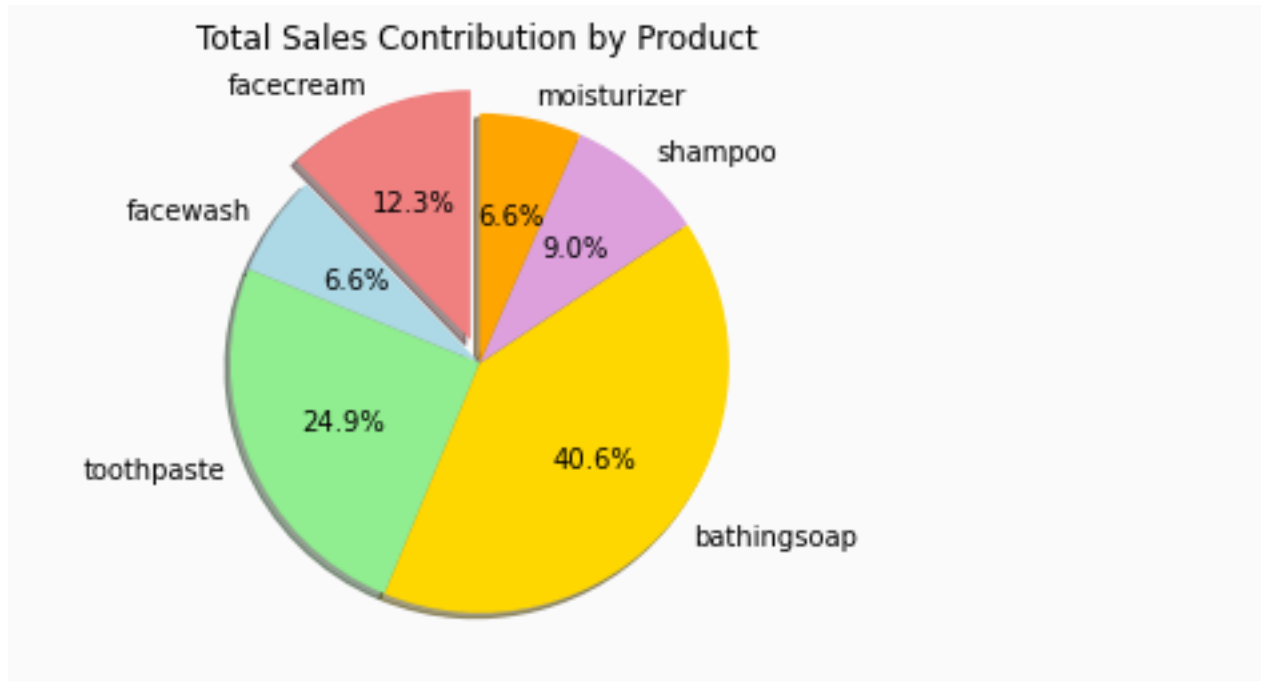
- `sizes`: Values for each slice
- `labels`: Names for each slice
- `autopct`: Format for percentage display
- `colors`: Colors for each slice
- `startangle`: Starting angle (90 = top)
- `explode`: Pull slices out (0 = normal, 0.1 = pulled out)

## Example with Sales Data

Code:

```
# Total sales contribution by product
products = ['facecream', 'facewash', 'toothpaste', 'bathingssoap', 'shampoo', 'moisturizer']
total_sales = [df[product].sum() for product in products]
colors = ['lightcoral', 'lightblue', 'lightgreen', 'gold', 'plum', 'orange']
explode = (0.1, 0, 0, 0, 0, 0) # Explode the first slice

plt.pie(total_sales, labels=products, autopct='%1.1f%%', colors=colors,
        startangle=90, explode=explode, shadow=True)
plt.title('Total Sales Contribution by Product')
plt.axis('equal') # Ensures pie is circular
plt.show()
```



### Best For

- Showing proportions/percentages
- Market share analysis
- Budget allocation

## Comparison Table

Plot Type	Best For	Key Parameters
<b>Line</b>	Trends over time	<code>linestyle</code> , <code>marker</code> , <code>linewidth</code>
<b>Scatter</b>	Relationships	<code>s(size)</code> , <code>c(color)</code> , <code>alpha</code>
<b>Bar</b>	Comparisons	<code>width</code> , <code>color</code> , multiple categories
<b>Histogram</b>	Distributions	<code>bins</code> , <code>edgecolor</code> , frequency
<b>Pie</b>	Proportions	<code>autopct</code> , <code>explode</code> , <code>startangle</code>



# plt.boxplot() - Box Plot (Box-and-Whisker Plot)

## What it Does

A **box plot** shows the **distribution** of numerical data through their **quartiles**. It's excellent for:

- Identifying **central tendency** and **spread**
- Detecting **outliers** and **skewness**
- Comparing **distributions** across groups

## Syntax

python

```
plt.boxplot(data, labels=group_names, notch=True, vert=True, patch_artist=True)
```

## Key Parameters

- **data**: Array or list of arrays of data values
- **labels**: Names for each dataset (for multiple boxes)
- **notch**: If True, creates notched boxes (shows confidence intervals)
- **vert**: If True (default), vertical boxes; False for horizontal
- **patch\_artist**: If True, fills boxes with color
- **showmeans**: If True, shows mean marker
- **meanline**: If True, shows mean as a line

## Understanding the Box Plot Components

text

Q3 (75th percentile)



Box: Interquartile Range (IQR)

Median (50th percentile) ———— ← Middle line

Q1 (25th percentile) ————

Whiskers: ———— ← Extends to 1.5\*IQR or min/max

Outliers: ○ ○ ← Data points beyond whiskers

## Example with Sales Data

python

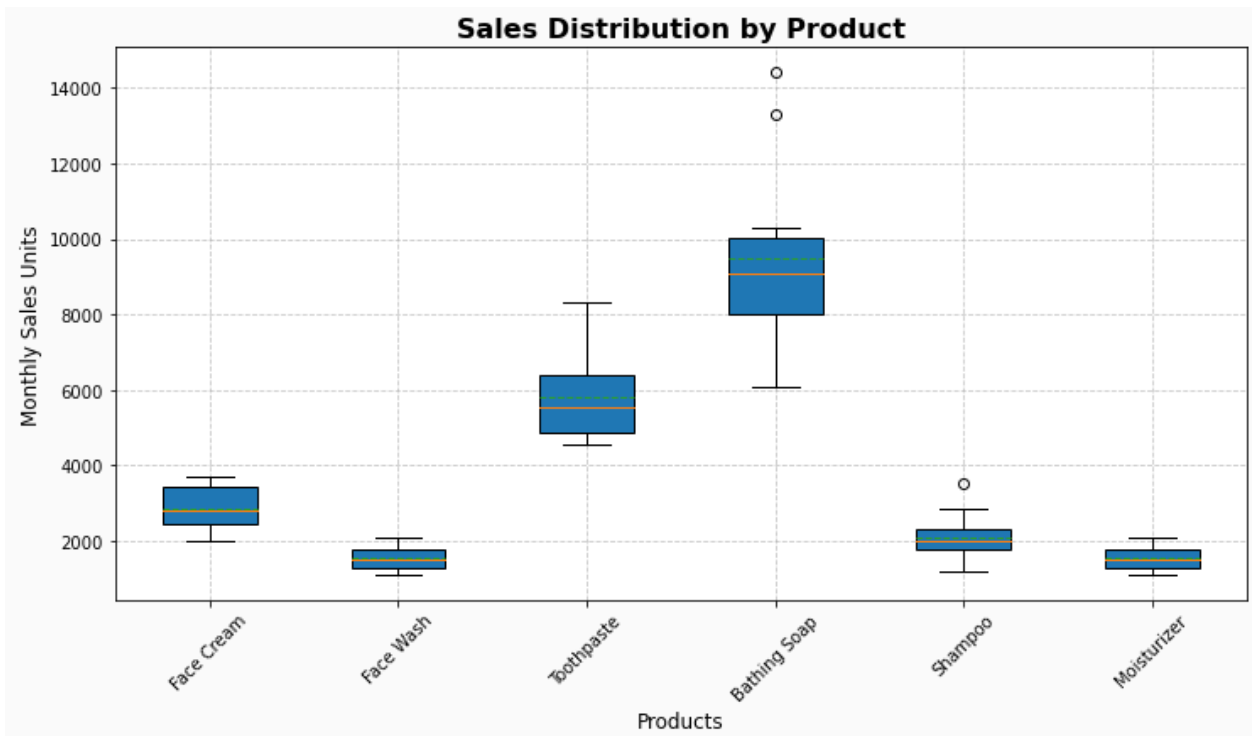
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Load the sales data
df = pd.read_csv('sales_data.csv')

# Create box plot for product sales
product_data = [df['facecream'], df['facewash'], df['toothpaste'],
                df['bathingsoap'], df['shampoo'], df['moisturizer']]
product_names = ['Face Cream', 'Face Wash', 'Toothpaste',
                 'Bathing Soap', 'Shampoo', 'Moisturizer']

plt.figure(figsize=(12, 6))
plt.boxplot(product_data, labels=product_names, patch_artist=True,
            showmeans=True, meanline=True)

plt.title('Sales Distribution by Product', fontsize=16, fontweight='bold')
plt.xlabel('Products', fontsize=12)
plt.ylabel('Monthly Sales Units', fontsize=12)
plt.xticks(rotation=45)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

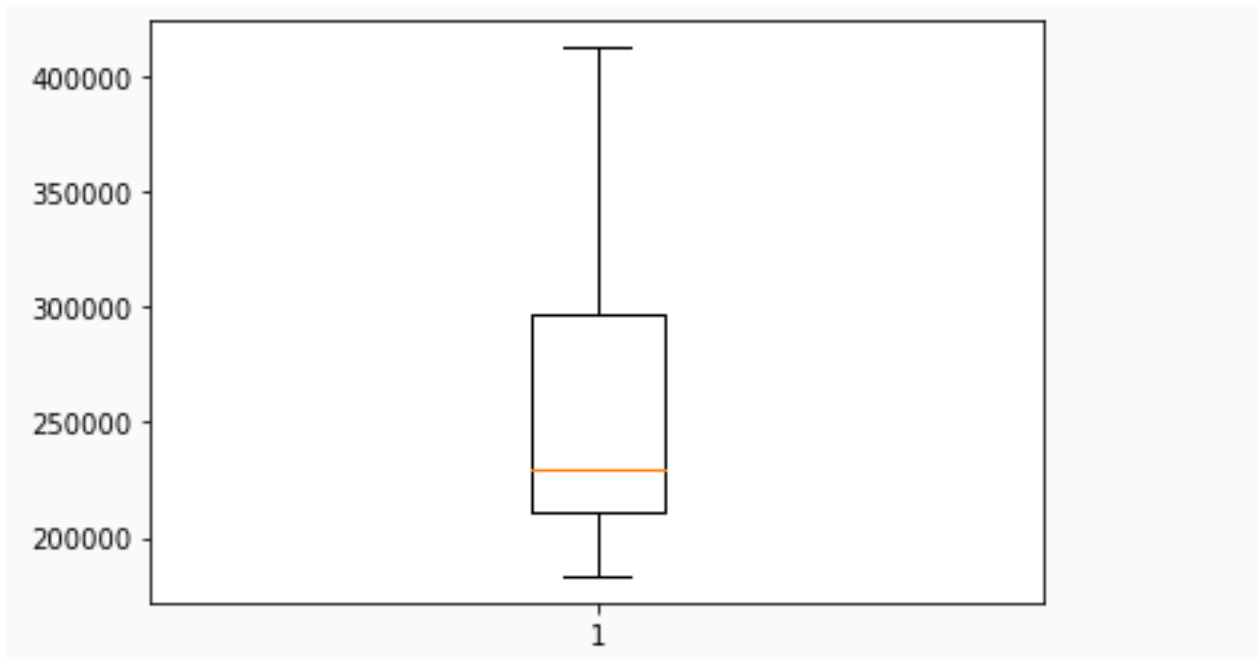


## Reading the Box Plot - What to Look For

```
# Extract statistics from a box plot
stats = plt.boxplot(df['total_profit'])

# Key values (for educational purposes)
q1 = np.percentile(df['total_profit'], 25) # 25th percentile
median = np.median(df['total_profit']) # 50th percentile (median)
q3 = np.percentile(df['total_profit'], 75) # 75th percentile
iqr = q3 - q1 # Interquartile Range

print(f"Q1 (25th percentile): ${q1:,.0f}")
print(f"Median: ${median:,.0f}")
print(f"Q3 (75th percentile): ${q3:,.0f}")
print(f"IQR: ${iqr:,.0f}")
```



### Components of chart:

```
plt.plot_type(x, y, parameters)  # ← Create the plot
plt.title()                     # ← Add title
plt.xlabel()                     # ← X-axis label
plt.ylabel()                     # ← Y-axis label
plt.legend()                     # ← Show legend (if needed)
plt.grid()                       # ← Add grid (optional)
plt.show()                       # ← Display it!
```

## 1. plt.title() - Adding Plot Titles

### What it does

Adds a title to the top of your plot to describe what the visualization represents.

### Syntax

Code:

```
plt.title('Your Title Here', fontsize=14, fontweight='bold', color='blue')
```

### Parameters

- **label**: The text string for the title (**required**)

- `fontsize`: Size of the font (default: 10)
- `fontweight`: 'normal', 'bold', 'light'
- `color`: Text color
- `loc`: Alignment ('left', 'center', 'right') - default: 'center'

## Examples

Code:

```
# Basic title
plt.title('Monthly Sales Data')

# Styled title
plt.title('2023 Product Performance', fontsize=16, fontweight='bold',
color='darkred')

# Left-aligned title
plt.title('Quarterly Revenue Analysis', loc='left')
```

## 2. plt.xlabel() & plt.ylabel() - Axis Labels

### What they do

Add descriptive labels to the x-axis and y-axis to explain what the data represents.

### Syntax

Code:

```
plt.xlabel('X-Axis Label', fontsize=12)
plt.ylabel('Y-Axis Label', fontsize=12)
```

### Parameters

- `xlabel/ylabel`: The text label (**required**)
- `fontsize`: Size of the font
- `color`: Text color
- `labelpad`: Padding between label and axis

## Examples

Code:

```
plt.xlabel('Month Number', fontsize=12, color='navy')
plt.ylabel('Sales Units', fontsize=12, color='navy')

# With padding
plt.xlabel('Time (months)', labelpad=15) # Extra space below label
```

## 3. plt.legend() - Showing the Legend

### What it does

Displays a legend that identifies different data series when you have multiple lines/plots.

### Syntax

Code:

```
plt.legend(loc='best', fontsize=10, frameon=True)
```

### Parameters

- `loc`: Location ('best', 'upper right', 'lower left', 'center', etc.)
- `fontsize`: Size of legend text
- `frameon`: Whether to draw a frame around legend (True/False)
- `title`: Add a title to the legend

**Important:** You must add `label` parameter to your plots first!

Code:

```
plt.plot(x, y, label='Product A') # ← Add label here
plt.plot(x, z, label='Product B') # ← Add label here
plt.legend() # ← Then show legend
```

## Examples

Code:

```
# Basic legend
plt.legend()
```

```
# Custom positioned legend
plt.legend(loc='upper left', fontsize=9)

# Legend with title and no frame
plt.legend(loc='center', title='Products', frameon=False)
```

## 4. plt.grid() - Adding Grid Lines

### What it does

Adds grid lines to make it easier to read values from the plot.

### Syntax

Code:

```
plt.grid(True, linestyle='--', alpha=0.7, color='gray')
```

### Parameters

- `visible`: True/False to show/hide grid
- `axis`: 'both', 'x', or 'y' (which axes to grid)
- `linestyle**`: '-', '--', ':', '-.' (line styles)
- `alpha`: Transparency (0.0 to 1.0)
- `color`: Grid line color

### Examples

Code:

```
# Basic grid
plt.grid(True)

# Dashed grid with transparency
plt.grid(True, linestyle='--', alpha=0.5)

# Only horizontal grid lines
plt.grid(True, axis='y') # Only y-axis grid lines
```

## 5. plt.show() - Displaying the Plot

### What it does

**Displays the plot** you've created. This is essential to actually see your visualization.

### Syntax

Code:

```
plt.show()
```

### Important Notes

- **Always call this last** after all your plotting commands
- In Jupyter notebooks, you might use `%matplotlib inline` instead
- Without this, your plot won't display!

### Example Workflow

Code:

```
# 1. Create your plot
plt.plot(df['month_number'], df['total_profit'])
plt.title('Monthly Profit')
plt.xlabel('Month')
plt.ylabel('Profit ($)')
plt.grid(True)

# 2. Show it!
plt.show() # ← This makes it appear
```

## Complete Example with All Elements

Code:

```
import matplotlib.pyplot as plt
import pandas as pd
```



```
# Load data
df = pd.read_csv('sales_data.csv')

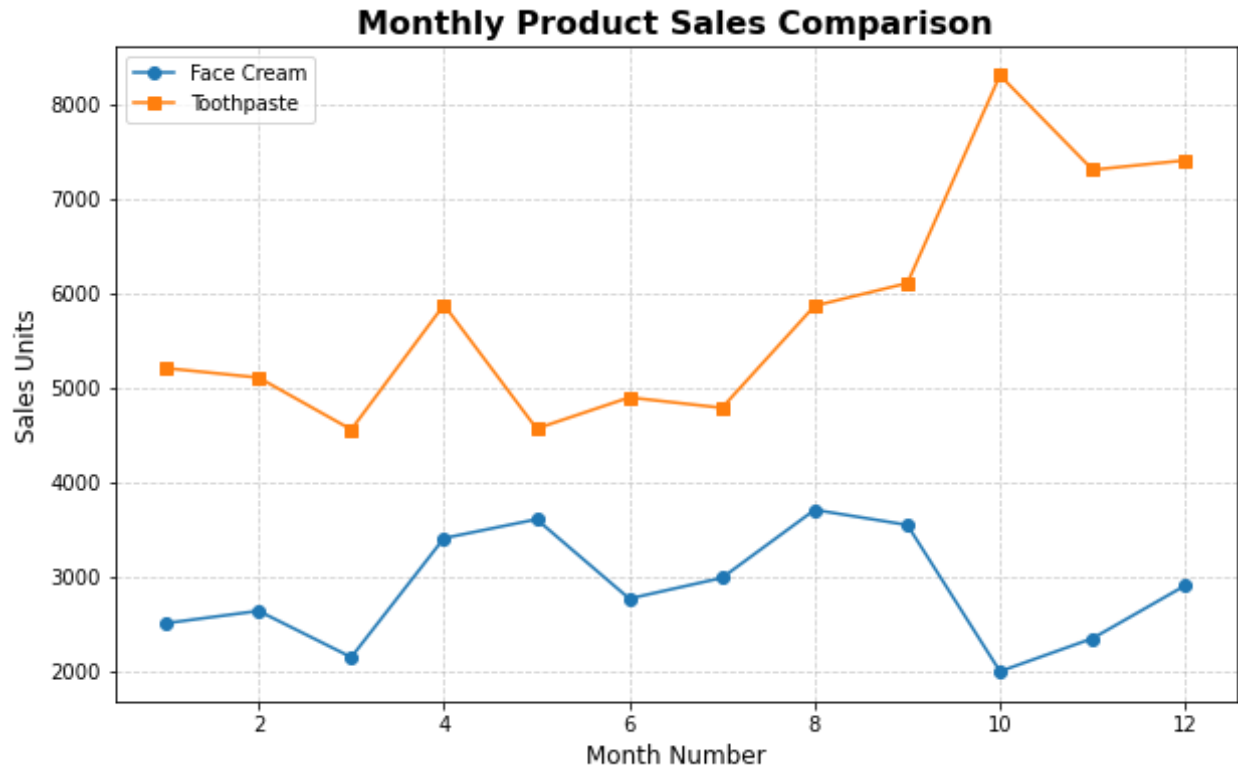
# Create plot with all labeling elements
plt.figure(figsize=(10, 6))

# Plot data with labels for legend
plt.plot(df['month_number'], df['facecream'], label='Face Cream', marker='o')
plt.plot(df['month_number'], df['toothpaste'], label='Toothpaste', marker='s')

# Add titles and labels
plt.title('Monthly Product Sales Comparison', fontsize=16, fontweight='bold')
plt.xlabel('Month Number', fontsize=12)
plt.ylabel('Sales Units', fontsize=12)

# Add grid and legend
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend(loc='upper left', fontsize=10)

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



## Common Mistakes

1. **Forgetting `plt.show()`** → Plot doesn't display
2. **Adding legend without labels** → Legend shows empty
3. **Wrong order** → Add title/labels before `plt.show()`
4. **Spelling mistakes** → `plt.lable()` instead of `plt.label()`

## Memory Aid :

"Title Labels Legend Grid Show"

→ The order you typically add elements before displaying!

## Understanding `alpha` in Data Visualization

### What is Alpha?

**Alpha** is a parameter that controls the **transparency** or **opacity** of visual elements in plots. It ranges from:

- **0.0** (completely transparent/invisible)
- **1.0** (completely opaque/solid)

## Syntax Examples

Code:

```
plt.scatter(x, y, alpha=0.5)           # 50% transparent
plt.bar(categories, values, alpha=0.7) # 70% opaque
plt.plot(x, y, alpha=0.3)              # 70% transparent
```

## Why Use Alpha?

### 1. Avoid Overplotting

When data points overlap, alpha helps see density:

Code:

```
# Without alpha - can't see overlapping points
plt.scatter(x, y, color='blue')

# With alpha - shows density of points
plt.scatter(x, y, color='blue', alpha=0.3)
```

### 2. Multiple Layer Visualization

When plotting multiple datasets:

Code:

```
plt.scatter(x1, y1, color='red', alpha=0.5, label='Group A')
plt.scatter(x2, y2, color='blue', alpha=0.5, label='Group B')
# You can see where points overlap (purple areas)
```

### 3. Histograms and Bars

Code:

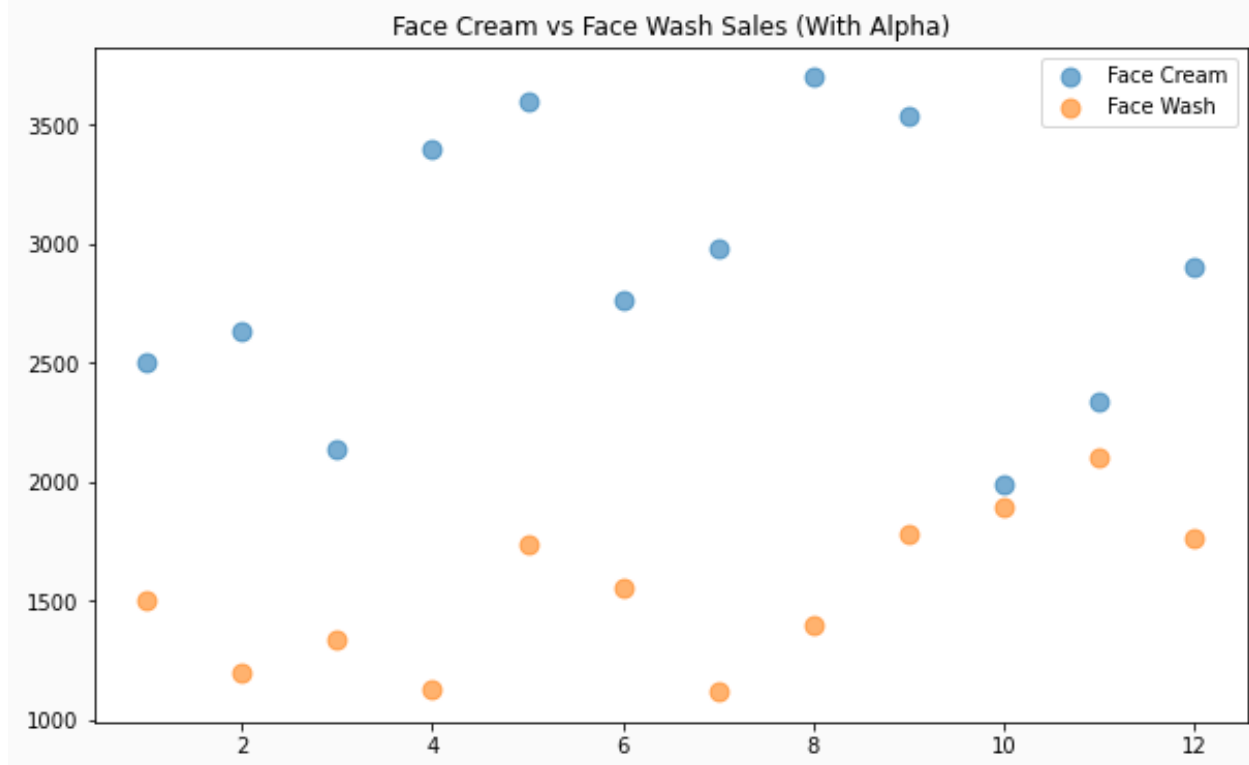
```
plt.hist(data1, alpha=0.6, label='Dataset 1')
plt.hist(data2, alpha=0.6, label='Dataset 2')
# Both distributions are visible even when overlapping
```

## Practical Examples with Sales Data

### Example 1: Overlapping Product Sales

Code:

```
plt.figure(figsize=(10, 6))
plt.scatter(df['month_number'], df['facecream'], alpha=0.6, label='Face Cream', s=80)
plt.scatter(df['month_number'], df['facewash'], alpha=0.6, label='Face Wash', s=80)
plt.title('Face Cream vs Face Wash Sales (With Alpha)')
plt.legend()
plt.show()
```



### Example 2: Transparent Bar Charts

Code:

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

```
plt.bar(df['month_number'] - 0.2, df['toothpaste'], width=0.4, alpha=0.7, label='Toothpaste')
plt.bar(df['month_number'] + 0.2, df['shampoo'], width=0.4, alpha=0.7, label='Shampoo')
plt.title('Toothpaste vs Shampoo Sales')
plt.legend()
plt.show()
```

### Example 3: Area Plot with Alpha

Code:

```
plt.figure(figsize=(10, 6))
plt.fill_between(df['month_number'], df['total_profit'], alpha=0.4, color='green', label='Profit')
plt.plot(df['month_number'], df['total_profit'], alpha=0.8, color='darkgreen')
plt.title('Profit Trend with Transparency')
plt.legend()
plt.show()
```

### Recommended Alpha Values

Use Case	Recommended Alpha	Why
<b>Scatter plots</b> with many points	0.1 - 0.3	Shows density without overwhelming
<b>Bar charts</b> with few bars	0.7 - 0.9	Maintains color vibrancy
<b>Overlapping histograms</b>	0.5 - 0.7	Allows seeing both distributions
<b>Background elements</b>	0.1 - 0.3	Keeps focus on main data

Use Case	Recommended Alpha	Why
Foreground elements	0.8 - 1.0	Ensures main data is clear

## Alpha with Different Plot Types

Code:

```
# Line plot
plt.plot(x, y, alpha=0.7)

# Bar plot
plt.bar(x, y, alpha=0.8)

# Histogram
plt.hist(data, alpha=0.6)

# Scatter plot
plt.scatter(x, y, alpha=0.4)

# Fill between
plt.fill_between(x, y1, y2, alpha=0.3)
```

**Alpha** is one of the most useful parameters for creating clear, professional-looking visualizations, especially when dealing with overlapping data or multiple datasets!

### What Are Subplots?

- Normally, `plt.plot()` or `plt.hist()` creates **one chart per figure**.
- **Subplots** let you put **multiple charts inside a single figure**, arranged in rows and columns.
- This is very useful when you want to **compare plots side by side**.

Think of a subplot like a small "box" inside the big canvas.

### Basic Syntax

```
plt.subplot(nrows, ncols, index)
```

- **nrows** → number of rows in the figure
- **ncols** → number of columns in the figure
- **index** → which subplot you are drawing on (count starts from 1, left to right, top to bottom)

### Example: Two Plots Side by Side

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

```
x = [1, 2, 3, 4]
```

```
y1 = [10, 20, 25, 30]
```

```
y2 = [5, 15, 10, 25]
```

```
plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st subplot
```

```
plt.plot(x, y1)
```

```
plt.title("Line Chart")
```

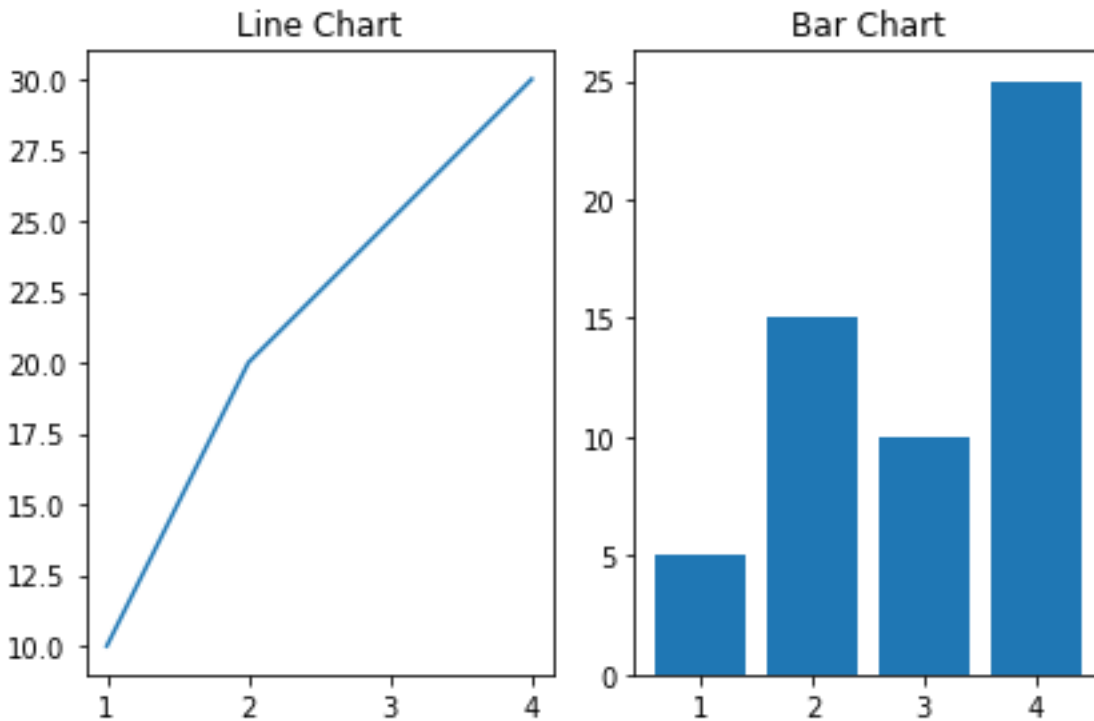
```
plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd subplot
```

```
plt.bar(x, y2)
```

```
plt.title("Bar Chart")
```

```
plt.tight_layout() # adjusts spacing so plots don't overlap
```

plt.show()



### What happens:

- The figure is divided into 1 row × 2 columns.
- First chart goes into position 1, second chart goes into position 2.
- `tight_layout()` makes sure titles and axes don't overlap.

### Using `plt.subplots()` (Recommended)

A more modern way is to use `plt.subplots()`, which creates figure + axes in one go:

```
fig, ax = plt.subplots(1, 2) # 1 row, 2 columns
```

```
ax[0].plot(x, y1)
```



```
ax[0].set_title("Line Chart")
```

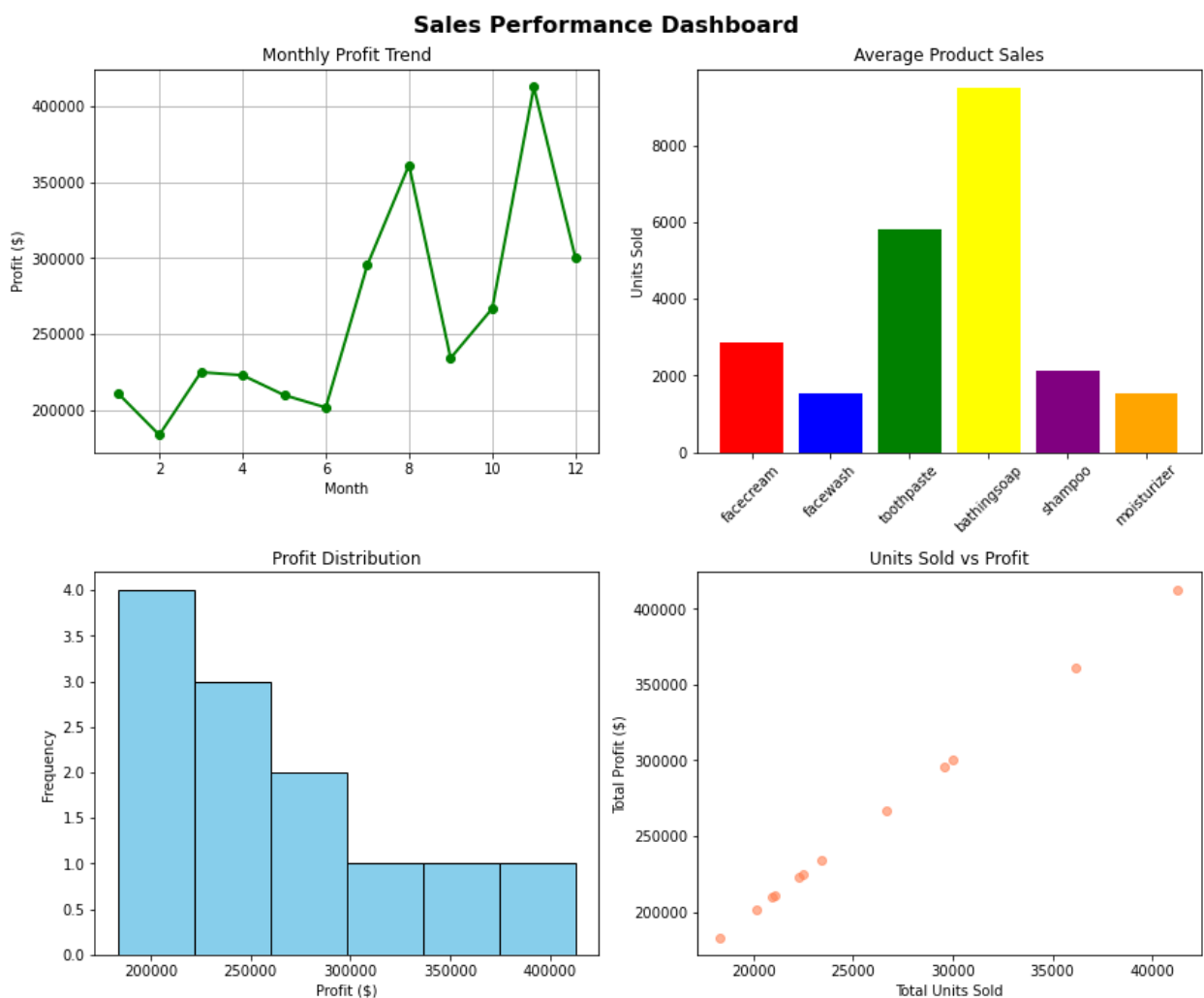
```
ax[1].bar(x, y2)
```

```
ax[1].set_title("Bar Chart")
```

```
plt.tight_layout()
```

```
plt.show()
```

- ax is an array of axes, so you can call ax[0], ax[1], etc.
- This approach is cleaner when you have many subplots.



## When to Use Subplots

- Comparing **two or more related plots**.
- Showing **different views** of the same data (e.g., histogram + boxplot).
- Combining multiple metrics in one figure for a report

## Short Answer Questions (2-5 marks)

1. What is the purpose of `plt.xlabel()` and `plt.ylabel()`?
2. Explain the difference between `plt.plot()` and `plt.scatter()`.
3. Why do we use `import matplotlib.pyplot as plt` instead of importing `matplotlib` directly?
4. What is the function of `plt.legend()`?
5. What type of data is best represented by a histogram?
6. Why is data preprocessing important before visualization?
7. What are three common methods for handling missing values?
8. How do you identify outliers in a dataset?

## Long Answer Questions (5-10 marks)

1. Explain the step-by-step process to create a line chart with proper labels, title, and grid using `matplotlib`.
2. Compare and contrast scatter plots, line charts, and bar charts. When would you use each type?
3. Describe how to create subplots with 2 rows and 2 columns using `matplotlib`. Include code examples.
4. What are all the components of a complete `matplotlib` visualization? Explain each component's purpose.
5. How would you customize the appearance of a plot (colors, markers, line styles, etc.)? Provide examples.

6. Explain the complete data preprocessing pipeline with examples for each step.
7. What are the different types of missing data? How would you handle each type?
8. Describe various methods for outlier detection and treatment. When would you use each method?