UNIT 5

Python packages

UNIT-5: Python Packages: Simple programs using the built-in functions of packages matplotlib, numpy, pandas etc. GUI Programming: Tkinter introduction, Tkinter and Python Programming, Tk Widgets, Tkinter examples. Python programming with IDE.

5.1 INTRODUCTIONS TO PYTHON PACKAGES:

A **Python package** is a directory (folder) that contains related Python files called modules along with a special file called along with a special __init__.py file.

- A **module** is a single **.py** file containing Python code (functions, classes, variables).
- A package is a directory containing multiple modules and a special file called **__init__.py** (can be empty) that tells Python it's a package.

BENEFITS OF PYTHON PACKAGES

1. **Organized Code Structure**
   - *Groups related modules together, making projects easier to navigate and maintain.*
2. **Reusability**
   - *Write code once and reuse it across multiple projects without duplication.*
3. **Modularity**
   - *Breaks large programs into smaller, manageable components.*
4. **Ease of Sharing**
   - *Distribute packages via PyPI or other repositories so others can install and use them easily.*
5. **Namespace Management**
   - *Prevents naming conflicts by isolating modules in separate namespaces.*
6. **Extensibility**
   - *Easily integrate third-party packages to expand Python's capabilities.*
7. **Collaboration Friendly**
   - *Allows multiple developers to work on different modules within the same package without interference.*

5.2 CLASSIFICATION OF PYTHON PACKAGES

Python packages are grouped into categories based on what they are used for. This helps developers choose the right tool for their task.

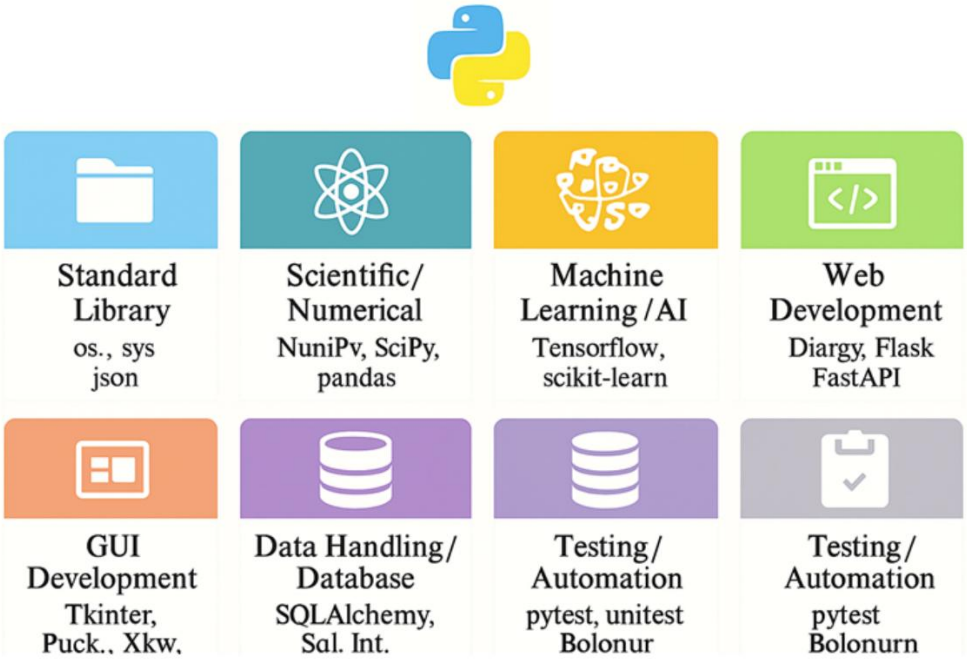


Figure 5.1: Major Python Package Classifications

Table: Classification of Python Packages

Category	Package Examples	Explanation	Purpose
Standard Library	**math, datetime, os, sys, random, json**	Built-in with Python, no installation needed; handles math, date/time, file handling, system tasks, and data formats.	Handle basic tasks like math operations, date/time, file handling, and system interaction.
Scientific & Numerical	**numpy, scipy, matplotlib, pandas, sympy**	Used for calculations, data analysis, and visualization.	Perform numerical computations, analyze datasets, and visualize results.
Machine Learning & AI	**scikit-learn, tensorflow, keras, xgboost, nltk**	Used to build models for predictions, deep learning, and natural language processing.	Create machine learning models, deep learning networks, and natural language processing systems.
Web Development	**flask, django, requests, beautifulsoup4, fastapi**	Used to create websites, APIs, and scrape web data.	Develop web applications, connect to web services, and scrape data from websites.
GUI Development	**tkinter, PyQt, Kivy, wxPython**	Used to create desktop applications with graphical interfaces.	Build apps with buttons, menus, forms, and interactive elements.
Data Handling & Database	**sqlite3, sqlalchemy, openpyxl, csv**	Used to store, read, and manage data in files or databases.	Handle databases, read/write Excel and CSV files.
Testing & Automation	**unittest, pytest, selenium, robotframework**	**Used to test code and automate repetitive tasks.**	Ensure code works correctly and automate repetitive processes.

5.3 BUILT-IN FUNCTIONS OF PYTHON PACKAGES

In Python, each package (or module) comes with its own built-in functions, these are functions provided by that package to perform specific tasks without requiring you to write the logic from scratch.

The below **table** summarizing some commonly used packages and their important built-in functions with simple explanations:

Table: Commonly used packages and their important built-in functions

Package	Common Built-in Functions	Simple Explanation
Basic Built-in Functions	(print(), len(), type(), id(), dir(), help())	The **basic built-in functions** in Python provide quick ways to display output, check types, get object details, and access documentation. They simplify coding by offering ready-to-use tools for debugging, exploring, and managing data.
math	sqrt(x), pow(x, y), factorial(x), ceil(x), floor(x)	Perform mathematical calculations like square root, powers, factorial, rounding up/down.
random	random(), randint(a, b), choice(seq), shuffle(seq)	Generate random numbers, choose random items, shuffle data.
datetime	date.today(), datetime.now(), timedelta(days=n)	Work with dates and times, get current date/time, calculate time differences.
os	getcwd(), listdir(path), mkdir(name), remove(file)	Interact with the operating system folders, files, paths.
sys	exit(), getsizeof(obj), version	Manage Python runtime, exit program, check memory size, get Python version.
statistics	mean(data), median(data), mode(data), stdev(data)	Perform statistical calculations easily.
json	dump(obj, file), load(file), dumps(obj), loads(str)	Work with JSON data—read, write, and convert between JSON and Python objects.
re	search(pattern, string), match(pattern, string), findall(pattern, string)	Use regular expressions to search, match, and extract patterns in text.
collections	Counter(seq), defaultdict(type), namedtuple(name, fields)	Special data structures for counting, default values, and structured tuples.

5.4 MATPLOTLIB – BUILT-IN FUNCTIONS

- Matplotlib's **pyplot** module is used to create graphs and charts in Python.
- Always import the module first: ***import matplotlib.pyplot as plt***
- Use ***plt.show()*** at the end to render the plot.

Table: Summary of built-in Functions of matplotlib.pyplot

Function	Syntax Example	Explanation
plot()	plt.plot(x, y)	Draws a line graph between x and y values.
bar()	plt.bar(x, height)	Creates a vertical bar chart.
barh()	plt.barh(y, width)	Creates a horizontal bar chart.
scatter()	plt.scatter(x, y)	Plots individual data points as a scatter plot.
hist()	plt.hist(data, bins=...)	Displays a histogram showing frequency distribution.
pie()	plt.pie(sizes, labels=...)	Creates a pie chart to show proportions.
xlabel()	plt.xlabel("X-axis Label")	Adds a label to the X-axis.
ylabel()	plt.ylabel("Y-axis Label")	Adds a label to the Y-axis.
title()	plt.title("Plot Title")	Adds a title to the plot.
legend()	plt.legend()	Displays a legend for labeled plot elements.

grid()	plt.grid(True)	Adds grid lines to the plot for better readability.
xlim()	plt.xlim(min, max)	Sets limits for the X-axis.
ylim()	plt.ylim(min, max)	Sets limits for the Y-axis.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *plot()*

SYNTAX

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

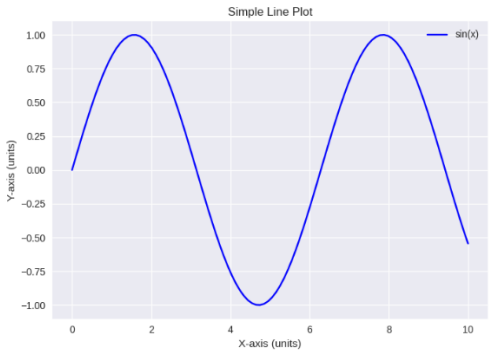
Explanation:

- plot() draws a line graph connecting the points defined by the x and y arrays.
- It's ideal for visualizing trends, relationships, or changes over a continuous range.
- Optional arguments include:
  - ✓ label: for legend
  - ✓ color: line color
  - ✓ marker: symbol at each data point (e.g., 'o', 'x', etc.)
  - ✓ linestyle: line style ('-', '--', ':', etc.)

Example Program: Simple Line Plot

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

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y, label='sin(x)', color='blue')
plt.title('Simple Line Plot')
plt.xlabel('X-axis (units)')
plt.ylabel('Y-axis (units)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



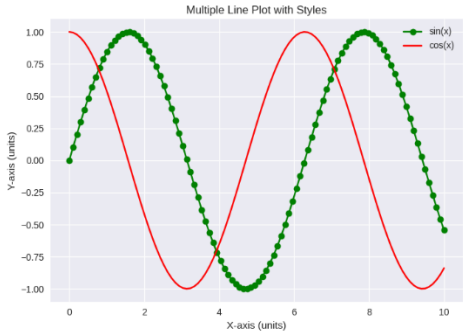
Explanation:

- np.linspace(0, 10, 100) generates 100 evenly spaced values from 0 to 10.
- np.sin(x) computes the sine of each value.
- The plot shows a smooth sine wave.
- Labels, title, legend, and grid enhance readability.

Example Program: Simple Line Plot

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

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

plt.plot(x, y1, label='sin(x)', color='green', marker='o')
plt.plot(x, y2, label='cos(x)', color='red', marker='x')
plt.title('Multiple Line Plot with Styles')
plt.xlabel('X-axis (units)')
plt.ylabel('Y-axis (units)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



Explanation:

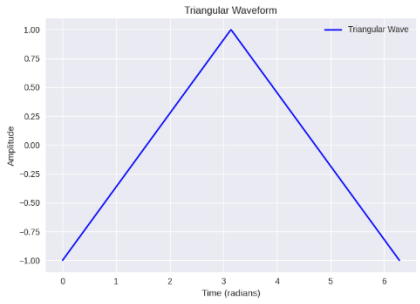
- Two functions (sin(x) and cos(x)) are plotted on the same graph.
- Different colors and markers distinguish the lines.
- Useful for comparing multiple datasets visually.

Example Program: Simple Line Plot

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import sawtooth

# Generate time values
t = np.linspace(0, 2 * np.pi, 500)

# Generate triangular waveform
triangular_wave = sawtooth(t, width=0.5)

# Plotting
plt.plot(t, triangular_wave, label='Triangular Wave', color='blue')
plt.title('Triangular Waveform')
plt.xlabel('Time (radians)')
plt.ylabel('Amplitude')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

OUTPUT:



Explanation:

- sawtooth(t, width=0.5) generates a symmetric triangular wave.
- width=0.5 makes the waveform peak in the middle of each cycle.
- The plot shows a repeating linear rise and fall pattern.
- Useful in signal processing and waveform analysis.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *bar()*

SYNTAX

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

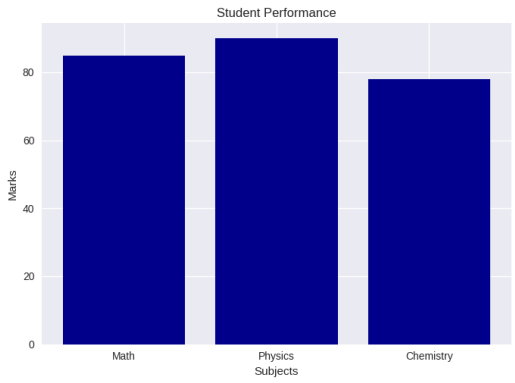
Explanation:

- bar() creates a **vertical bar chart**.
- x: categories or positions on the X-axis.
- height: values or heights of the bars.
- Optional arguments:
  - ✓ color: bar color
  - ✓ width: width of each bar
  - ✓ label: for legend
  - ✓ align: alignment of bars ('center' or 'edge')

Example Program: Basic Bar Chart

```
import matplotlib.pyplot as plt

subjects = ['Math', 'Physics', 'Chemistry']
marks = [85, 90, 78]

plt.bar(subjects, marks, color='darkblue')
plt.xlabel('Subjects')
plt.ylabel('Marks')
plt.title('Student Performance')
plt.grid(True, axis='y')
plt.show()
```

OUTPUT:



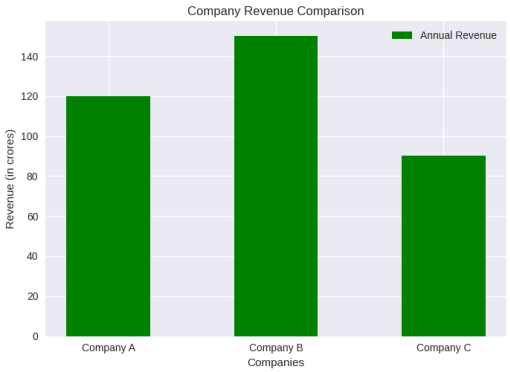
Explanation:

- Each subject is represented by a vertical bar.
- grid(True, axis='y') adds horizontal grid lines for better readability.

Example Program: Bar Chart with Custom Width and Labels

```
import matplotlib.pyplot as plt

companies = ['Company A', 'Company B', 'Company C']
revenue = [120, 150, 90]

plt.bar(companies, revenue, color='green', width=0.5, label='Annual Revenue')
plt.xlabel('Companies')
plt.ylabel('Revenue (in crores)')
plt.title('Company Revenue Comparisxon')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



Explanation:

- Bars are narrower (width=0.5) and colored green.
- legend() displays the label for the bars.
- Useful for comparing financial or categorical data.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *barh()*

- barh() in **Matplotlib** is used to create **horizontal bar charts**.

SYNTAX

```
plt.barh(y, width)
```

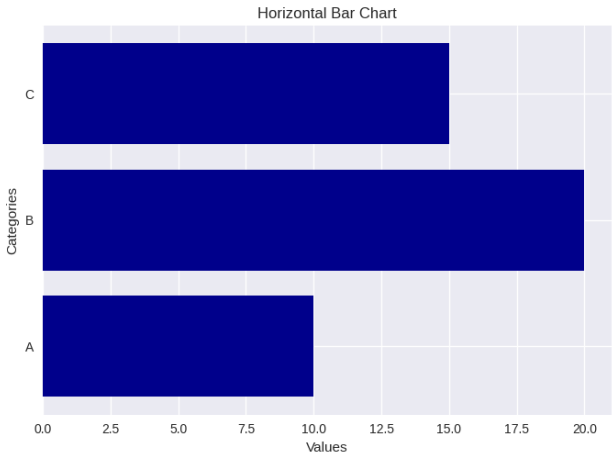
Explanation:

- y: Specifies the y-coordinates of the bars (i.e., the positions on the vertical axis where bars are placed). This can be a list of values or categories.
- width: Specifies the lengths (widths) of the horizontal bars, representing the data values.

Example Program: Simple Line Plot

```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C']
values = [10, 20, 15]
plt.barh(categories, values, color='darkblue')
plt.xlabel('Values')
plt.ylabel('Categories')  # Added for clarity, since categories are on the y-axis
plt.title('Horizontal Bar Chart')
plt.show()
```

OUTPUT:



Explanation:

Code	Purpose	Output/Effect
import matplotlib.pyplot as plt	Brings in the plotting library with alias	Makes plotting functions available
categories = ['A', 'B', 'C']	Creates list of category labels	Stores ['A', 'B', 'C'] in memory
values = [10, 20, 15]	Creates list of numerical data	Stores [10, 20, 15] in memory
plt.barh(categories, values, color='darkblue')	Generates horizontal bars	Creates 3 dark blue horizontal bars
plt.xlabel('Values')	Adds label to x-axis	Displays "Values" below chart
plt.ylabel('Categories')	Adds label to y-axis	Displays "Categories" on left side
plt.title('Horizontal Bar Chart')	Adds title to chart	Displays title at top of chart
plt.show()	Renders and shows the chart	Opens chart window/displays chart

Data Mapping:

Category	Value	Bar Length
A	10	Short
B	20	Long
C	15	Medium

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *scatter()*

The matplotlib.pyplot.scatter function in Matplotlib is used to **create a scatter plot of individual data points on a 2D plane**. Below is the detailed syntax, including all key parameters, as provided by the Matplotlib library.

SYNTAX

```
matplotlib.pyplot.scatter(
    x,
    y,
    s=None,
    c=None,
    marker=None,
    cmap=None,
    norm=None,
    vmin=None,
    vmax=None,
    alpha=None,
    linewidths=None,
    edgecolors=None,
    plotnonfinite=False,
    *,
    data=None,
    **kwargs
)
```

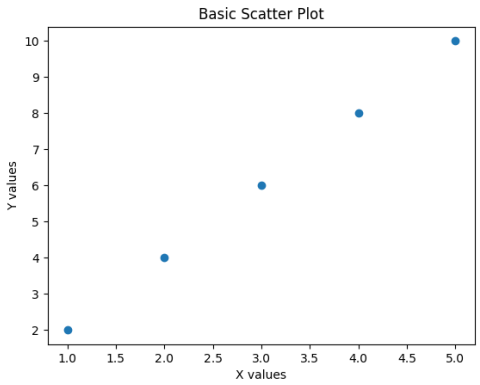
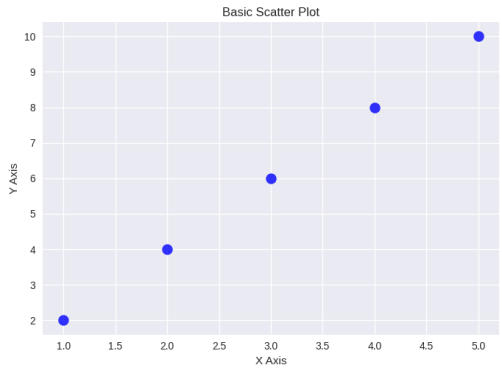
Explanation:

Parameter	Description
x	Sequence of x-values
y	Sequence of y-values
s	Size of points (scalar or array)
c	Color(s) of points (single color, array, or colormap)
marker	Marker style ('o', 'x', '^', etc.)
cmap	Colormap for mapping numeric c values
norm	Normalization for colormap scaling
vmin, vmax	Color scale limits for colormap
alpha	Transparency level (0.0–1.0)
linewidths	Marker edge width
edgecolors	Color of marker edges
plotnonfinite	Whether to plot points with NaN/Inf values
data	Optional data source (dict, DataFrame, etc.)
kwargs	Additional customization options

Example Program: Simple Line Plot

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.scatter(x, y)
plt.title("Basic Scatter Plot")
plt.xlabel("X values")
```

```
import matplotlib.pyplot as plt

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

# Create scatter plot
```

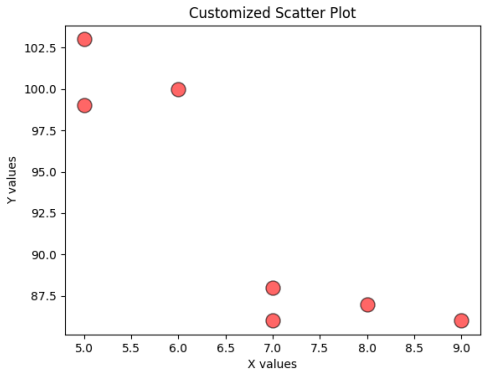
plt.ylabel("Y values") *plt.show()*	*plt.scatter(x, y, c='blue', s=100, marker='o', alpha=0.8)* *plt.title("Basic Scatter Plot")* *plt.xlabel("X Axis")* *plt.ylabel("Y Axis")* *plt.grid(True)* *plt.show()*
OUTPUT: 	**OUTPUT:** 
Explanation: • **Data:** The x and y lists define the coordinates of the points. • **Parameters:** c='blue': Sets all markers to blue. s=100: Sets a uniform marker size. marker='o': Uses circular markers. alpha=0.8: Makes markers slightly transparent. • **Customization:** The plt.title(), plt.xlabel(), plt.ylabel(), and plt.grid(True) enhance the plot's clarity.	**Explanation:** • **Data:** The x and y lists define the coordinates of the points, forming pairs that indicate a linear relationship (y = 2x). • **Parameters:** **c:** Not specified, defaults to blue markers based on Matplotlib's default style. **s:** Not specified, uses default marker size (typically 20 points^2). **marker:** Not specified, defaults to circular markers ('o'). **alpha:** Not specified, defaults to fully opaque (1.0). • **Customization:** The plt.title(), plt.xlabel(), and plt.ylabel() add a title and axis labels to improve readability. No grid or legend is included, as plt.grid() or label parameters weren't used.

Example Program: Scatter Plot with Varying Marker Sizes and Colors

import matplotlib.pyplot as plt *x = [5, 7, 8, 7, 6, 9, 5]* *y = [99, 86, 87, 88, 100, 86, 103]*	*import matplotlib.pyplot as plt* *# Sample data* *x = [1, 2, 3, 4, 5]* *y = [3, 5, 2, 8, 7]*

```
plt.scatter(x, y, c="red", s=150, alpha=0.6,
edgecolors="black")
plt.title("Customized Scatter Plot")
plt.xlabel("X values")
plt.ylabel("Y values")
        plt.show()
```

OUTPUT:

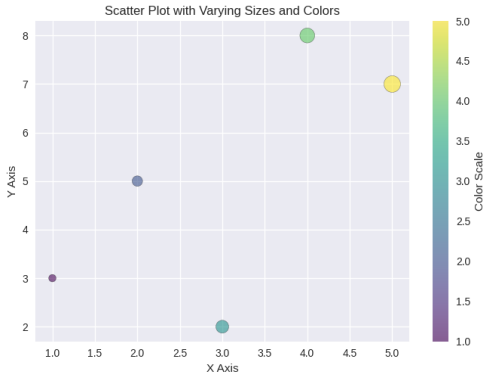


Explanation:

- Red dots, semi-transparent (alpha=0.6), with black edges.
- Dot size is larger (s=150).
- Here, visual customization is added using c for color, s for size, alpha for transparency, and edgecolors for border color.

```
sizes = [50, 100, 150, 200, 250]
colors = [1, 2, 3, 4, 5]

# Create scatter plot
plt.scatter(x, y, s=sizes, c=colors, cmap='viridis',
alpha=0.6, edgecolors='black')
plt.title("Scatter Plot with Varying Sizes and
Colors")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.colorbar(label='Color Scale')
plt.grid(True)
plt.show()
```

OUTPUT:



Explanation:

Data:

The x and y lists define the coordinates of the points, representing a non-linear dataset. The sizes list specifies varying marker sizes, and the colors list provides values mapped to colors via a colormap.

Parameters:

- **s=sizes:** Sets different marker sizes (50 to 250 points^2) for each point to highlight variation.
- **c=colors:** Maps each point's color to a value in the colors list using the 'viridis' colormap.
- **cmap='viridis':** Uses the 'viridis' colormap to translate numeric color values into a gradient.
- **alpha=0.6:** Makes markers semi-transparent to improve visibility if points overlap.
- **edgecolors='black':** Adds black edges to markers for better contrast.

	Customization: The plt.title(), plt.xlabel(), and plt.ylabel() add a title and axis labels for clarity. The plt.grid(True) adds a grid, and plt.colorbar(label='Color Scale') includes a colorbar to show the color mapping.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *hist()*

The hist() function in matplotlib.pyplot is used to **compute and plot histograms**, which are **graphical representations of the distribution of numerical data**. It divides the data into equally spaced **intervals called bins** and counts how many data points fall into each bin, **displaying the frequency as bars.**

SYNTAX

```
matplotlib.pyplot.hist(
    x,
    bins=None,
    range=None,
    density=False,
    weights=None,
    cumulative=False,
    bottom=None,
    histtype='bar',
    align='mid',
    orientation='vertical',
    rwidth=None,
    log=False,
    color=None,
    label=None,
    stacked=False,
    data=None,
    **kwargs
)
```

Explanation:

- x: Input data (array-like or sequence of arrays) to plot.
- bins: Number of bins (int), bin edges (sequence), or binning method (str, e.g., 'auto'). Default: 10 bins.
- range: Tuple (lower, upper) to set the data range for binning. Default: Uses data min/max.
- density: If True, normalizes histogram to a probability density (area sums to 1). Default: False.
- weights: Array of weights for each data point, same shape as x. Default: None (equal weights).
- cumulative: If True, plots cumulative histogram; if -1, reverse cumulative. Default: False.
- bottom: Starting height for bars (scalar or array). Useful for stacking. Default: 0.
- histtype: Histogram style ('bar', 'barstacked', 'step', 'stepfilled'). Default: 'bar'.
- align: Bar alignment ('left', 'mid', 'right'). Default: 'mid'.

- orientation: Bar direction ('vertical' or 'horizontal'). Default: 'vertical'.
- rwidth: Relative bar width (float < 1 for gaps). Default: Bars touch.
- log: If True, uses logarithmic scale for y-axis. Default: False.
- color: Bar color(s) (single color or list). Default: Uses default color cycle.
- label: Legend label for the histogram. Default: None.
- stacked: If True, stacks multiple datasets (with histtype='barstacked'). Default: False.
- data: Optional dictionary/DataFrame to reference x by key/column. Default: None.
- kwargs: Additional styling options (e.g., alpha, edgecolor, linewidth).

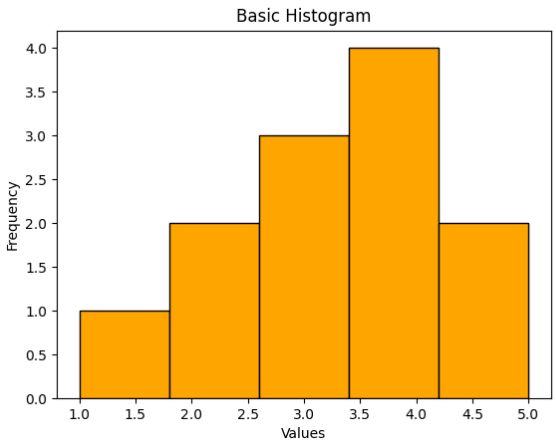
Return Value

- **A tuple:**
  - ✓ **n:** Bin counts.
  - ✓ **bins:** Bin edges.
  - ✓ **patches:** List of Patch objects (the drawn bars).

Example Program: Basic Histogram Plot

```
import matplotlib.pyplot as plt

# Sample data
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5]

# Plot histogram
plt.hist(data, bins=5, color='orange', edgecolor='black')

# Add labels and title
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Basic Histogram")
plt.show()
```

OUTPUT:

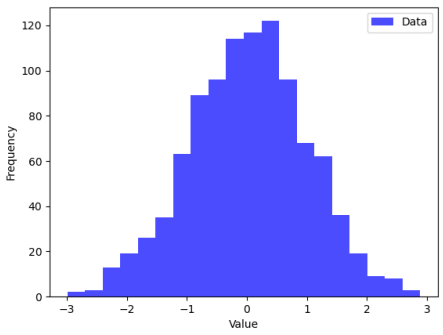


Example Program: Histogram Plot

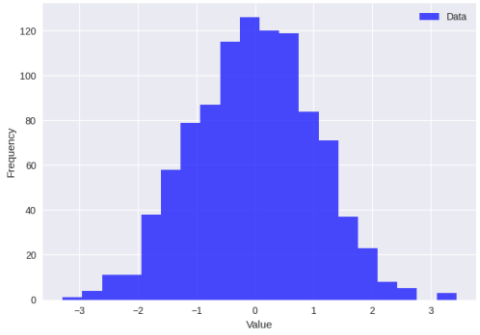
```
import matplotlib.pyplot as plt
import numpy as np

# Generate 1000 random numbers from a standard normal distribution (mean=0, std=1)

data = np.random.randn(1000)

# Plot histogram
plt.hist(
   x=data,        # Input data
   bins=20,     # Divide data into 20 intervals (bins)
   color='blue',    # Bar color
   alpha=0.7,                #Transparency (0=transparent, 1=opaque)
   label='Data'    # Label for legend
)

# Labels & legend
plt.xlabel('Value')     # X-axis label
plt.ylabel('Frequency')  # Y-axis label
plt.legend()         # Show legend ("Data")
plt.show()           # Display the plot
```

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

data = np.random.randn(1000)
plt.hist(x=data, bins=20, color='blue', alpha=0.7, label='Data')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

OUTPUT:



OUTPUT:

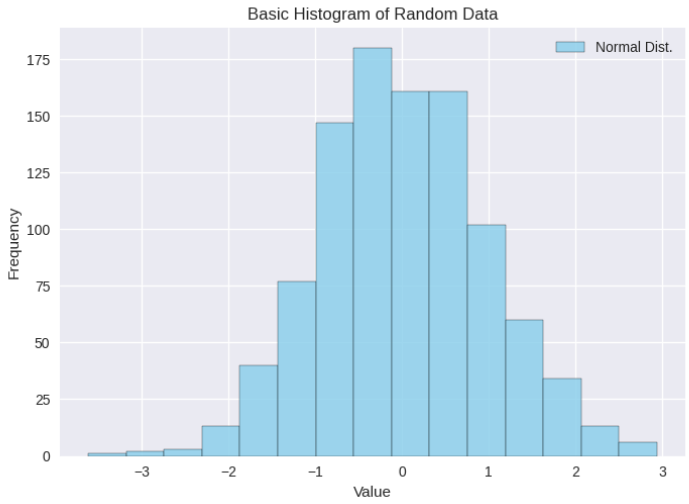


Example Program: Basic Histogram with Random Data

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

# Generate sample data (1000 random numbers from a normal distribution)
data = np.random.randn(1000)

# Create histogram
plt.hist(
   x=data,          # Input data
   bins=15,         # Number of bins
   color='skyblue',    # Bar color
   edgecolor='black',  # Bar edge color
```

```
    alpha=0.8,        # Transparency
    label='Normal Dist.' # Legend label
)

# Add labels and title
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Basic Histogram of Random Data')
plt.legend()

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

OUTPUT:



Explanation:

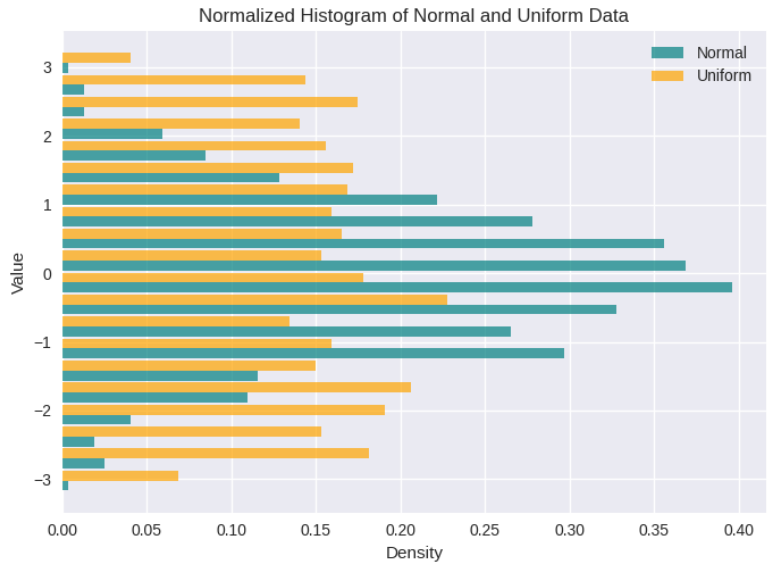
- **Data**: 1000 random numbers from a normal distribution (np.random.randn).
- **Parameters Used**:
  - ✓ x=data: The input data.
  - ✓ bins=15: Uses 15 bins for the histogram.
  - ✓ color='skyblue': Sets bar color.
  - ✓ edgecolor='black': Adds black outlines for clarity.
  - ✓ alpha=0.8: Slight transparency for visual effect.
  - ✓ label='Normal Dist.': Adds a legend label.
- **Output**: A histogram showing the frequency of values in 15 bins, with labeled axes and a legend.

Example Program: Normalized Histogram with Multiple Datasets

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

# Generate sample data
data1 = np.random.randn(1000)  # Normal distribution
data2 = np.random.uniform(-3, 3, 1000)  # Uniform distribution

# Create histogram
```

```
plt.hist(
    x=[data1, data2],       # Multiple datasets
    bins=20,              # Number of bins
    density=True,         # Normalize to probability density
    histtype='bar',       # Bar-style histogram
    color=['teal', 'orange'],  # Colors for each dataset
    label=['Normal', 'Uniform'],  # Legend labels
    orientation='horizontal',  # Horizontal bars
    alpha=0.7,            # Transparency
    rwidth=0.9            # Bar width with small gaps
)

# Add labels and title
plt.xlabel('Density')
plt.ylabel('Value')
plt.title('Normalized Histogram of Normal and Uniform Data')
plt.legend()

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

OUTPUT:



Explanation:

- **Data**: Two datasets (data1: normal distribution, data2: uniform distribution between -3 and 3).
- **Parameters Used**:
  - ✓ x=[data1, data2]: Plots two datasets in the same histogram.
  - ✓ bins=20: Uses 20 bins.
  - ✓ density=True: Normalizes the histogram to a probability density.
  - ✓ histtype='bar': Uses bar-style histogram.
  - ✓ color=['teal', 'orange']: Different colors for each dataset.
  - ✓ label=['Normal', 'Uniform']: Labels for the legend.
  - ✓ orientation='horizontal': Flips the histogram to horizontal bars.

- ✓ alpha=0.7: Adds transparency.
- ✓ rwidth=0.9: Creates small gaps between bars.

Output: A horizontal histogram comparing the normalized distributions of two datasets, with a legend distinguishing them.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *pie()*

The pie() function in matplotlib.pyplot is used to create pie charts, which are circular statistical graphics divided into slices to illustrate numerical proportions.

SYNTAX

```
matplotlib.pyplot.pie(
    x,
    explode=None,
    labels=None,
    colors=None,
    autopct=None,
    shadow=False,
    startangle=0,
    radius=1,
    counterclock=True,
    wedgeprops=None,
    textprops=None,
    center=(0, 0),
    frame=False,
    rotatelabels=False,
    **kwargs
)
```

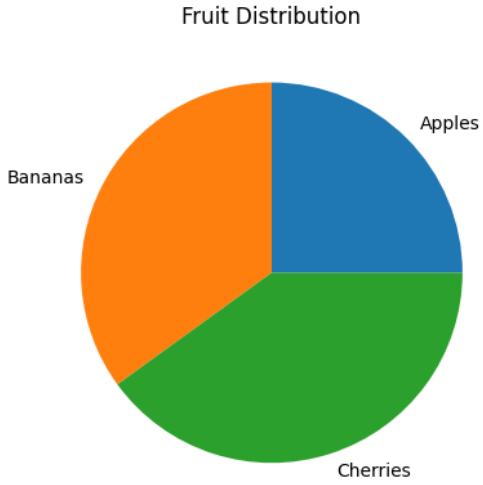
Explanation:

Parameter	Description
x	Array-like values representing the wedge sizes.
explode	List of offsets to "explode" slices outward (e.g., [0.1, 0, 0]).
labels	Labels for each slice.
colors	List of colors for the slices.
autopct	Format string or function to display values (e.g., '%1.1f%%').
shadow	If True, adds a shadow beneath the pie.
startangle	Angle (in degrees) to start the first slice.
radius	Radius of the pie chart (default is 1).
counterclock	If True, slices are drawn counterclockwise.
wedgeprops	Dictionary of properties for the wedges (e.g., {'edgecolor': 'black'}).
textprops	Dictionary of properties for the text labels (e.g., {'fontsize': 12}).
center	Tuple specifying the center of the pie chart.
frame	If True, draws a frame around the pie.
rotatelabels	If True, rotates labels to match slice angles.
**kwargs	Additional keyword arguments passed to matplotlib.patches.Patch.

Example Program: Basic Pie Chart

```
import matplotlib.pyplot as plt

# Data values (percentages of the whole)
sizes = [25, 35, 40]

# Labels for each slice
labels = ['Apples', 'Bananas', 'Cherries']

# Plot pie chart
plt.pie(sizes, labels=labels)

# Add a title
plt.title('Fruit Distribution')

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

OUTPUT:

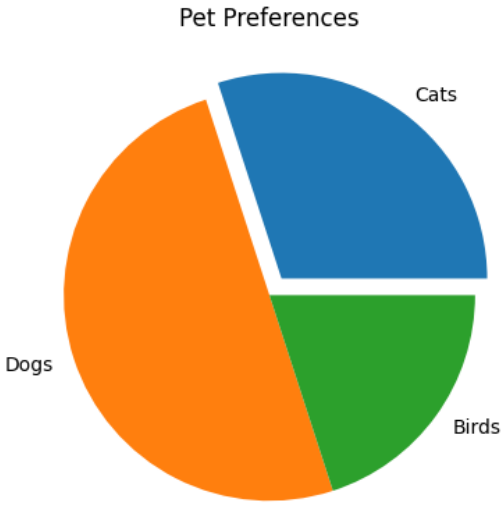


Example Program: Exploded Pie Chart

```
import matplotlib.pyplot as plt

# Data values
sizes = [30, 50, 20]

# Labels for each slice
labels = ['Cats', 'Dogs', 'Birds']

# Explode: highlight the first slice (Cats) by 0.1 offset
explode = [0.1, 0, 0]

# Plot pie chart
plt.pie(sizes, labels=labels, explode=explode)

# Add title
plt.title('Pet Preferences')
```

```
# Display chart
plt.show()
```

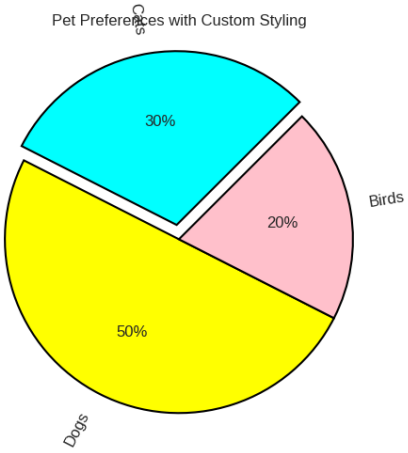
OUTPUT:



Example Program: Pie Chart with Custom Styling and Rotated Labels

```
import matplotlib.pyplot as plt

# Data
sizes = [30, 50, 20]
labels = ['Cats', 'Dogs', 'Birds']
explode = [0.1, 0, 0]  # Highlight Cats

# Create pie chart with customizations
plt.pie(
   x=sizes,               # Wedge sizes
   labels=labels,          # Wedge labels
   explode=explode,        # Explode Cats slice
   colors=['cyan', 'yellow', 'pink'],  # Custom colors
   autopct='%.0f%%',       # Show whole number percentages
   startangle=45,          # Start at 45 degrees
   wedgeprops={'edgecolor': 'black', 'linewidth': 1.5},  # Black edges
   textprops={'fontsize': 12},  # Larger label font
   rotatelabels=True,      # Rotate labels to align with wedges
   radius=1.2              # Larger pie chart
)

# Add title
plt.title('Pet Preferences with Custom Styling')

# Ensure circular shape
plt.axis('equal')

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

OUTPUT:



Explanation:

Additions:

- *colors=['cyan', 'yellow', 'pink']: Bright colors for each slice.*
- *autopct='%.0f%%': Shows percentages without decimals (e.g., 30%).*
- *startangle=45: Starts the chart at a 45-degree angle.*
- *wedgeprops={'edgecolor': 'black', 'linewidth': 1.5}: Adds black outlines to wedges.*
- *textprops={'fontsize': 12}: Increases label font size.*
- *rotatelabels=True: Rotates labels to align with each slice's angle.*
- *radius=1.2: Makes the pie chart slightly larger.*
- *plt.axis('equal'): Ensures circular shape.*

Output: A larger pie chart with the "Cats" slice exploded, black-edged wedges, rotated labels, whole-number percentages, and vibrant colors.

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *xlabel()*

The **xlabel()** function in Matplotlib is used to set the **label (name) of the x-axis** in a plot. This improves clarity by describing what values on the x-axis represent.

SYNTAX

```
matplotlib.pyplot.xlabel(
    xlabel,
    fontdict=None,
    labelpad=None,
    loc=None,
    **kwargs
)
```

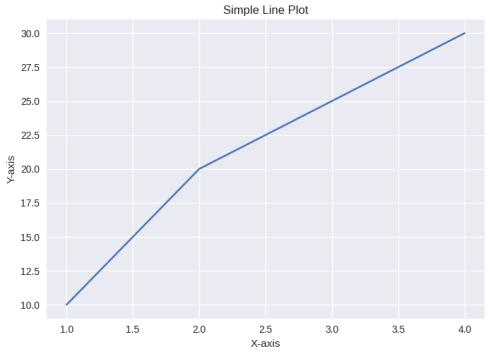
Explanation:

- **xlabel: *str*.** The x-axis label text (required).
- **fontdict:** *dict, optional*. Custom font properties, such as size, weight, color, etc.
- **labelpad:** *float, optional*. Padding (space) between label and axis.
- **loc:** *{'left', 'center', 'right'}, optional*. Alignment of label (default is 'center').
- **kwargs:** Additional text properties like *color*, *fontsize*, etc.

Example Program: Simple X-axis Label

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

plt.plot(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

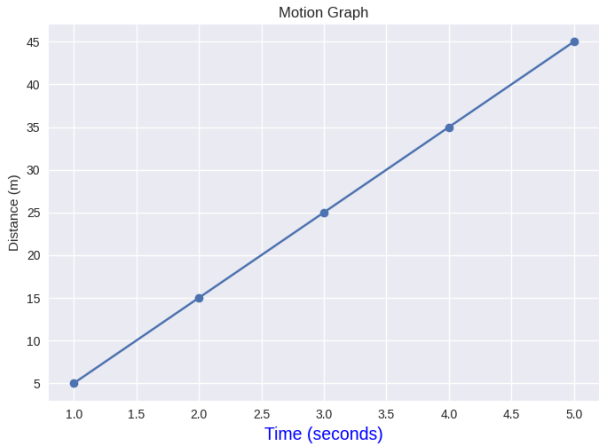
- x = [1, 2, 3, 4]: X-coordinates for the plot.
- y = [10, 20, 25, 30]: Y-coordinates for the plot.
  - ✓ **plt.plot(x, y)**: Creates a line plot connecting the points (1, 10), (2, 20), (3, 25), and (4, 30).
  - ✓ **plt.xlabel("X-axis")**: Sets the x-axis label to "X-axis" with default styling.
  - ✓ **plt.ylabel("Y-axis")**: Sets the y-axis label to "Y-axis".
  - ✓ **plt.title("Simple Line Plot")**: Sets the plot title.
  - ✓ **plt.show()**: Displays the plot.

Output: A line plot with a linear trend, labeled x-axis ("X-axis"), y-axis ("Y-axis"), and title ("Simple Line Plot"), using Matplotlib's default styling.

Example Program: Custom Font and Color

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [5, 15, 25, 35, 45]

plt.plot(x, y, marker='o')
plt.xlabel("Time (seconds)", fontdict={'fontsize':14, 'color':'blue'})
plt.ylabel("Distance (m)")
plt.title("Motion Graph")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

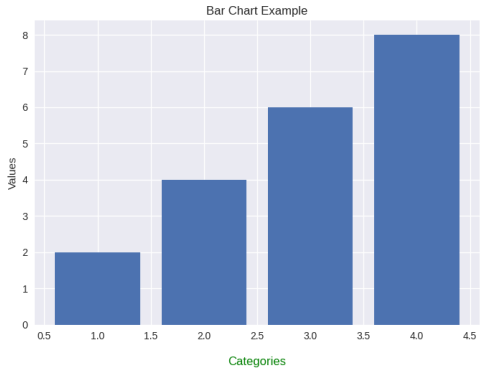
- x = [1, 2, 3, 4, 5]: X-coordinates (time in seconds).
- y = [5, 15, 25, 35, 45]: Y-coordinates (distance in meters).
  - ✓ **plt.plot(x, y, marker='o')**: Creates a line plot with circular markers at each point (1, 5), (2, 15), (3, 25), (4, 35), (5, 45).
  - ✓ **plt.xlabel("Time (seconds)", fontdict={'fontsize':14, 'color':'blue'})**:
- Sets the x-axis label to "Time (seconds)".
- fontdict={'fontsize':14, 'color':'blue'}: Uses a font size of 14 and blue color.
  - ✓ **plt.ylabel("Distance (m)")**: Sets the y-axis label to "Distance (m)".
  - ✓ **plt.title("Motion Graph")**: Sets the plot title.
  - ✓ **plt.show()**: Displays the plot.

Output: A line plot with circular markers showing a linear relationship between time and distance, with a blue x-axis label ("Time (seconds)"), y-axis label ("Distance (m)"), and title ("Motion Graph").

Example Program: Label with Padding and Rotation

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.bar(x, y)
plt.xlabel("Categories", fontsize=12, color='green', labelpad=15, rotation=0)
plt.ylabel("Values")
plt.title("Bar Chart Example")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

- x = [1, 2, 3, 4]: X-coordinates for the bars (categories).
- y = [2, 4, 6, 8]: Heights of the bars (values).
  - ✓ **plt.bar(x, y)**: Creates a bar chart with bars at x-positions 1, 2, 3, 4, with heights 2, 4, 6, 8.
  - ✓ **plt.xlabel("Categories", fontsize=12, color='green', labelpad=15, rotation=0)**:
- Sets the x-axis label to "Categories".
- fontsize=12: Sets font size to 12.
- color='green': Uses green text.
- labelpad=15: Adds 15 points of spacing from the x-axis.
- rotation=0: Ensures no rotation (default).
  - ✓ **plt.ylabel("Values")**: Sets the y-axis label to "Values".
  - ✓ **plt.title("Bar Chart Example")**: Sets the plot title.
  - ✓ **plt.show()**: Displays the bar chart.

Output: A bar chart with four bars of increasing height, a green x-axis label ("Categories") with size 12 and 15-point padding, a y-axis label ("Values"), and a title ("Bar Chart Example").

BUILT-IN FUNCTIONS OF matplotlib.pyplot: *xlabel(), ylabel(), title()*

SYNTAX: xlabel()

Sets the label (title) for the x-axis.

```
matplotlib.pyplot.xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)
```

Explanation:

- **xlabel** → Text for the x-axis.
- **fontdict** → Font styling (fontsize, weight, color).
- **labelpad** → Distance between label and axis.

SYNTAX: ylabel()

Sets the label (title) for the y-axis.

```
matplotlib.pyplot.ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)
```

Explanation:

Works the same way as xlabel(), but for the y-axis.

SYNTAX: title()

Sets the main title for the plot.

```
matplotlib.pyplot.title(label, fontdict=None, loc=None, pad=None, **kwargs)
```

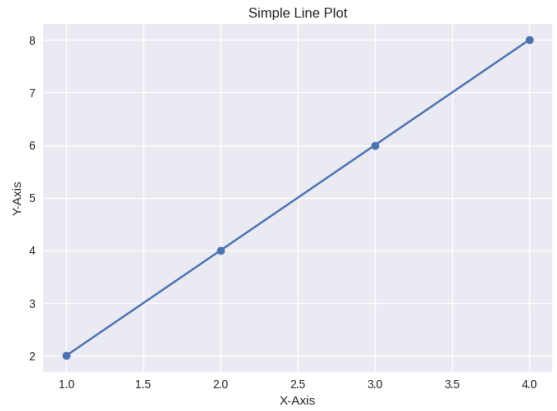
Explanation:

- **label** → Title text.
- **loc** → Position ('center' (default), 'left', 'right').
- **pad** → Spacing between title and plot.

Example Program: Line Plot with Labels and Title

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y, marker='o')
plt.xlabel("X-Axis")
plt.ylabel("Y-Axis")
plt.title("Simple Line Plot")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

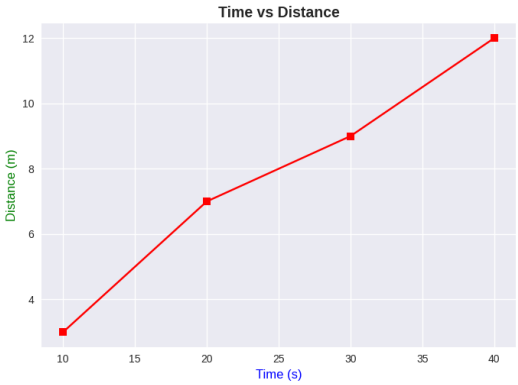
- **x = [1, 2, 3, 4]:** X-coordinates.
- **y = [2, 4, 6, 8]:** Y-coordinates.
  - ✓ **plt.plot(x, y, marker='o'):** Creates a line plot connecting points (1, 2), (2, 4), (3, 6), and (4, 8) with circular markers.
  - ✓ **plt.xlabel("X-Axis"):** Sets the x-axis label to "X-Axis" with default styling.
  - ✓ **plt.ylabel("Y-Axis"):** Sets the y-axis label to "Y-Axis" with default styling.
  - ✓ **plt.title("Simple Line Plot"):** Sets the plot title to "Simple Line Plot" with default styling.
  - ✓ **plt.show():** Displays the plot.

Output: A line plot with circular markers showing a linear trend, labeled x-axis ("X-Axis"), y-axis ("Y-Axis"), and title ("Simple Line Plot"), using Matplotlib's default styling.

Example Program: Custom Font and Color

```
import matplotlib.pyplot as plt

x = [10, 20, 30, 40]
y = [3, 7, 9, 12]

plt.plot(x, y, marker='s', color='red')
plt.xlabel("Time (s)", fontdict={'fontsize':12, 'color':'blue'})
plt.ylabel("Distance (m)", fontdict={'fontsize':12, 'color':'green'})
plt.title("Time vs Distance", fontdict={'fontsize':14, 'weight':'bold'}, loc='center')
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

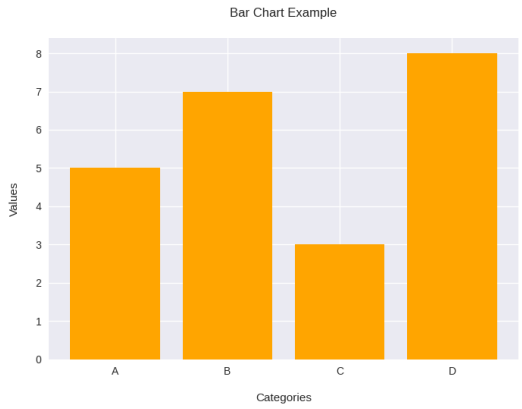
- x = [10, 20, 30, 40]: X-coordinates (time in seconds).
- y = [3, 7, 9, 12]: Y-coordinates (distance in meters).
- **plt.plot(x, y, marker='s', color='red')**: Creates a red line plot with square markers at points (10, 3), (20, 7), (30, 9), and (40, 12).
- **plt.xlabel("Time (s)", fontdict={'fontsize':12, 'color':'blue'})**:
  - ✓ Sets the x-axis label to "Time (s)".
  - ✓ fontdict={'fontsize':12, 'color':'blue'}: Size 12, blue color.
- **plt.ylabel("Distance (m)", fontdict={'fontsize':12, 'color':'green'})**:
  - ✓ Sets the y-axis label to "Distance (m)".
  - ✓ fontdict={'fontsize':12, 'color':'green'}: Size 12, green color.
- **plt.title("Time vs Distance", fontdict={'fontsize':14, 'weight':'bold'}, loc='center')**:
  - ✓ Sets the title to "Time vs Distance".
  - ✓ fontdict={'fontsize':14, 'weight':'bold'}: Size 14, bold.
  - ✓ loc='center': Centers the title (default).
- **plt.show()**: Displays the plot.

Output: A red line plot with square markers showing a trend between time and distance, with a blue x-axis label ("Time (s)"), green y-axis label ("Distance (m)"), and bold, centered title ("Time vs Distance").

Example Program: Bar Chart with Padding

```
import matplotlib.pyplot as plt

x = ['A', 'B', 'C', 'D']
y = [5, 7, 3, 8]

plt.bar(x, y, color='orange')
plt.xlabel("Categories", labelpad=15)
plt.ylabel("Values", labelpad=15)
plt.title("Bar Chart Example", pad=20)
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

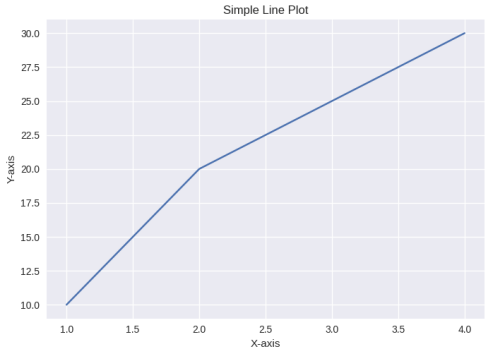
- x = ['A', 'B', 'C', 'D']: Categorical x-coordinates (labels for categories).
- y = [5, 7, 3, 8]: Heights of the bars (values).
- **plt.bar(x, y, color='orange')**: Creates a bar chart with orange bars at categories A, B, C, D, with heights 5, 7, 3, 8.
- **plt.xlabel("Categories", labelpad=15)**:
  - ✓ Sets the x-axis label to “Categories”.
  - ✓ labelpad=15: Adds 15 points of spacing from the x-axis.
- **plt.ylabel("Values", labelpad=15)**:
  - ✓ Sets the y-axis label to “Values”.
  - ✓ labelpad=15: Adds 15 points of spacing from the y-axis.
- **plt.title("Bar Chart Example", pad=20)**:
  - ✓ Sets the title to “Bar Chart Example”.
  - ✓ pad=20: Adds 20 points of spacing above the plot.
- **plt.show()**: Displays the bar chart.

Output: A bar chart with four orange bars, labeled x-axis (“Categories”) and y-axis (“Values”) with 15-point padding, and a title (“Bar Chart Example”) with 20-point padding, using Matplotlib’s default styling.

Example Program: Simple X-axis Label

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

plt.plot(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

- x = [1, 2, 3, 4]: X-coordinates for the plot.
- y = [10, 20, 25, 30]: Y-coordinates for the plot.
  - ✓ **plt.plot(x, y)**: Creates a line plot connecting the points (1, 10), (2, 20), (3, 25), and (4, 30).
  - ✓ **plt.xlabel("X-axis")**: Sets the x-axis label to "X-axis" with default styling.
  - ✓ **plt.ylabel("Y-axis")**: Sets the y-axis label to "Y-axis".
  - ✓ **plt.title("Simple Line Plot")**: Sets the plot title.
  - ✓ **plt.show()**: Displays the plot.

Output: A line plot with a linear trend, labeled x-axis ("X-axis"), y-axis ("Y-axis"), and title ("Simple Line Plot"), using Matplotlib's default styling.

SYNTAX: legend()

- The legend() function in Matplotlib is used to display a legend box that identifies the elements (lines, bars, etc.) in your plot.
- A legend identifies the plotted elements, clarifies their meaning, and makes complex graphs readable.

```
matplotlib.pyplot.legend(
    loc='best',
    labels=None,
    fontsize=None,
    title=None,
    shadow=False,
    frameon=True,
    ncol=1,
    bbox_to_anchor=None,
```

```
        **kwargs
)
```

Explanation:

loc → Position of legend ('best', 'upper left', 'upper right', 'lower left', 'lower right', 'center', etc.). Default = 'best'.

labels → Custom labels for the legend.

fontsize → Size of legend text.

title → Title for the legend box.

shadow → If True, draws shadow under legend.

frameon → If False, removes the box frame.

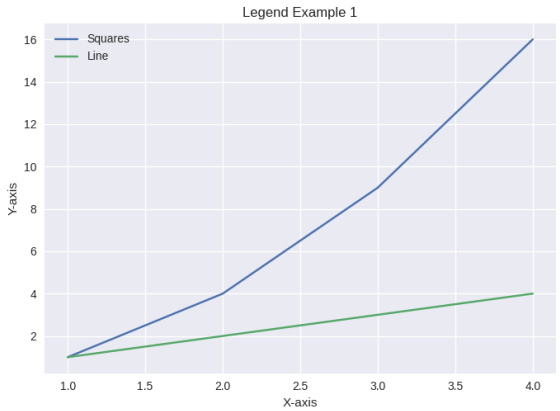
ncol → Number of columns in legend.

bbox_to_anchor → Places legend outside the plot ((x, y) coordinates).

Example Program: Simple Line Plot with Legend

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y1 = [1, 4, 9, 16]
y2 = [1, 2, 3, 4]

plt.plot(x, y1, label="Squares")
plt.plot(x, y2, label="Line")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Legend Example 1")
plt.legend()  # default position
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

- x = [1, 2, 3, 4]: X-coordinates.
- y1 = [1, 4, 9, 16]: Y-coordinates for the first dataset (squares of x).
- y2 = [1, 2, 3, 4]: Y-coordinates for the second dataset (linear).
- **plt.plot(x, y1, label="Squares")**: Plots a line for y1 with label “Squares”.
- **plt.plot(x, y2, label="Line")**: Plots a line for y2 with label “Line”.

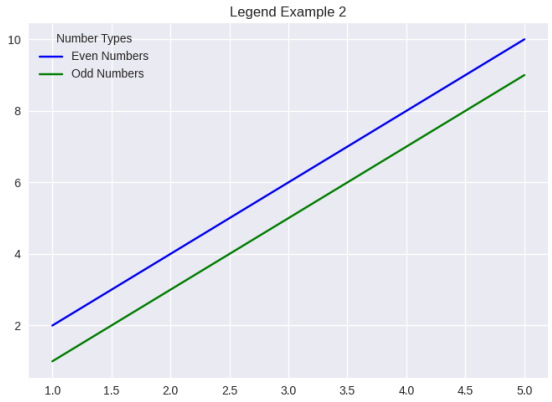
- **plt.xlabel("X-axis")**: Sets the x-axis label to "X-axis" with default styling.
- **plt.ylabel("Y-axis")**: Sets the y-axis label to "Y-axis" with default styling.
- **plt.title("Legend Example 1")**: Sets the title to "Legend Example 1" with default styling.
- **plt.legend()**: Adds a legend using default settings (loc='best', frameon=True, shadow=False, ncol=1), displaying "Squares" and "Line" based on plot labels.
- **plt.show()**: Displays the plot.

Output: A line plot with two lines (one for squares, one linear), labeled x-axis ("X-axis"), y-axis ("Y-axis"), title ("Legend Example 1"), and a legend automatically placed to minimize overlap, showing "Squares" and "Line".

Example Program: Legend with Custom Location & Title

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 3, 5, 7, 9]

plt.plot(x, y1, label="Even Numbers", color='blue')
plt.plot(x, y2, label="Odd Numbers", color='green')
plt.title("Legend Example 2")
plt.legend(loc='upper left', title="Number Types", fontsize=10)
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

- x = [1, 2, 3, 4, 5]: X-coordinates.
- y1 = [2, 4, 6, 8, 10]: Y-coordinates for the first dataset (even numbers).
- y2 = [1, 3, 5, 7, 9]: Y-coordinates for the second dataset (odd numbers).

- **plt.plot(x, y1, label="Even Numbers", color='blue')**: Plots a blue line for y1 with label "Even Numbers".
- **plt.plot(x, y2, label="Odd Numbers", color='green')**: Plots a green line for y2 with label "Odd Numbers".

- **plt.title("Legend Example 2")**: Sets the title to “Legend Example 2” with default styling.
- **plt.legend(loc='upper left', title="Number Types", fontsize=10)**:
  - ✓ Adds a legend in the upper left corner.
  - ✓ title="Number Types": Sets the legend title.
  - ✓ fontsize=10: Sets legend text size to 10.
  - ✓ Uses default settings for other parameters (frameon=True, shadow=False, ncol=1).
- **plt.show()**: Displays the plot.

Output: A line plot with two lines (blue for even numbers, green for odd numbers), a title (“Legend Example 2”), and a legend in the upper left with title “Number Types” and labels “Even sNumbers” and “Odd Numbers”. No axis labels are included.

SYNTAX: grid()

The grid() function in Matplotlib is used to add grid lines to your plot for better readability of values.

```
matplotlib.pyplot.grid(
   b=None,
   which='major',
   axis='both',
   color=None,
   linestyle=None,
   linewidth=None,
   **kwargs
  )
```

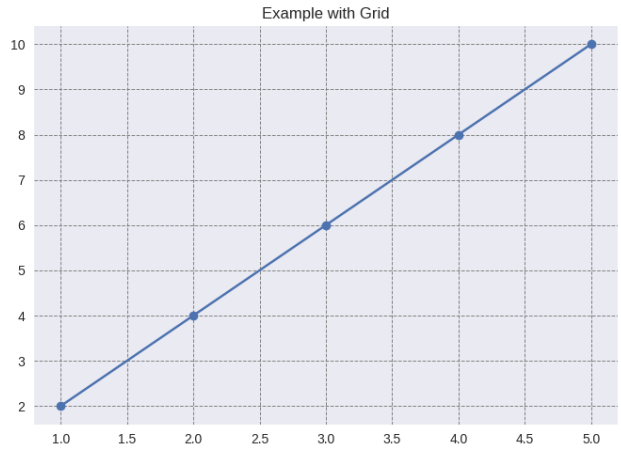
Explanation:

- **b** → Boolean (True / False) to turn grid on or off.
- **which** → 'major' (default), 'minor', or 'both' → controls which ticks have grid lines.
- **axis** → 'both' (default), 'x', or 'y' → controls where the grid appears.
- **color** → Color of grid lines (e.g., 'grey', 'red').
- **linestyle** → Style of grid ('-' solid, '--' dashed, ':' dotted, '-.' dash-dot).
- **linewidth** → Thickness of grid lines.
- **kwargs** → Extra customization.

Example Program: Simple Line Plot with Legend

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y, marker='o')
plt.title("Example with Grid")

# Enable grid lines
plt.grid(True, which='both', axis='both', color='gray', linestyle='--', linewidth=0.7)

plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

x = [1, 2, 3, 4, 5]: X-coordinates.
y = [2, 4, 6, 8, 10]: Y-coordinates (linear relationship, y = 2x).

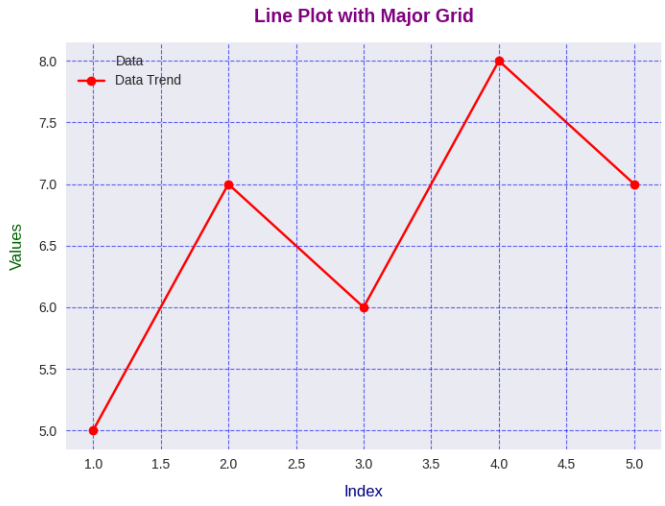
- **plt.plot(x, y, marker='o'):** Creates a line plot with circular markers at points (1, 2), (2, 4), (3, 6), (4, 8), (5, 10).
- **plt.title("Example with Grid"):** Sets the title to "Example with Grid" with default styling.
- **plt.grid**(True, which='both', axis='both', color='gray', linestyle='--', linewidth=0.7): Enables grid lines (True).
  which='both': Includes both major and minor grid lines.
  axis='both': Applies grid lines to both x- and y-axes.
  color='gray': Sets grid lines to gray.
  linestyle='--': Uses dashed lines.
  linewidth=0.7: Sets line thickness to 0.7 points.
- **plt.show():** Displays the plot.

Output: A line plot with circular markers, a title ("Example with Grid"), and a grid of dashed gray lines (major and minor) on both axes. No axis labels or legend are included.

Example Program: Line Plot with Styled Major Grid and Legend

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [5, 7, 6, 8, 7]

# Create line plot with label
plt.plot(x, y, marker='o', color='red', label='Data Trend')

# Set labels and title
plt.xlabel('Index', fontdict={'fontsize': 12, 'color': 'navy'}, labelpad=10)
plt.ylabel('Values', fontdict={'fontsize': 12, 'color': 'darkgreen'}, labelpad=10)
plt.title('Line Plot with Major Grid', fontdict={'fontsize': 14, 'fontweight': 'bold',
'color': 'purple'}, pad=15)

# Enable grid with customizations
plt.grid(True, which='major', axis='both', color='blue', linestyle='--', linewidth=0.8,
alpha=0.6)

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

# Display plot
plt.show()
```

OUTPUT:



Explanation:

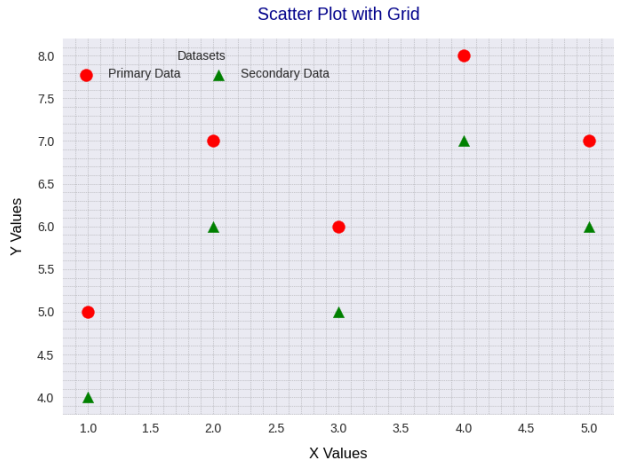
- **Plot:** A red line plot with circular markers.
- **xlabel:** "Index", size 12, navy, 10-point padding.
- **ylabel:** "Values", size 12, dark green, 10-point padding.
- **title:** "Line Plot with Major Grid", bold, size 14, purple, 15-point padding.
- **grid:** Major grid lines on both axes, blue, dashed, 0.8-point width, 60% opacity.
- **legend:** Upper left, size 10, with title "Data".

Output: A line plot with a blue dashed major grid, styled axis labels, a bold title, and a legend.

Example Program: Scatter Plot with Major and Minor Grid

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [5, 7, 6, 8, 7]
y2 = [4, 6, 5, 7, 6]  # Second dataset for comparison

# Create scatter plots with labels
plt.scatter(x, y, color='red', s=100, marker='o', label='Primary Data')
plt.scatter(x, y2, color='green', s=80, marker='^', label='Secondary Data')

# Set labels and title
plt.xlabel('X Values', fontdict={'fontsize': 12, 'color': 'black'}, labelpad=10)
plt.ylabel('Y Values', fontdict={'fontsize': 12, 'color': 'black'}, labelpad=10)
plt.title('Scatter Plot with Grid', fontdict={'fontsize': 14, 'color': 'darkblue'},
pad=15)

# Enable minor ticks
plt.minorticks_on()

# Enable grid with customizations
plt.grid(True, which='both', axis='both', color='gray', linestyle=':', linewidth=0.5,
alpha=0.7)

# Add legend
plt.legend(loc='best', fontsize=10, title='Datasets', ncol=2, framealpha=0.8)

# Display plot
plt.show()
```

OUTPUT:



Explanation:

- **Plot:** Two scatter plots (red circles for y, green triangles for y2).
- **xlabel:** "X Values", size 12, black, 10-point padding.
- **ylabel:** "Y Values", size 12, black, 10-point padding.
- **title:** "Scatter Plot with Grid", size 14, dark blue, 15-point padding.

- **grid:** Major and minor grid lines (which='both'), gray, dotted, 0.5-point width, 70% opacity. Minor ticks enabled with plt.minorticks_on().
- **legend:** Automatically placed (loc='best'), size 10, title "Datasets", two columns, semi-transparent frame.
- **Output:** A scatter plot with major and minor gray dotted grids, axis labels, and a two-column legend.

SYNTAX: *xlim() and ylim()*

These functions are used to set or get the limits of the x-axis and y-axis in a plot.

```
plt.xlim([xmin, xmax])
plt.ylim([ymin, ymax])

        or
plt.xlim(xmin, xmax)
plt.ylim(ymin, ymax)
```

Note: Both forms are valid.

Explanation:

xlim(): Sets the limits of the x-axis.

- **xmin:** Minimum value to display on the x-axis.
- **xmax:** Maximum value to display on the x-axis.

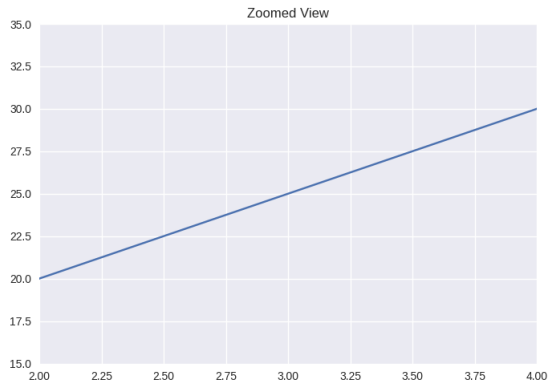
ylim(): Sets the limits of the y-axis.

- **ymin:** Minimum value to display on the y-axis.
- **ymax:** Maximum value to display on the y-axis.

Example Program: Simple Line Plot with Legend

```
import matplotlib.pyplot as plt

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

plt.plot(x, y)
plt.xlim(2, 4)
plt.ylim(15, 35)
plt.title("Zoomed View")
plt.show()
```

OUTPUT:



Explanation:

Import: Imports matplotlib.pyplot for plotting.

Data:

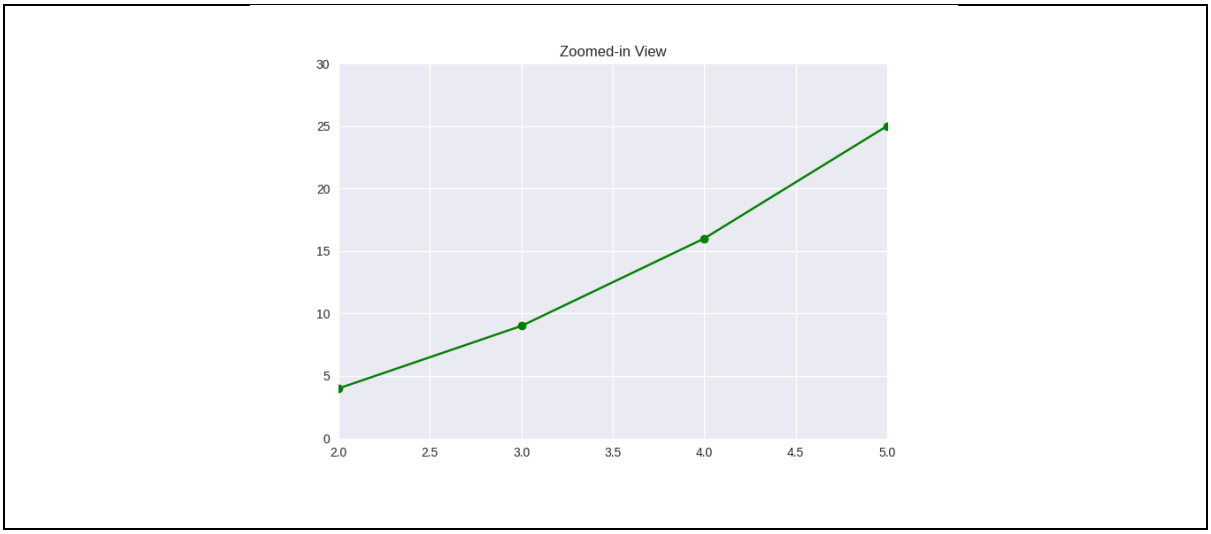
- **x = [1, 2, 3, 4, 5]:** X-coordinates.
- **y = [10, 20, 25, 30, 40]:** Y-coordinates.
- **plt.plot(x, y):** Creates a line plot connecting points (1, 10), (2, 20), (3, 25), (4, 30), (5, 40) with default styling (blue line, no markers).
- **plt.xlim(2, 4):** Sets the x-axis range from 2 to 4, zooming in on a subset of the data.
- **plt.ylim(15, 35):** Sets the y-axis range from 15 to 35, focusing on the corresponding y-values.
- **plt.title("Zoomed View"):** Sets the title to "Zoomed View" with default styling.
- **plt.show():** Displays the plot.

Output: A line plot showing only the portion of the data between x=2 and x=4 (points (2, 20), (3, 25), (4, 30)), with y-values between 15 and 35. No axis labels, legend, or grid are included.

Example Program: Zooming into Part of the Graph

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5, 6]
y = [0, 1, 4, 9, 16, 25, 36]

plt.plot(x, y, marker='o', color='green')
plt.title("Zoomed-in View")
plt.xlim(2, 5)
plt.ylim(0, 30)
plt.show()
```

OUTPUT:



Explanation:
Import: Imports matplotlib.pyplot for plotting.
Data:

- **x = [0, 1, 2, 3, 4, 5, 6]:** X-coordinates.
- **y = [0, 1, 4, 9, 16, 25, 36]:** Y-coordinates (quadratic relationship, $y = x^2$).

plt.plot(x, y, marker='o', color='green'): Creates a green line plot with circular markers at points (0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36).
plt.title("Zoomed-in View"): Sets the title to "Zoomed-in View" with default styling.
plt.xlim(2, 5): Sets the x-axis range from 2 to 5, zooming in to show points (2, 4), (3, 9), (4, 16), (5, 25).
plt.ylim(0, 30): Sets the y-axis range from 0 to 30, focusing on y-values up to 30.
plt.show(): Displays the plot.
Output: A green line plot with circular markers, showing only the portion of the data between x=2 and x=5 (points (2, 4), (3, 9), (4, 16), (5, 25)), with y-values between 0 and 30. No axis labels, legend, or grid are included.

5.5 NumPy Built-in functions

NumPy is a fundamental Python library for numerical operations (computations), offering a wide range of built-in functions for array creation, manipulation, mathematical operations, and statistical analysis.

The different NumPy type built-in functions are:

- Array Creation
- Array Inspection
- Math Functions
- Elementwise Operations
- Linear Algebra
- Random
- Reshape & Manipulation
- Sorting & Searching

5.5.1 Array Creation built-in functions of packages NumPy

Array creation functions in NumPy are built-in methods that allow users to initialize arrays of various shapes, sizes, and values. These functions are foundational for numerical computing, enabling efficient storage, manipulation, and processing of data in structured formats.
They support:

- Manual input (e.g., from lists or tuples)
- Automated generation (e.g., ranges, random values, identity matrices)
- Custom initialization (e.g., zeros, ones, specific values)

Table: Summary of Array Creation built-in functions of packages NumPy

Method	Syntax	Example	Explanation
array()	np.array(object, dtype=None)	np.array([1, 2, 3]) → [1 2 3]	Converts a list or tuple into a NumPy array.
zeros()	np.zeros(shape, dtype=float)	np.zeros((2, 3)) → [[0. 0. 0.],[0. 0. 0.]]	Creates an array filled with zeros. Shape is (rows, columns).
ones()	np.ones(shape, dtype=float)	np.ones((3, 2)) → [[1. 1.],[1. 1.],[1. 1.]]	Creates an array filled with ones.
empty()	np.empty(shape, dtype=float)	np.empty((2, 2)) → [[6.9e-310 6.9e-310],[0.0 0.0]] *(values may vary)*	Creates an array without initializing entries (may contain garbage values).
full()	np.full(shape, fill_value)	np.full((2, 2), 7) → [[7 7],[7 7]]	Creates an array filled with a specified value.
arange()	np.arange(start, stop, step)	np.arange(0, 10, 2) → [0 2 4 6 8]	Creates array with evenly spaced values within a range.
linspace()	np.linspace(start, stop, num)	np.linspace(0, 1, 5) → [0. 0.25 0.5 0.75 1.]	Creates array with evenly spaced values between two points.
eye()	np.eye(N, M=None, k=0)	np.eye(3) → [[1. 0. 0.],[0. 1. 0.],[0. 0. 1.]]	Creates a 2D identity matrix.
random.rand()	np.random.rand(d0, d1, ..., dn)	np.random.rand(2, 2) → [[0.12 0.73],[0.55 0.91]] *(values vary)*	Creates array with random values from a uniform distribution over [0, 1).
random.randint()	np.random.randint(low, high, size)	np.random.randint(1, 10, size=(2, 3)) → [[3 7 2],[9 1 5]] *(values vary)*	Creates array with random integers in a specified range.

Example Program: Array Creation in NumPy

```
import numpy as np

# 1. array()
print("1. array()")
arr = np.array([1, 2, 3])
print(arr)   # → [1 2 3]
print()

# 2. zeros()
print("2. zeros()")
z = np.zeros((2, 3))
print(z)   # → [[0. 0. 0.], [0. 0. 0.]]
print()

# 3. ones()
print("3. ones()")
o = np.ones((3, 2))
print(o)   # → [[1. 1.], [1. 1.], [1. 1.]]
print()
```

```
# 4. empty()
print("4. empty()")
e = np.empty((2, 2))
print(e)   # values may vary (garbage values)
print()

# 5. full()
print("5. full()")
f = np.full((2, 2), 7)
print(f)   # → [[7 7], [7 7]]
print()

# 6. arange()
print("6. arange()")
a = np.arange(0, 10, 2)
print(a)   # → [0 2 4 6 8]
print()

# 7. linspace()
print("7. linspace()")
l = np.linspace(0, 1, 5)
print(l)   # → [0.   0.25 0.5  0.75 1. ]
print()

# 8. eye()
print("8. eye()")
I = np.eye(3)
print(I)   # → [[1. 0. 0.], [0. 1. 0.], [0. 0. 1.]]
print()

# 9. random.rand()
print("9. random.rand()")
r = np.random.rand(2, 2)
print(r)   # values vary (uniform [0,1))
print()

# 10. random.randint()
print("10. random.randint()")
ri = np.random.randint(1, 10, size=(2, 3))
print(ri)  # values vary between 1 and 9
```

OUTPUT: ***(values of random functions will vary)***

```
1. array()
[1 2 3]

2. zeros()
[[0. 0. 0.]
 [0. 0. 0.]]

3. ones()
[[1. 1.]
 [1. 1.]
 [1. 1.]]

4. empty()
[[6.94333818e-310 6.94333818e-310]
```

```
 [0.00000000e+000 0.00000000e+000]]
   (values will differ each time)

5. full()
[[7 7]
 [7 7]]

6. arange()
[0 2 4 6 8]

7. linspace()
[0.   0.25 0.5  0.75 1.  ]

8. eye()
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

9. random.rand()
[[0.312 0.789]
 [0.452 0.987]]
   (values will differ each time)

10. random.randint()
[[8 3 5]
 [2 7 9]]
   (values will differ each time)
```

Example Program: Array Creation in NumPy

```
import numpy as np

# 1. array()
arr = np.array([1, 2, 3])
print("array():", arr)

# 2. zeros()
z = np.zeros((2, 3))
print("\nzeros():\n", z)

# 3. ones()
o = np.ones((3, 2))
print("\nones():\n", o)

# 4. empty()
e = np.empty((2, 2))
print("\nempty():\n", e)

# 5. full()
f = np.full((2, 2), 7)
print("\nfull():\n", f)

# 6. arange()
a = np.arange(0, 10, 2)
print("\narange():", a)

# 7. linspace()
```

```
l = np.linspace(0, 1, 5)
print("\nlinspace():", l)

# 8. eye()
I = np.eye(3)
print("\neye():\n", I)

# 9. random.rand()
r = np.random.rand(2, 2)
print("\nrandom.rand():\n", r)

# 10. random.randint()
ri = np.random.randint(1, 10, size=(2, 3))
print("\nrandom.randint():\n", ri)
```

Output: ***(values of random functions will vary)***

```
array(): [1 2 3]

zeros():
 [[0. 0. 0.]
  [0. 0. 0.]]

ones():
 [[1. 1.]
  [1. 1.]
  [1. 1.]]

empty():
 [[0.00000000e+000 0.00000000e+000]
  [4.65661320e-310 6.93211133e-310]]   # (garbage values may change)

full():
 [[7 7]
  [7 7]]

arange(): [0 2 4 6 8]

linspace(): [0.   0.25 0.5  0.75 1.  ]

eye():
 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

random.rand():
 [[0.45 0.67]
  [0.21 0.89]]

random.randint():
 [[3 7 2]
  [9 1 5]]
```

5.5.2 Array Inspection built-in functions of packages NumPy

Array inspection functions help you **look inside** a NumPy array to understand its structure, size, shape, and data type. Think of them as tools to "ask questions" about the array like how big it is, what kind of data it holds, and how it's organized.

Table: Summary of Array Inspection built-in functions of packages NumPy

Method	Syntax	Example	Output	Explanation
array.shape	a.shape	a = np.array([[1,2],[3,4],[5,6]]); a.shape	(3, 2)	Shows the dimensions of the array (e.g., 3 rows, 2 columns).
array.ndim	a.ndim	a.ndim	2	Returns the number of dimensions (e.g., 2 for a 2D array).
array.size	a.size	a.size	6	Counts the total number of elements (e.g., 3 rows × 2 columns = 6).
array.dtype	a.dtype	a.dtype	dtype('int32') (system-dependent)	Shows the data type of elements (e.g., int32 for integers).
array.itemsize	a.itemsize	a.itemsize	4 (for int32)	Size in bytes of each element (e.g., 4 bytes for int32).
array.nbytes	a.nbytes	a.nbytes	24	Total memory used (e.g., 6 elements × 4 bytes = 24 bytes).
array.T	a.T	a.T	[[1, 3, 5], [2, 4, 6]]	Transposes the array (swaps rows and columns).
np.info(obj)	np.info(a)	np.info(a)	(Detailed metadata printed)	Shows full details like shape, dtype, strides, and memory layout.
np.isnan(a)	np.isnan(a)	np.isnan([1, np.nan])	[False, True]	Checks if elements are NaN (Not a Number).
np.isinf(a)	np.isinf(a)	np.isinf([1, np.inf])	[False, True]	Checks if elements are infinite (e.g., inf or -inf).

Note:

- The examples assume import numpy as np and a = np.array([[1, 2], [3, 4], [5, 6]]) unless specified otherwise.
- Outputs for dtype, itemsize, and nbytes may vary depending on the system (e.g., int32 vs. int64).
- np.info(a) prints a detailed report to the console, not a return value, so the output isn't a simple array.

Example Program: Integer 2D Array

```
import numpy as np

# Example 1: 2D integer array
a = np.array([[1, 2], [3, 4], [5, 6]])

print("Array:\n", a)

print("Shape:", a.shape)          # (3, 2)
print("Dimensions:", a.ndim)      # 2
print("Size:", a.size)            # 6
print("Data Type:", a.dtype)      # int32 (may vary)
```

```
print("Item Size:", a.itemsize)    # 4 (bytes per element)
print("Total Bytes:", a.nbytes)    # 24 (6 elements × 4 bytes)
print("Transpose:\n", a.T)         # Transposed array

print("\nCheck for NaN:", np.isnan(a))   # All False
print("Check for Inf:", np.isinf(a))     # All False

print("\nArray Info:")
np.info(a)
```

OUTPUT:

```
Array:
 [[1 2]
 [3 4]
 [5 6]]
Shape: (3, 2)
Dimensions: 2
Size: 6
Data Type: int32
Item Size: 4
Total Bytes: 24
Transpose:
 [[1 3 5]
 [2 4 6]]

Check for NaN:
 [[False False]
 [False False]
 [False False]]
Check for Inf:
 [[False False]
 [False False]
 [False False]]

Array Info:
class:  ndarray
shape:  (3, 2)
strides:  (8, 4)
itemsize: 4
aligned:  True
contiguous:  True
dtype: int32
```

Example Program: Covers All Array Inspection Functions

```
import numpy as np

# Create a 1D and 2D array
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([[1.5, 2.5], [3.5, 4.5], [5.5, 6.5]])

print("Array 1:\n", arr1)
print("Array 2:\n", arr2)

# Inspection on arr1
print("\n--- Inspection of arr1 ---")
print("Number of dimensions:", arr1.ndim)
print("Shape:", arr1.shape)
```

```
print("Size:", arr1.size)
print("Data type:", arr1.dtype)
print("Item size:", arr1.itemsize)
print("Total bytes used:", arr1.nbytes)
print("Transpose:\n", arr1.T)

# Inspection on arr2
print("\n--- Inspection of arr2 ---")
print("Number of dimensions:", arr2.ndim)
print("Shape:", arr2.shape)
print("Size:", arr2.size)
print("Data type:", arr2.dtype)
print("Item size:", arr2.itemsize)
print("Total bytes used:", arr2.nbytes)
print("Transpose:\n", arr2.T)
```

OUTPUT:

```
Array 1:
 [10 20 30 40 50]
Array 2:
 [[1.5 2.5]
  [3.5 4.5]
  [5.5 6.5]]

--- Inspection of arr1 ---
Number of dimensions: 1
Shape: (5,)
Size: 5
Data type: int32
Item size: 4
Total bytes used: 20
Transpose:
 [10 20 30 40 50]

--- Inspection of arr2 ---
Number of dimensions: 2
Shape: (3, 2)
Size: 6
Data type: float64
Item size: 8
Total bytes used: 48
Transpose:
 [[1.5 3.5 5.5]
  [2.5 4.5 6.5]]
```

Example Program: Array with NaN and Inf

```
import numpy as np

# Create array with NaN and Inf values
b = np.array([1.0, np.nan, np.inf, -np.inf])

print("Array b:", b)

print("Shape:", b.shape)       # (4,)
print("Dimensions:", b.ndim)   # 1
print("Size:", b.size)         # 4
```

```
print("Data Type:", b.dtype)     # float64
print("Item Size:", b.itemsize)  # 8 (for float64)
print("Total Bytes:", b.nbytes)  # 32 (4 elements × 8 bytes)
print("Transpose:", b.T)        # same as b (1D array has no change)

print("\nCheck for NaN:", np.isnan(b))   # [False True False False]
print("Check for Inf:", np.isinf(b))     # [False False  True  True]

print("\nArray Info:")
np.info(b)
```

OUTPUT:

```
Array b: [  1.  nan  inf -inf]
Shape: (4,)
Dimensions: 1
Size: 4
Data Type: float64
Item Size: 8
Total Bytes: 32
Transpose: [  1.  nan  inf -inf]

Check for NaN: [False  True False False]
Check for Inf: [False False  True  True]

Array Info:
class:  ndarray
shape:  (4,)
strides:  (8,)
itemsize: 8
aligned:  True
contiguous:  True
dtype: float64
```

Example Program:

```
import numpy as np

# Create a sample 2D array
a = np.array([[1, 2], [3, 4], [5, 6]])

# 1. array.shape: Shows the dimensions (rows, columns)
print("Shape:", a.shape)

# 2. array.ndim: Number of dimensions
print("Number of dimensions:", a.ndim)

# 3. array.size: Total number of elements
print("Size:", a.size)

# 4. array.dtype: Data type of elements
print("Data type:", a.dtype)

# 5. array.itemsize: Size of each element in bytes
print("Item size (bytes):", a.itemsize)

# 6. array.nbytes: Total memory used in bytes
print("Total bytes:", a.nbytes)
```

```
# 7. array.T: Transpose (swap rows and columns)
print("Transpose:\n", a.T)

# 8. np.info(obj): Detailed metadata about the array
print("Array info:")
np.info(a)

# 9. np.isnan(a): Check for NaN values
# Create a new array with a NaN value for this example
b = np.array([1, np.nan, 3])
print("Is NaN:", np.isnan(b))

# 10. np.isinf(a): Check for infinite values
# Create a new array with an infinite value for this example
c = np.array([1, np.inf, 3])
print("Is infinite:", np.isinf(c))
```

OUTPUT:

```
Shape: (3, 2)
Number of dimensions: 2
Size: 6
Data type: int64
Item size (bytes): 8
Total bytes: 48
Transpose:
 [[1 3 5]
 [2 4 6]]
Array info:
class:  ndarray
shape:  (3, 2)
strides:  (16, 8)
itemsize:  8
aligned:  True
contiguous:  True
fortran:  False
data pointer: 0x1730de0
byteorder:  little
byteswap:  False
type: int64
Is NaN: [False  True False]
Is infinite: [False  True False]
```

5.5.3 Math Functions of NumPy

Math functions in NumPy are built-in tools that let you perform **mathematical operations directly on arrays** quickly and efficiently. Instead of looping through each element like in regular Python, NumPy applies the operation to the entire array at once.

Table: Summary of Math Functions built-in functions of packages NumPy

Type	Syntax	Example	Output	Explanation
Arithmetic Functions	np.add(x, y)	np.add([1,2,3], [4,5,6])	[5 7 9]	Adds elements of two arrays
	np.subtract(x, y)	np.subtract([10,20,30], [1,2,3])	[9 18 27]	Subtracts second array from first
	np.multiply(x, y)	np.multiply([2,3,4], [5,6,7])	[10 18 28]	Multiplies element-wise
	np.divide(x, y)	np.divide([10,20,30], [2,5,10])	[5. 4. 3.]	Divides element-wise
	np.power(x, y)	np.power([2,3,4], 2)	[4 9 16]	Raises elements to a power
	np.mod(x, y)	np.mod([10,20,30], 7)	[3 6 2]	Remainder after division
Trigonometric Functions	np.sin(x)	np.sin(np.deg2rad([0,30,90]))	[0. 0.5 1.]	Computes sine values
	np.cos(x)	np.cos(np.deg2rad([0,60,90]))	[1. 0.5 0.]	Computes cosine values
	np.tan(x)	np.tan(np.deg2rad([0,45]))	[0. 1.]	Computes tangent values
	np.arcsin(x)	np.arcsin([0,1])	[0. 1.5708]	Inverse sine (radians)
	np.deg2rad(x)	np.deg2rad([180])	[3.1416]	Converts degrees → radians
	np.rad2deg(x)	np.rad2deg([np.pi])	[180.]	Converts radians → degrees
Exponential & Log Functions	np.exp(x)	np.exp([1,2])	[2.71828183 7.3890561]	Computes exponential (e^x)
	np.log(x)	np.log([1, np.e, np.e**2])	[0. 1. 2.]	Natural log (base e)
	np.log10(x)	np.log10([1,10,100])	[0. 1. 2.]	Log base 10
	np.log2(x)	np.log2([1,2,4,8])	[0. 1. 2. 3.]	Log base 2
Rounding Functions	np.around(x, n)	np.around([1.234, 2.567], 2)	[1.23 2.57]	Rounds to given decimals
	np.floor(x)	np.floor([1.2, 2.9, -3.7])	[1. 2. -4.]	Rounds down to nearest integer
	np.ceil(x)	np.ceil([1.2, 2.9, -3.7])	[2. 3. -3.]	Rounds up to nearest integer

NOTE:

Arithmetic : add, subtract, multiply, divide, power, mod
Trigonometric : sin, cos, tan, arcsin, rad2deg, deg2rad
Exponential & Log : exp, log, $\log_{10}$, $\log_2$
Rounding : around, floor, ceil

Example Program: Arithmetic and Trigonometric Functions

```
import numpy as np

# Arithmetic operations
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
```

```
print("Addition:", np.add(a1, a2))         # [5 7 9]
print("Subtraction:", np.subtract([10, 20, 30], [1, 2, 3]))  # [9 18 27]
print("Multiplication:", np.multiply([2, 3, 4], [5, 6, 7]))  # [10 18 28]
print("Division:", np.divide([10, 20, 30], [2, 5, 10]))      # [5. 4. 3.]
print("Power:", np.power([2, 3, 4], 2))      # [4 9 16]
print("Modulus:", np.mod([10, 20, 30], 7))    # [3 6 2]

# Trigonometric operations
angles_deg = np.array([0, 30, 90])
angles_rad = np.deg2rad(angles_deg)
print("Sine:", np.sin(angles_rad))         # [0. 0.5 1.]
print("Cosine:", np.cos(np.deg2rad([0, 60, 90])))  # [1. 0.5 0.]
print("Tangent:", np.tan(np.deg2rad([0, 45])))     # [0. 1.]
print("Arcsin:", np.arcsin([0, 1]))         # [0. 1.5708]
print("Degrees to Radians:", np.deg2rad([180]))    # [3.1416]
print("Radians to Degrees:", np.rad2deg([np.pi]))  # [180.]
```

OUTPUT:

```
Addition: [5 7 9]
Subtraction: [ 9 18 27]
Multiplication: [10 18 28]
Division: [5. 4. 3.]
Power: [ 4  9 16]
Modulus: [3 6 2]
Sine: [0.  0.5 1. ]
Cosine: [1.  0.5 0. ]
Tangent: [0. 1.]
Arcsin: [0.     1.5708]
Degrees to Radians: [3.1416]
Radians to Degrees: [180.]
```

Example Program: Exponential, Logarithmic, and Rounding Functions

```
import numpy as np

# Exponential and logarithmic functions
print("Exponential:", np.exp([1, 2]))           # [2.71828183 7.3890561]
print("Natural Log:", np.log([1, np.e, np.e**2]))   # [0. 1. 2.]
print("Log base 10:", np.log10([1, 10, 100]))      # [0. 1. 2.]
print("Log base 2:", np.log2([1, 2, 4, 8]))        # [0. 1. 2. 3.]

# Rounding functions
print("Around (2 decimals):", np.around([1.234, 2.567], 2))  # [1.23 2.57]
print("Floor:", np.floor([1.2, 2.9, -3.7]))              # [ 1.  2. -4.]
print("Ceil:", np.ceil([1.2, 2.9, -3.7]))               # [ 2.  3. -3.]
```

OUTPUT:

```
Exponential: [2.71828183 7.3890561 ]
Natural Log: [0. 1. 2.]
Log base 10: [0. 1. 2.]
Log base 2: [0. 1. 2. 3.]
Around (2 decimals): [1.23 2.57]
Floor: [ 1.  2. -4.]
Ceil: [ 2.  3. -3.]
```

Example Program: Arithmetic and Trigonometric Functions, Exponential, Logarithmic, and Rounding Functions

```
import numpy as np
```

```
a = np.array([2, 4])
b = np.array([1, 3])

print("---- Arithmetic ----")
print("a + b =", np.add(a, b))      # [3 7]
print("a * b =", np.multiply(a, b)) # [2 12]

print("\n---- Trigonometric ----")
angles = np.array([0, np.pi/2])     # 0 and 90 degrees in radians
print("sin:", np.sin(angles))       # [0. 1.]
print("cos:", np.cos(angles))       # [1. 0.]

print("\n---- Exponential & Logarithm ----")
print("exp([1,2]) =", np.exp([1,2]))     # [ 2.718  7.389]
print("log([1, e]) =", np.log([1, np.e])) # [0. 1.]

print("\n---- Rounding ----")
nums = np.array([2.6, -2.6])
print("around:", np.around(nums))   # [ 3. -3.]
print("floor:", np.floor(nums))     # [ 2. -3.]
print("ceil:", np.ceil(nums))       # [ 3. -2.]
```

OUTPUT:

```
---- Arithmetic ----
a + b = [3 7]
a * b = [ 2 12]

---- Trigonometric ----
sin: [0. 1.]
cos: [1. 0.]

---- Exponential & Logarithm ----
exp([1,2]) = [2.71828183 7.3890561 ]
log([1, e]) = [0. 1.]

---- Rounding ----
around: [ 3. -3.]
floor: [ 2. -3.]
ceil: [ 3. -2.]
```

Example Program: Arithmetic and Trigonometric Functions, Exponential, Logarithmic, and Rounding Functions

```
import numpy as np

# Sample arrays
a = np.array([10, 20, 30])
b = np.array([2, 5, 10])

print("---- Arithmetic Functions ----")
print("Add:", np.add(a, b))         # [12 25 40]
print("Subtract:", np.subtract(a, b)) # [ 8 15 20]
print("Multiply:", np.multiply(a, b)) # [ 20 100 300]
print("Divide:", np.divide(a, b))     # [5. 4. 3.]
print("Power:", np.power([2, 3, 4], 2))  # [ 4  9 16]
print("Mod:", np.mod(a, 7))         # [3 6 2]
```

```
print("\n---- Trigonometric Functions ----")
angles = np.array([0, 30, 45, 90])
print("Sine:", np.sin(np.deg2rad(angles)))    # [0. 0.5 0.7071 1.]
print("Cosine:", np.cos(np.deg2rad(angles)))  # [1. 0.866 0.7071 0.]
print("Tangent:", np.tan(np.deg2rad([0, 45])))# [0. 1.]
print("Arcsin:", np.arcsin([0, 1]))         # [0. 1.5708]
print("Deg → Rad:", np.deg2rad([180]))       # [3.1416]
print("Rad → Deg:", np.rad2deg([np.pi]))     # [180.]

print("\n---- Exponential & Logarithmic Functions ----")
x = np.array([1, 2, 4, 8])
print("Exponential:", np.exp([1, 2]))         # [2.718 7.389]
print("Natural log:", np.log([1, np.e, np.e**2])) # [0. 1. 2.]
print("Log base 10:", np.log10([1, 10, 100]))   # [0. 1. 2.]
print("Log base 2:", np.log2(x))              # [0. 1. 2. 3.]

print("\n---- Rounding Functions ----")
nums = np.array([1.234, 2.567, -3.789])
print("Around (2 decimals):", np.around(nums, 2)) # [ 1.23  2.57 -3.79]
print("Floor:", np.floor(nums))               # [ 1.  2. -4.]
print("Ceil:", np.ceil(nums))                 # [ 2.  3. -3.]
```

OUTPUT:

```
---- Arithmetic Functions ----
Add: [12 25 40]
Subtract: [ 8 15 20]
Multiply: [  20  100  300]
Divide: [5. 4. 3.]
Power: [ 4  9 16]
Mod: [3 6 2]

---- Trigonometric Functions ----
Sine: [0.     0.5    0.7071 1.    ]
Cosine: [1.     0.866  0.7071 0.    ]
Tangent: [0. 1.]
Arcsin: [0.     1.5708]
Deg → Rad: [3.1416]
Rad → Deg: [180.]

---- Exponential & Logarithmic Functions ----
Exponential: [2.71828183 7.3890561 ]
Natural log: [0. 1. 2.]
Log base 10: [0. 1. 2.]
Log base 2: [0. 1. 2. 3.]

---- Rounding Functions ----
Around (2 decimals): [ 1.23  2.57 -3.79]
Floor: [ 1.  2. -4.]
Ceil: [ 2.  3. -3.]
```

5.6 pandas packages

When we create a **DataFrame** (a 2D table of rows and columns), pandas provide several **built-in** functions to help you inspect or understand **the structure, data types, and content** of our data before starting analysis.

Array Inspection in pandas are classified into three groups:

- ***Structural Inspection***
- ***Data Type & Content Inspection***
- ***Missing Data Inspection***

Structural Inspection

These methods tell us about the **shape, size, and labels** of the DataFrame.

Method	Syntax	Example	Output	Explanation
shape	df.shape	df.shape	(rows, columns)	Dimensions of the DataFrame
ndim	df.ndim	df.ndim	2	Number of dimensions (1 for Series, 2 for DataFrame)
size	df.size	df.size	12	Total number of elements (rows × columns)
index	df.index	df.index	RangeIndex(0, 4)	Displays row index/labels
columns	df.columns	df.columns	Index([...])	Displays column labels
values	df.values	df.values	array([...])	Numpy array representation of the data

Data Type & Content Inspection

These methods help to check **data types, summary, and preview** of the data.

Method	Syntax	Example	Output	Explanation
dtypes	df.dtypes	df.dtypes	int64, float64, object	Shows datatype of each column
info()	df.info()	df.info()	Summary of DataFrame	Gives index range, non-null counts, column dtypes, memory usage
head(n)	df.head(n)	df.head(3)	First 3 rows	Displays first n rows
tail(n)	df.tail(n)	df.tail(2)	Last 2 rows	Displays last n rows
describe()	df.describe()	df.describe()	Summary stats	Gives mean, std, min, max, etc. for numeric columns

Missing Data Inspection

These methods help identify and handle **missing values (NaN/None)**.

Method	Syntax	Examplea	Output	Explanation
isnull()	df.isnull()	df.isnull()	DataFrame of True/False	Shows True where data is missing
notnull()	df.notnull()	df.notnull()	DataFrame of True/False	Shows True where data is not missing
isna()	df.isna()	df.isna()	Same as isnull()	Alias for isnull()
notna()	df.notna()	df.notna()	Same as notnull()	Alias for notnull()

NOTE:

- **Shape, ndim, size** → Structure of DataFrame
- **Columns, index, dtypes, values** → Metadata & types
- **Head, tail** → Peek at data
- **Info, describe** → Summary of data

- **isna / notna** → Missing values check

Example Program: Student Data covering Array Inspection built-in functions of packages NumPy

```
import pandas as pd

# Sample DataFrame
data = {
   "Name": ["Rahul", "Bharath", "Manoj", "Rakesh"],
   "Age": [35, 38, 36, 35],
   "Marks": [85.5, 91.0, 78.0, None]
}
df = pd.DataFrame(data)

print("Shape:", df.shape)
print("Dimensions:", df.ndim)
print("Size:", df.size)
print("\nData Types:\n", df.dtypes)
print("\nIndex:", df.index)
print("Columns:", df.columns)
print("\nValues:\n", df.values)
print("\nHead:\n", df.head(2))
print("\nTail:\n", df.tail(2))
print("\nInfo:"); print(df.info())
print("\nDescribe:\n", df.describe())
print("\nMissing values:\n", df.isna())
print("\nNot Missing values:\n", df.notna())
```

OUTPUT:

```
Shape: (4, 3)
Dimensions: 2
Size: 12

Data Types:
Name     object
Age       int64
Marks   float64
dtype: object

Index: RangeIndex(start=0, stop=4, step=1)
Columns: Index(['Name', 'Age', 'Marks'], dtype='object')

Values:
[['Rahul' 35 85.5]
 ['Bharath' 38 91.0]
 ['Manoj' 36 78.0]
 ['Rakesh' 35 nan]]

Head:
     Name  Age  Marks
0   Rahul   35   85.5
1  Bharath   38   91.0

Tail:
    Name  Age  Marks
2   Manoj   36   78.0
```

```
3  Rakesh   35    NaN

Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    4 non-null      object
 1   Age     4 non-null      int64
 2   Marks   3 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 228.0+ bytes
None

Describe:
            Age      Marks
count   4.000000   3.000000
mean   36.000000  84.833333
std     1.414214   6.658328
min    35.000000  78.000000
max    38.000000  91.000000

Missing values:
    Name    Age  Marks
0  False  False  False
1  False  False  False
2  False  False  False
3  False  False   True

Not Missing values:
    Name   Age  Marks
0   True  True   True
1   True  True   True
2   True  True   True
3   True  True  False
```

Example Program: Product Data covering Array Inspection built-in functions of packages NumPy

```
import pandas as pd

# Sample DataFrame
products = {
    "Product": ["Pen", "Pencil", "Eraser", "Notebook", "Scale"],
    "Price": [10, 5, 3, 50, 7],
    "Stock": [100, 200, None, 50, 80]
}
df2 = pd.DataFrame(products)

print("Shape:", df2.shape)
print("Dimensions:", df2.ndim)
print("Size:", df2.size)
print("\nData Types:\n", df2.dtypes)
print("\nIndex:", df2.index)
print("Columns:", df2.columns)
print("\nValues:\n", df2.values)
```

```
print("\nHead:\n", df2.head(3))
print("\nTail:\n", df2.tail(2))
print("\nInfo:"); print(df2.info())
print("\nDescribe:\n", df2.describe())
print("\nMissing values:\n", df2.isna())
print("\nNot Missing values:\n", df2.notna())
```

OUTPUT:

```
Shape: (5, 3)
Dimensions: 2
Size: 15

Data Types:
Product    object
Price       int64
Stock     float64
dtype: object

Index: RangeIndex(start=0, stop=5, step=1)
Columns: Index(['Product', 'Price', 'Stock'], dtype='object')

Values:
[['Pen' 10 100.0]
 ['Pencil' 5 200.0]
 ['Eraser' 3 nan]
 ['Notebook' 50 50.0]
 ['Scale' 7 80.0]]

Head:
   Product  Price  Stock
0      Pen     10  100.0
1   Pencil      5  200.0
2   Eraser      3    NaN

Tail:
    Product  Price  Stock
3  Notebook     50   50.0
4     Scale      7   80.0

Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Product  5 non-null      object
 1   Price    5 non-null      int64
 2   Stock    4 non-null      float64
dtypes: float64(1), int64(1), object(1)

Describe:
         Price       Stock
count   5.00000    4.000000
mean   15.00000  107.500000
std    19.62142   70.710678
min     3.00000   50.000000
max    50.00000  200.000000
```

```
Missing values:
  Product  Price  Stock
0   False  False  False
1   False  False  False
2   False  False   True
3   False  False  False
4   False  False  False
```

Example Program: Fruits and Prices covering Array Inspection built-in functions of packages NumPy

```
import pandas as pd

# Sample DataFrame
data = {
   'Fruit': ['Apple', 'Banana', 'Mango'],
   'Price': [120, 60, None],
   'Category': ['Premium', 'Regular', 'Seasonal']
}

df = pd.DataFrame(data)

# Inspection
print("Shape:", df.shape)
print("Dimensions:", df.ndim)
print("Size:", df.size)
print("Columns:", df.columns)
print("Data Types:\n", df.dtypes)
print("Missing Values:\n", df.isnull())
print("Info:")
df.info()
```

OUTPUT:

```
Shape: (3, 3)
Dimensions: 2
Size: 9
Columns: Index(['Fruit', 'Price', 'Category'], dtype='object')

Data Types:
Fruit      object
Price     float64
Category    object
dtype: object

Missing Values:
  Fruit  Price  Category
0  False  False    False
1  False  False    False
2  False   True    False

Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
```

```
 0  Fruit     3 non-null     object
 1  Price     2 non-null     float64
 2  Category  3 non-null     object
dtypes: float64(1), object(2)
memory usage: 200.0+ bytes
```

5.7 GUI PROGRAMMING WITH Tkinter

5.7.1 INTRODUCTION TO Tkinter

- A GUI (Graphical User Interface) allows users to interact with software using windows, buttons, menus, forms, etc., instead of typing commands.
- **Tkinter** is Python's **standard library for GUI (Graphical User Interface) programming**.
- It comes bundled with Python (no need for extra installation in most cases).
- It is built on top of the **Tcl/Tk toolkit**. It is **cross-platform**: works on Windows, macOS, and Linux.
- Tkinter allows developers to create windows, buttons, labels, text boxes, menus, and many other GUI elements.
- Tkinter comes bundled with Python (no separate installation needed).
- It is simple for beginners yet powerful enough for academic and small professional projects.

ADVANTAGES OF Tkinter

- **Beginner-friendly:** *Easy to learn and use.*
- **Cross-platform:** *Works on Windows, Mac, and Linux.*
- **Widget-based:** *Provides elements like buttons, labels, text boxes, menus, etc.*
- **Event-driven:** *GUI runs based on user actions (click, type, select).*
- **No external installation required:** *Comes pre-installed with Python*.
- **Simple syntax:** *Enables rapid development.*
- **Ideal for prototyping and small desktop applications:** *Quick way to build functional GUIs.*

Tkinter AND PYTHON PROGRAMMING

Python is a versatile, high-level programming language widely used in software development and education. Tkinter is its standard GUI library, enabling developers to build interactive desktop applications with minimal code. Together, Python and Tkinter offer a powerful platform for creating user-friendly interfaces, ideal for teaching event-driven programming and rapid application development.

Tkinter Basic Syntax

```
import tkinter as tk                    # 1. Import Tkinter

root = tk.Tk()                          # 2. Create the main window (root)
root.title("Window Title")              # 3. Optional: set window title

widget = tk.WidgetName(root, options)   # 4. Create and configure widgets (Label,
                                        Button, etc.)
```

```
widget.pack()          # 5. Place (layout) widgets in the window

root.mainloop()        # 6. Start the event loop
```

Explanation

1. **Import Tkinter:** ***(import tkinter as tk)***
   - tkinter is the Python library for GUI programming.
   - as tk is an alias to make code shorter (so we write tk.Label instead of tkinter.Label).
2. **Create Main Window:**
   root = tk.Tk()
   - Tk() initializes the main application window.
   - The variable root represents the window (you can name it anything).
3. **Set Window Title (Optional):**
   root.title("My App")
   You can specify what appears in the title bar.
4. **Add Widgets:**
   widget = tk.WidgetName(root, options)
   widget.pack()

   Widgets are GUI elements like **Label, Button, Entry, Text, Menu**.
   Example:
   label = tk.Label(root, text="Hello Tkinter")
   button = tk.Button(root, text="Click Me")

   pack(), grid(), or place() methods are used to **arrange widgets** inside the window.
5. **Arrange Widgets:**
   You need to tell Tkinter where to put widgets uing layout managers
   like .pack(), .grid(), or .place().

   Example:
   label.pack()
   btn.pack()
6. **Start the Main Loop:**
   root.mainloop()
   - *This keeps the window open and waits for user actions (like button clicks, typing, etc.).*
   - *Without this line, the GUI will close immediately after opening.*

5.7.2 Tk Widgets (Tkinter Widgets)

In Tkinter, **widgets** are the building blocks of a GUI application. They are small elements like buttons, text boxes, labels, menus, etc., that allow users to interact with the program.

HOW DO WIDGETS WORK?

- Each widget is an **object (instance of a class)**.
- We create a widget (like a Button or Label).

- Then we **place it** inside the main window using geometry managers (pack(), grid(), place()).
- Widgets can be customized (size, color, text, font, etc.).
- Widgets respond to user actions (clicking, typing, selecting, etc.).

Table: TYPES OF WIDGETS

Type	Widget	Purpose	Syntax	Explanation
Basic Widgets	**Label**	Display static text or images	Label(master, options)	Used to show text or images that cannot be edited by the user.
	Button	Creates a clickable button	Button(master, options)	Triggers an action or command on click.
	Entry	Single-line text input	Entry(master, options)	Allows the user to enter or edit text in one line.
	Frame	Container for other widgets	Frame(master, options)	Used to group and organize other widgets.
Selection Widgets	**Checkbutton**	Checkbox for multiple options	Checkbutton(master, options)	Used for on/off or multiple selections.
	Radiobutton	Radio button for one option	Radiobutton(master, options)	Used when only one option among many can be selected.
	Listbox	Displays a list of items	Listbox(master, options)	Allows the user to select one or more items from a list.
	Combobox (ttk)	Dropdown menu	ttk.Combobox(master, options)	Provides a dropdown list with text entry.
	Spinbox	Numeric input with arrows	Spinbox(master, options)	Lets the user select from a range of numbers.
	Scale	Slider for numeric values	Scale(master, options)	Allows selecting values within a range using a slider.
Display Widgets	**Text**	Multi-line text input	Text(master, options)	Allows editing of multi-line text.
	Canvas	Drawing area	Canvas(master, options)	Used for shapes, images, or custom drawings.
	Message	Displays text messages	Message(master, options)	Similar to Label but supports longer texts.
	Progressbar (ttk)	Shows task progress	ttk.Progressbar(master, options)	Visual representation of task progress.
	Treeview (ttk)	Displays hierarchical data	ttk.Treeview(master, options)	Used for tables, file browsers, etc.
Container & Menu Widgets	**Menu**	Creates menus	Menu(master, options)	Used to build menus like File, Edit, etc.
	Menubutton	Button with dropdown menu	Menubutton(master, options)	Displays a menu when clicked.
	Toplevel	Creates a new window	Toplevel(master, options)	Opens additional windows.
Scrollbar Widgets	**Scrollbar**	Adds scroll bar to widgets	Scrollbar(master, options)	Provides scrolling for Text, Listbox, Canvas, etc.
	Treeview Scrollbar	Adds scrollbar to Treeview	Scrollbar(master, options)	Enables scrolling inside Treeview.

Dialog Widgets	**Messagebox**	Popup dialogs	messagebox.showinfo("Title", "Message")	Shows alerts, warnings, or confirmations.

GEOMETRY MANAGERS

Manager	Syntax	Purpose	Simple Explanation
pack()	***widget.pack(options)***	Arranges widgets in blocks (top, bottom, left, right).	Places widgets **one after another** in a sequence. Useful for simple layouts. Options like side, fill, expand control placement.
grid()	***widget.grid(row=r, column=c, options)***	Arranges widgets in a table-like structure (rows & columns).	Divides the window into **rows and columns**. Widgets can be positioned at specific cells. Best for forms or structured layouts.
place()	***widget.place(x=val, y=val)***	Places widgets at specific coordinates.	Provides **exact control** over position using x & y (pixels) or relative values (relx, rely). Best for custom designs.

Note:

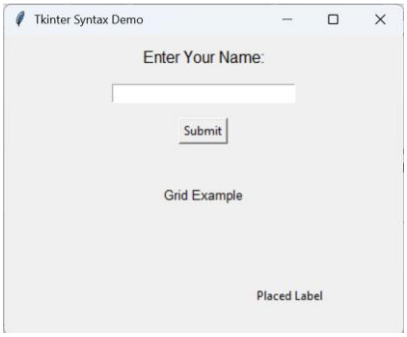
- Use **pack()** → for simple stacking (quick demos).
- Use **grid()** → for structured forms (login, calculator).
- Use **place()** → when you need **exact positions** (games, designs).

Tkinter Syntax in detail

```
import tkinter as tk
from tkinter import messagebox

# Create main window first
window = tk.Tk()
window.title("Tkinter Syntax Demo")
window.geometry("400x300")  # Set window size

# Special variable example (must come AFTER window)
name_var = tk.StringVar()

# Event handler function
def on_click():
    messagebox.showinfo("Greeting", f"Hello, {name_var.get()}!")

# Label widget
label = tk.Label(window, text="Enter Your Name:", font=("Arial", 12))
label.pack(pady=10)

# Entry widget
entry = tk.Entry(window, textvariable=name_var, width=30)
entry.pack(pady=5)

# Button widget
button = tk.Button(window, text="Submit", command=on_click)
button.pack(pady=10)
```

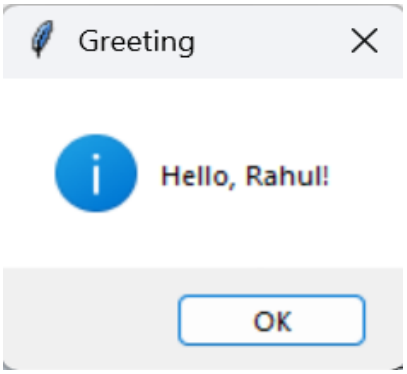
```
    # Frame for grid layout
    frame = tk.Frame(window)
    frame.pack(pady=20)

    grid_label = tk.Label(frame, text="Grid Example", font=("Arial", 10))
    grid_label.grid(row=0, column=0, padx=10, pady=10)

    # Place layout example
    place_label = tk.Label(window, text="Placed Label")
    place_label.place(x=250, y=250)

    # Start the event loop
window.mainloop()
```

Output:
A window titled "Tkinter Syntax Demo" opens.



You'll see:
A label: "Enter Your Name:"
An entry box
A "Submit" button
A frame with the text "Grid Example"
A placed label at bottom-right: "Placed Label"

If you type a name (e.g., Rahul) and click Submit, a popup appears:



Greeting
Hello, Rahul!

Explanation

Component	Syntax	Explanation
Import Tkinter	***import tkinter as tk***	Imports the Tkinter module with alias tk for easy usage.
window= tk.Tk()	***Create main window***	Create main window using tkinter
Event Handler Function	***def on_click():***	Defines a function that runs when the button is clicked.

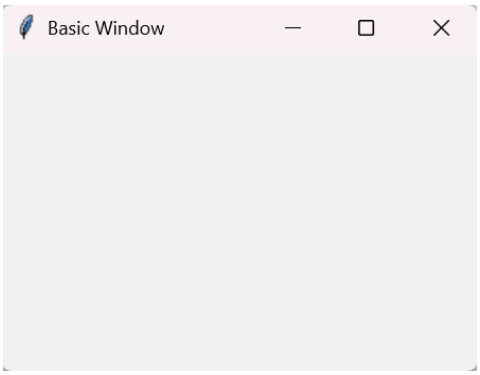
Show Info Popup with Name	***messagebox.showinfo("Greeting", f"Hello, {name_var.get()}!")***	Displays a popup greeting with the user's entered name (retrieved using name_var.get()).
Special Variable	***name_var = tk.StringVar()***	Creates a StringVar to hold dynamic text input from the Entry widget.
Create Main Window	***window = tk.Tk()***	Initializes the main application window.
Set Window Title	***window.title("Tkinter Syntax Demo")***	Sets the title bar text of the window.
Set Window Size	***window.geometry("400x300")***	Defines the window size (400px × 300px).
Label Widget	***label = tk.Label(window, text="Enter Your Name:", font=("Arial", 12))***	Creates a label with text "Enter Your Name:" in Arial font size 12.
Pack Layout (Label)	***label.pack(pady=10)***	Places the label in the window with vertical padding of 10px.
Entry Widget	***entry = tk.Entry(window, textvariable=name_var, width=30)***	Creates a single-line text box linked to name_var for user input.
Pack Layout (Entry)	***entry.pack(pady=5)***	Positions the entry box with vertical padding of 5px.
Button Widget	***button = tk.Button(window, text="Submit", command=on_click)***	Creates a button labeled "Submit" that triggers on_click when clicked.
Pack Layout (Button)	***button.pack(pady=10)***	Places the button in the window with vertical padding of 10px.
Frame Widget	***frame = tk.Frame(window)***	Creates a frame (container) inside the main window for grouping widgets.
Pack Layout (Frame)	***frame.pack(pady=20)***	Positions the frame with vertical padding of 20px.
Label with Grid (inside Frame)	***grid_label = tk.Label(frame, text="Grid Example", font=("Arial", 10))***	Creates a label inside the frame with text "Grid Example".
Grid Layout	***grid_label.grid(row=0, column=0, padx=10, pady=10)***	Places the label at row=0, column=0 inside the frame using grid layout.
Label with Place	***place_label = tk.Label(window, text="Placed Label")***	Creates a label with text "Placed Label".
Place Layout	***place_label.place(x=250, y=250)***	Positions the label at absolute coordinates (250, 250) in the window.
Event Loop	***window.mainloop()***	Starts the Tkinter event loop to keep the window running and responsive.

Example: Basic Window

```
import tkinter as tk   # Import Tkinter library

# Create main window
window = tk.Tk()       # Initialize the window
window.title("Basic Window")  # Set window title
```

```
window.geometry("300x200")    # Set window size

# Run the GUI loop
window.mainloop()
```

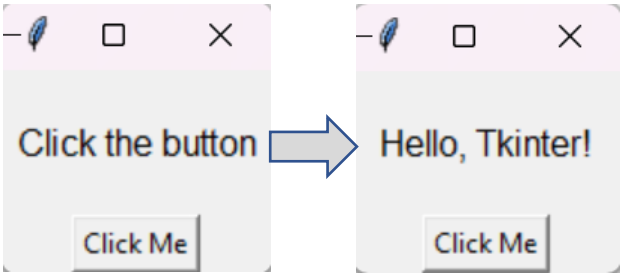
Output:



Example: Label and Button

```
import tkinter as tk   # Import Tkinter library

# Function to change label text
def say_hello():
   label.config(text="Hello, Tkinter!")

window = tk.Tk()            # Create main window
window.title("Label and Button")

# Create label
label = tk.Label(window, text="Click the button", font=("Arial", 12))
label.pack(pady=20)   # Place label with padding

# Create button
button = tk.Button(window, text="Click Me", command=say_hello)
button.pack()        # Place button below label

# Run the GUI loop
window.mainloop()
```

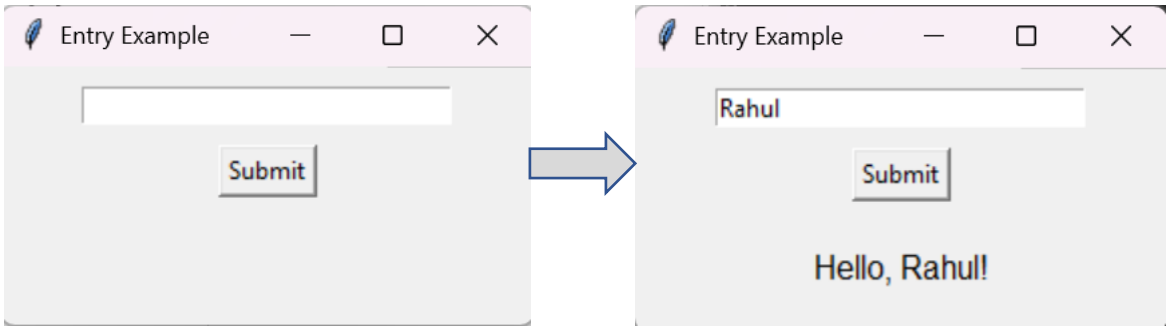
Output:



Example: Entry Widget (User Input)

```
import tkinter as tk

# Function to display entered name
def show_name():
    name = entry.get()     # Get input text from entry
    label.config(text=f"Hello, {name}!")  # Update label

window = tk.Tk()            # Create window
window.title("Entry Example")

# Entry widget for user input
entry = tk.Entry(window, width=30)
entry.pack(pady=10)

# Button to trigger function
button = tk.Button(window, text="Submit", command=show_name)
button.pack()
# Label to display result
label = tk.Label(window, text="", font=("Arial", 12))
label.pack(pady=20)

# Run the GUI loop
window.mainloop()
```

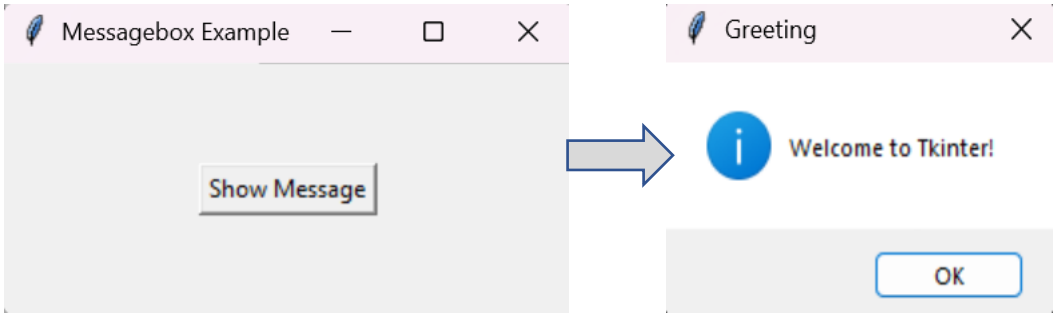
Output:



Example: Messagebox

```
import tkinter as tk
from tkinter import messagebox   # Import messagebox for popups

# Function to show messagebox
def show_message():
    messagebox.showinfo("Greeting", "Welcome to Tkinter!")

window = tk.Tk()            # Create main window
window.title("Messagebox Example")

# Button to show popup
button = tk.Button(window, text="Show Message", command=show_message)
button.pack(pady=50)

# Run the GUI loop
window.mainloop()
```

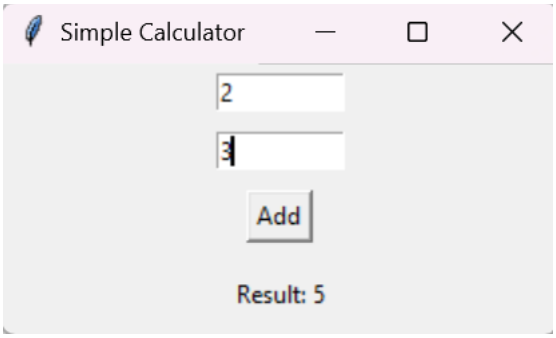
Output:



Example: Calculator (Addition Only)

```
import tkinter as tk

# Function to add numbers
def add_numbers():
    num1 = int(entry1.get())   # Get first number
    num2 = int(entry2.get())   # Get second number
    result_label.config(text=f"Result: {num1 + num2}")  # Show sum

window = tk.Tk()               # Create window
window.title("Simple Calculator")

# Entry for first number
entry1 = tk.Entry(window, width=10)
entry1.pack(pady=5)

# Entry for second number
entry2 = tk.Entry(window, width=10)
entry2.pack(pady=5)

# Button to add numbers
button = tk.Button(window, text="Add", command=add_numbers)
button.pack(pady=5)

# Label to display result
result_label = tk.Label(window, text="Result: ")
result_label.pack(pady=10)

# Run the GUI loop
window.mainloop()
```

Output:

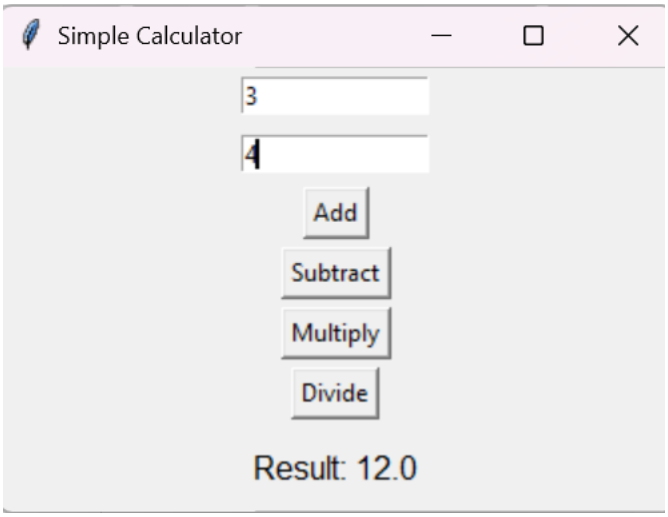


Example: Calculator with Add, Subtract, Multiply & Divide operations

```
import tkinter as tk   # Import Tkinter library

# Function to add numbers
def add_numbers():
    num1 = float(entry1.get())      # Get first number
    num2 = float(entry2.get())      # Get second number
    result_label.config(text=f"Result: {num1 + num2}")   # Display sum

# Function to subtract numbers
def subtract_numbers():
    num1 = float(entry1.get())
    num2 = float(entry2.get())
    result_label.config(text=f"Result: {num1 - num2}")   # Display difference

# Function to multiply numbers
def multiply_numbers():
    num1 = float(entry1.get())
    num2 = float(entry2.get())
    result_label.config(text=f"Result: {num1 * num2}")   # Display product

# Function to divide numbers
def divide_numbers():
    num1 = float(entry1.get())
    num2 = float(entry2.get())
    if num2 != 0:     # Check division by zero
        result_label.config(text=f"Result: {num1 / num2}")   # Display quotient
    else:
        result_label.config(text="Error: Division by Zero")

# Create main window
window = tk.Tk()
window.title("Simple Calculator")   # Title of the window

# Entry for first number
entry1 = tk.Entry(window, width=15)
entry1.pack(pady=5)

# Entry for second number
entry2 = tk.Entry(window, width=15)
entry2.pack(pady=5)

# Buttons for operations
add_button = tk.Button(window, text="Add", command=add_numbers)
add_button.pack(pady=2)

sub_button = tk.Button(window, text="Subtract", command=subtract_numbers)
sub_button.pack(pady=2)

mul_button = tk.Button(window, text="Multiply", command=multiply_numbers)
mul_button.pack(pady=2)
```

```
div_button = tk.Button(window, text="Divide", command=divide_numbers)
div_button.pack(pady=2)

# Label to show the result
result_label = tk.Label(window, text="Result: ", font=("Arial", 12))
result_label.pack(pady=10)

# Run the GUI loop
window.mainloop()
```

Output:

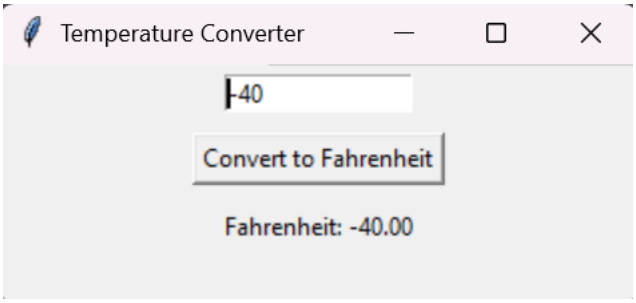


Example: Temperature Converter (Celsius ↔ Fahrenheit)

```
import tkinter as tk

# Convert Celsius to Fahrenheit
def convert():
    celsius = float(entry.get())
    fahrenheit = (celsius * 9/5) + 32
    result_label.config(text=f"Fahrenheit: {fahrenheit:.2f}")

# Main window
window = tk.Tk()
window.title("Temperature Converter")

# Entry for Celsius
entry = tk.Entry(window, width=15)
entry.pack(pady=5)

# Button to convert
convert_button = tk.Button(window, text="Convert to Fahrenheit",
command=convert)
convert_button.pack(pady=5)

# Result label
result_label = tk.Label(window, text="Fahrenheit: ")
```

```
result_label.pack(pady=5)

window.mainloop()
```

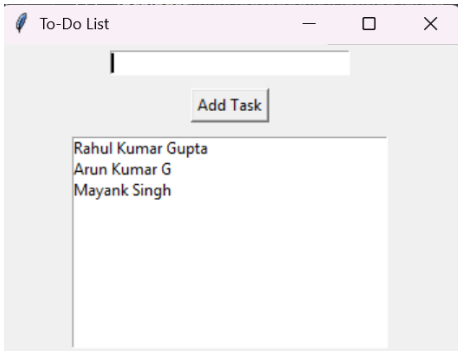
Output:



Example: Simple To-Do List

```
import tkinter as tk

# Add task to listbox
def add_task():
    task = entry.get()
    if task != "":
        listbox.insert(tk.END, task)
        entry.delete(0, tk.END)

# Main window
window = tk.Tk()
window.title("To-Do List")

# Entry box
entry = tk.Entry(window, width=30)
entry.pack(pady=5)

# Button to add task
add_button = tk.Button(window, text="Add Task", command=add_task)
add_button.pack(pady=5)

# Listbox to show tasks
listbox = tk.Listbox(window, width=40, height=10)
listbox.pack(pady=5)

window.mainloop()
```

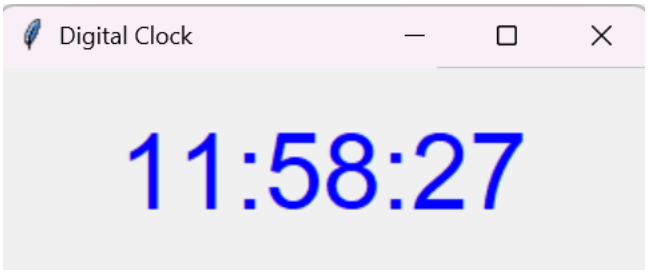
Output:



Example: Digital Clock

```
import tkinter as tk
import time

# Update time every second
def update_time():
    current_time = time.strftime("%H:%M:%S")
    label.config(text=current_time)
    window.after(1000, update_time)  # Refresh every 1 sec

# Main window
window = tk.Tk()
window.title("Digital Clock")

# Label to display time
label = tk.Label(window, font=("Arial", 40), fg="blue")
label.pack(pady=20)

# Start clock
update_time()
window.mainloop()
```

Output:



Example: Random Password Generator

```
import tkinter as tk
import random
import string

# Generate password
def generate_password():
    length = 8
    characters = string.ascii_letters + string.digits + string.punctuation
    password = "".join(random.choice(characters) for i in range(length))
    result_label.config(text=f"Password: {password}")

# Main window
window = tk.Tk()
window.title("Password Generator")

# Button
generate_button = tk.Button(window, text="Generate Password",
command=generate_password)
generate_button.pack(pady=10)
```

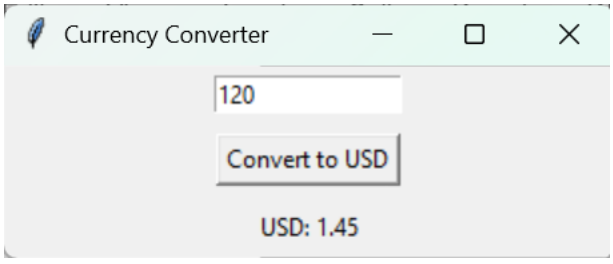
```
# Result
result_label = tk.Label(window, text="Password: ")
result_label.pack(pady=10)

window.mainloop()
```

Output:

Example: Currency Converter (Rupees ↔ Dollars)

```
import tkinter as tk

# Convert INR to USD
def convert():
   inr = float(entry.get())
   usd = inr / 83  # Approx rate
   result_label.config(text=f"USD: {usd:.2f}")

# Main window
window = tk.Tk()
window.title("Currency Converter")

# Entry for INR
entry = tk.Entry(window, width=15)
entry.pack(pady=5)

# Button
convert_button = tk.Button(window, text="Convert to USD", command=convert)
convert_button.pack(pady=5)

# Result
result_label = tk.Label(window, text="USD: ")
result_label.pack(pady=5)

window.mainloop()
```

Output:

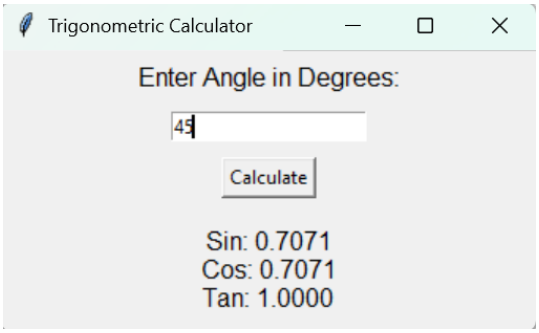


Example: Trigonometric Calculator (Sin, Cos, Tan)

```
import tkinter as tk
```

```
import math

# Function to calculate sin, cos, tan
def calculate():
    try:
        angle_deg = float(entry.get())        # Get input from user in degrees
        angle_rad = math.radians(angle_deg)    # Convert degree → radians

        sin_val = math.sin(angle_rad)         # Calculate sine
        cos_val = math.cos(angle_rad)         # Calculate cosine
        tan_val = math.tan(angle_rad)         # Calculate tangent

        result_label.config(text=f"Sin:         {sin_val:.4f}\nCos:         {cos_val:.4f}\nTan:
{tan_val:.4f}")
    except ValueError:
        result_label.config(text="Please enter a valid number!")

# Create main window
window = tk.Tk()
window.title("Trigonometric Calculator")

# Label
label = tk.Label(window, text="Enter Angle in Degrees:", font=("Arial", 12))
label.pack(pady=5)

# Entry box
entry = tk.Entry(window, width=20)
entry.pack(pady=5)

# Button
button = tk.Button(window, text="Calculate", command=calculate)
button.pack(pady=5)

# Result
result_label = tk.Label(window, text="", font=("Arial", 12))
result_label.pack(pady=10)

window.mainloop()
```

Output:



Explanation:

- User enters an **angle in degrees** (e.g., 30, 45, 60, 90).
- Program **converts** it into **radians** (since Python math functions use radians).

- Calculates sin, cos, tan using **math.sin(), math.cos(), math.tan().**
- **Displays** results up to **4 decimal places**.

ADVANTAGES OF Tkinter WIDGETS

1. **Easy to Use**
   - Tkinter widgets are **simple and beginner-friendly**.
   - They can be created and customized quickly using Python.
2. **Interactive GUI**
   - Widgets allow users to **interact with the program** (buttons, text input, checkboxes).
   - This makes applications more user-friendly compared to text-only programs.
3. **Wide Variety**
   - Tkinter provides **many widgets**: labels, buttons, entry fields, listboxes, canvas, menus, dialogs, and more.
   - Covers most needs for desktop GUI applications.
4. **Customizable**
   - Widgets can be **styled and configured** (text, font, color, size, border, etc.).
   - Supports building attractive and functional interfaces.
5. **Event Handling**
   - Widgets can **respond to user actions** like clicks, typing, or selection.
   - Supports building dynamic and interactive applications.
6. **Lightweight & Built-in**
   - Tkinter comes **pre-installed with Python**, so no additional installation is required.
   - Runs on **Windows, Mac, and Linux** without extra dependencies.
7. **Organized Layouts**
   - Widgets can be arranged neatly using **geometry managers** (pack, grid, place).
   - Supports building structured and easy-to-navigate interfaces.
8. **Extendable**
   - Widgets can be combined to create **complex GUIs** like forms, games, calculators, or editors.
   - Works well with additional libraries like ttk for modern themed widgets.

5.8 PYTHON IDE (Integrated Development Environment)

- An **IDE** is a software application that provides **tools to write, edit, debug, and run Python programs** efficiently.
- It combines a **code editor, debugger, and interpreter** in one interface.

5.8.1 CLASSIFICATION OF PYTHON IDEs

Python IDEs can be classified into **four main types** depending on their features, purpose, and target users:

Table: Classification of Python IDEs

Type	IDE	Description / Features
Beginner-Friendly / Educational IDEs	***IDLE***	Default IDE, simple interface, suitable for beginners and small programs.
	Thonny	Very easy for beginners, step-by-step debugger, simple interface.
	BlueJ	Focused on teaching programming and object-oriented concepts.
	• **Purpose:** Designed for **new learners** to practice Python programming. • **Features:** Simple interface, easy to use, basic debugging, minimal setup.	
Professional / Full-Featured IDEs	***PyCharm***	Advanced debugging, large project support, Git integration, refactoring.
	Wing IDE	Professional development, intelligent code analysis, productivity features.
	Komodo IDE	Supports multiple languages, debugging, and version control.
	Eric Python IDE	Good for PyQt applications, beginner-friendly but advanced enough for projects.
	PyDev (Eclipse Plugin)	Python plugin for Eclipse, supports professional development and Java integration.
	NetBeans (with Python plugin)	Full-featured IDE, suitable for large projects with multiple languages.
	• **Purpose:** For **professional development**, large projects, or commercial applications. • **Features:** Advanced debugging, project management, Git/version control, code refactoring, plugins support.	
Lightweight / Text Editor Based IDEs	***VS Code***	Lightweight editor, supports extensions, flexible for multiple languages.
	Atom	Hackable and customizable text editor with Python packages.
	Sublime Text	Fast, lightweight editor with Python plugin support.
	Geany	Lightweight, cross-platform, suitable for small projects.
	• **Purpose: Flexible and fast** coding, often used by students or developers who prefer minimal interfaces. • **Features:** Lightweight, customizable with plugins/extensions, supports multiple languages.	
Data Science / Interactive Coding IDEs	***Spyder***	Scientific IDE, integrates with data analysis and machine learning libraries.
	Jupyter Notebook	Interactive notebooks, supports code, text, visualizations; ideal for data science and learning.
	• **Purpose:** Designed for **data analysis, visualization, and scientific computing**. • **Features:** Supports interactive code execution, inline plots, notebooks, and integration with scientific libraries.	

PYTHON IDEs SYNTAX AND HOW TO LAUNCH

- An IDE (Integrated Development Environment) is a software tool used to write, run, and debug Python programs easily.
- Each IDE can be opened (launched) in different ways depending on how it is installed.

Table: Python IDEs Syntax and how to launch

Type	IDE	How to Launch / Syntax	Features
Beginner-Friendly / Educational	**IDLE**	idle (Windows: Start Menu → IDLE)	Default IDE; simple editor and console.
	Thonny	Launch from Start Menu / thonny in terminal	Beginner-friendly, step-by-step debugger.
	BlueJ	Launch from Start Menu / Installed application	Mainly for teaching object-oriented programming.
Professional / Full-Featured	**PyCharm**	Launch application or pycharm (if added to PATH)	Advanced debugging, project management, Git integration.
	Wing IDE	Launch from Start Menu / Installed app	Professional IDE with intelligent code analysis.
	Komodo IDE	Launch application	Supports multiple languages, version control.
	Eric Python IDE	Launch from Start Menu / Terminal	Good for PyQt development.
	PyDev (Eclipse Plugin)	Eclipse → PyDev perspective	Python plugin inside Eclipse IDE.
	NetBeans (with Python plugin)	NetBeans → Open Python project	Full-featured IDE, multi-language support.
Lightweight / Text Editor Based	**VS Code**	code in terminal / Launch from app	Requires Python extension for IDE features.
	Atom	Launch application	Install Python packages for full IDE support.
	Sublime Text	Launch application	Can install Python plugins for IDE functionality.
	Geany	Launch application	Lightweight editor, supports Python scripting.
Data Science / Interactive Coding	**Spyder**	Launch from Anaconda Navigator or terminal: spyder	Scientific IDE, integrates with Python libraries.
	Jupyter Notebook	Terminal: jupyter notebook	Opens browser interface for interactive notebooks.

BASIC STEPS TO USE A PYTHON IDE

1. *Install/Launch IDE (if not IDLE, install PyCharm or VS Code).*
2. *Create a new Python file (.py).*
3. *Write Python code in the editor.*
4. *Run the code using Run/Execute button or shortcut (e.g., F5).*
5. *View output in console/terminal.*

BENEFITS OF USING AN IDE

1. **All-in-One Environment**
   - Combines editor, debugger, terminal, and project management in one place.
   - No need to switch between multiple tools.
2. **Easy Code Writing**
   - Provides syntax highlighting (different colors for keywords, variables, strings).
   - Auto-completion suggests functions, variables, and libraries while typing.
3. **Debugging Support**
   - Helps find and fix errors with breakpoints and step-by-step execution.
   - Shows where the error occurred with clear messages.
4. **Code Management**
   - Organizes large projects with multiple files easily.
   - Supports version control (Git, SVN).
5. **Productivity Boost**
   - Built-in shortcuts and templates speed up coding.
   - Some IDEs have intelligent code analysis (suggest better ways to write code).
6. **Integrated Testing**
   - Many IDEs support unit testing and test frameworks directly.
   - Useful for professional and large-scale projects.
7. **Cross-Platform Development**
   - Most IDEs work on Windows, macOS, and Linux.
   - Makes collaboration easier across different systems.
8. **Specialized Tools**
   - Data Science IDEs (like Spyder, Jupyter) support plots, graphs, and interactive coding.
   - Web development IDEs (like VS Code, PyCharm) support frameworks (Django, Flask).

Note: IDEs that Support Multiple Languages

IDE	Languages Supported
VS Code	*Python, JS, C++, Java, HTML, etc.*
Eclipse	*Java, C++, Python (via plugins)*
IntelliJ IDEA	*Java, Kotlin, Python, JS*
PyCharm	*Python, HTML, JS*
JupyterLab	*Python, R, Julia*
Replit	*Python, JS, C++, Java, Bash*

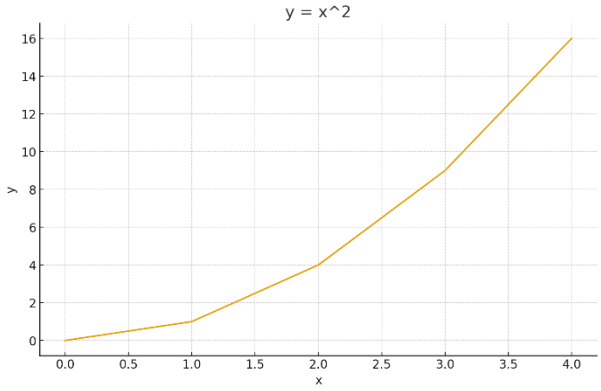
Example Programs

Example Python Programs – Unit-5 (Packages)

Q1. Matplotlib – Simple line plot

```
import matplotlib.pyplot as plt
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]
plt.plot(x, y)
plt.title("y = x^2")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Output:

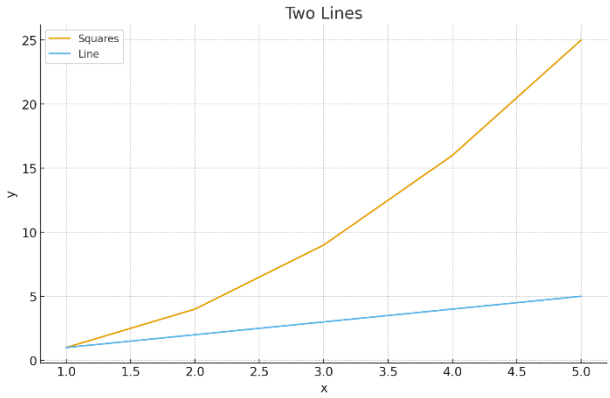


Explanation: plot() draws a basic line; labels/title add context.

Q2. Matplotlib – Line plot with legend & grid

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y1 = [1,4,9,16,25]
y2 = [1,2,3,4,5]
plt.plot(x, y1, label="Squares")
plt.plot(x, y2, label="Line")
plt.title("Two Lines")
plt.xlabel("x"); plt.ylabel("y")
plt.legend(); plt.grid(True)
plt.show()
```

Output:

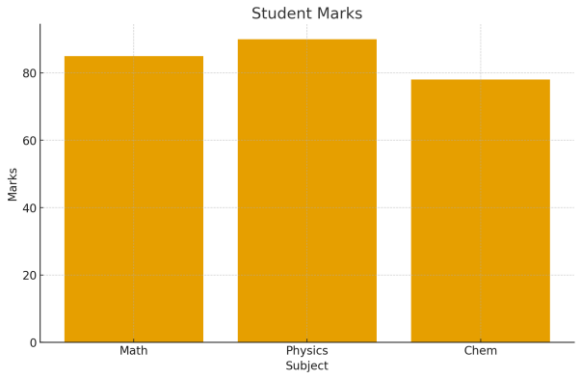


Explanation: legend() identifies each series; grid(True) improves readability.

Q3. Matplotlib – Bar chart

```
import matplotlib.pyplot as plt
subjects = ["Math","Physics","Chem"]
marks = [85, 90, 78]
plt.bar(subjects, marks)
plt.title("Student Marks"); plt.xlabel("Subject"); plt.ylabel("Marks")
plt.show()
```

Output: Vertical bars for the three subjects.

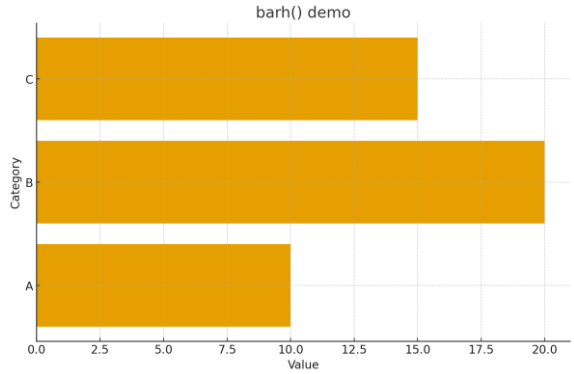


Explanation: bar() maps categories to bar heights.

Q4. Matplotlib – Horizontal bar chart

```
import matplotlib.pyplot as plt
cats = ["A","B","C"]
vals = [10, 20, 15]
plt.barh(cats, vals)
plt.xlabel("Value"); plt.ylabel("Category"); plt.title("barh() demo")
plt.show()
```

Output: Three horizontal bars (A=10, B=20, C=15).

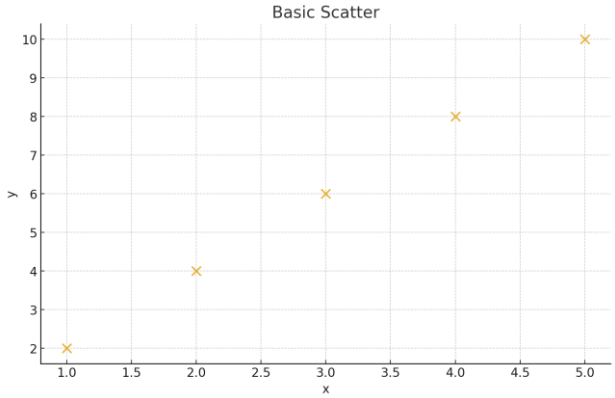


Explanation: barh() plots data horizontally.

Q5. Matplotlib – Scatter plot

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y = [2,4,6,8,10]
plt.scatter(x, y, s=80, alpha=0.8, edgecolors="black")
plt.title("Basic Scatter"); plt.xlabel("x"); plt.ylabel("y")
plt.show()
```

Output: Points along the line y=2x.

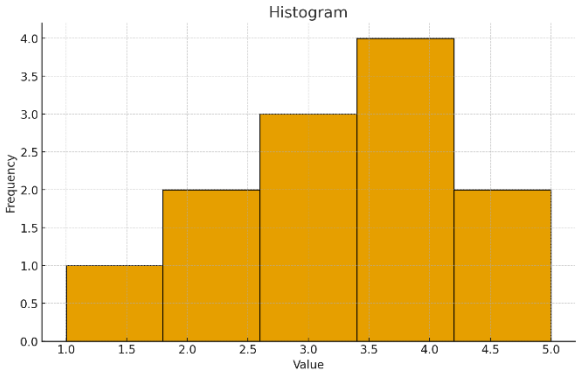


Explanation: scatter() plots individual points with size/alpha options.

Q6. Matplotlib – Histogram

```
import matplotlib.pyplot as plt
data = [1,2,2,3,3,3,4,4,4,4,5,5]
plt.hist(data, bins=5, edgecolor="black")
plt.xlabel("Value"); plt.ylabel("Frequency"); plt.title("Histogram")
plt.show()
```

Output: Histogram bars showing frequency by bin.

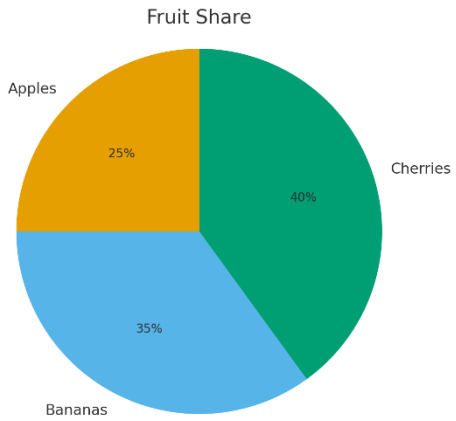


Explanation: hist() groups values into bins.

Q7. Matplotlib – Pie chart

```
import matplotlib.pyplot as plt
sizes = [25, 35, 40]
labels = ["Apples","Bananas","Cherries"]
plt.pie(sizes, labels=labels, autopct="%.0f%%", startangle=90)
plt.axis("equal")
plt.title("Fruit Share")
plt.show()
```

Output: Pie chart with percentages.

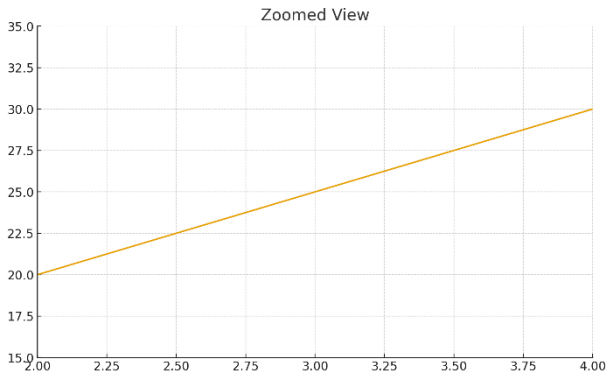


Explanation: pie() shows proportions; autopct prints percent.

Q8. Matplotlib – Axis limits (xlim/ylim)

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y = [10,20,25,30,40]
plt.plot(x, y)
plt.xlim(2, 4); plt.ylim(15, 35)
plt.title("Zoomed View")
plt.show()
```

Output: Plot cropped to x=2..4 and y=15..35.



Explanation: xlim/ylim zoom into a range.

Q9. NumPy – Array creation basics

```
import numpy as np
a = np.array([1,2,3])
z = np.zeros((2,2))
o = np.ones((2,3))
print(a); print(z); print(o)
```

Output:

```
[1 2 3]
[[0. 0.]
 [0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

Explanation:

- np.array([1,2,3]) → creates a 1D array [1 2 3].
- np.zeros((2,2)) → creates a 2×2 array filled with zeros.
- np.ones((2,3)) → creates a 2×3 array filled with ones.

Q10. NumPy – arange & linspace

```
import numpy as np
print(np.arange(0,10,2))     # step of 2
print(np.linspace(0,1,5))    # 5 evenly spaced numbers
```

Output:

```
[0 2 4 6 8]
[0. 0.25 0.5 0.75 1. ]
```

Explanation:

- np.arange(0,10,2) → generates numbers from 0 to 10 (exclusive) with a step of 2 → [0, 2, 4, 6, 8].
- np.linspace(0,1,5) → generates 5 evenly spaced numbers between 0 and 1 (inclusive) → [0., 0.25, 0.5, 0.75, 1.].

Q11. NumPy – Shape, ndim, size

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])
print(a.shape, a.ndim, a.size)
```

Output: (3, 2) 2 6

Explanation:

- a.shape → (3, 2) → array has 3 rows and 2 columns.
- a.ndim → 2 → it is a 2-dimensional array (matrix).
- a.size → 6 → total number of elements (3 × 2).

Q12. NumPy – dtype, itemsize, nbytes

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]], dtype=np.int32)
print(a.dtype, a.itemsize, a.nbytes)
```

Output (typical): int32 4 24

Explanation:

- a.dtype → int32 → each element is a 32-bit integer.
- a.itemsize → 4 → each element takes 4 bytes.
- a.nbytes → 24 → total memory = 6 elements × 4 bytes = 24 bytes.

Q13. NumPy – Slicing & reshape

```
import numpy as np
a = np.arange(1,7)       # [1 2 3 4 5 6]
print(a[1:4])            # [2 3 4]
print(a.reshape(2,3))    # 2x3 matrix
```

Output:

```
[2 3 4]
[[1 2 3]
 [4 5 6]]
```

Explanation:

- a = np.arange(1,7) → creates [1 2 3 4 5 6].
- a[1:4] → slices elements at indices 1, 2, 3 → [2 3 4].
- a.reshape(2,3) → reshapes array into a **2×3 matrix**.

Q14. NumPy – Elementwise ops & statistics

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
print(a + b)         # elementwise add
```

```
print(a * 2)        # scalar multiply
print(a.mean(), a.std())
```

Output:
[11 22 33 44]
[2 4 6 8]
2.5 1.118033988749895 (std ≈ 1.118)

Explanation:

- a + b → elementwise addition → [1+10, 2+20, 3+30, 4+40] = [11 22 33 44].
- a * 2 → scalar multiplication → [2, 4, 6, 8].
- a.mean() → average of [1,2,3,4] = 2.5.
- a.std() → standard deviation ≈ 1.118.

Q15. NumPy – Dot product

```
import numpy as np
a = np.array([1,2,3])
b = np.array([4,5,6])
print(np.dot(a,b))
```

Output: 32

Explanation:

- np.dot(a, b) → computes the dot product:
  1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32.

Q16. pandas – Series and DataFrame basics

```
import pandas as pd
s = pd.Series([10,20,30], index=["a","b","c"])
df = pd.DataFrame({"Name":["A","B","C"], "Score":[85,90,78]})
print(s)
print(df)
```

Output:

```
a   10
b   20
c   30
dtype: int64
  Name  Score
0   A     85
1   B     90
2   C     78
```

Explanation:

- pd.Series([10,20,30], index=["a","b","c"]) → creates a **Series** with custom index labels.
- pd.DataFrame({...}) → creates a **DataFrame** with two columns: "Name" and "Score".

Q17. pandas – Selecting columns & basic stats

```
import pandas as pd
df = pd.DataFrame({"Math":[80,90,85], "Sci":[78,92,88]})
print(df["Math"])       # column
print(df.mean(numeric_only=True))
```

Output:

```
0   80
1   90
2   85
Name: Math, dtype: int64
Math   85.0
Sci    86.0
dtype: float64
```

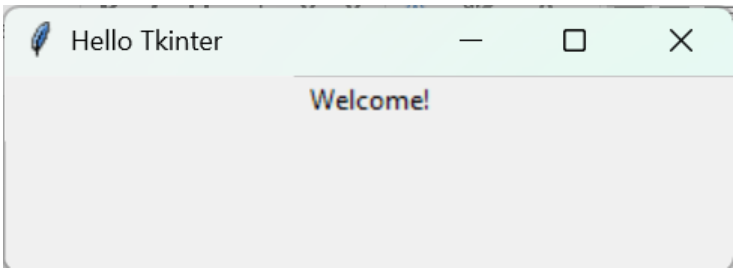
Explanation:

- df["Math"] → selects the **Math column**.
- df.mean(numeric_only=True) → computes the **mean of each numeric column**:
  - ✓ Math → (80+90+85)/3 = 85.0
  - ✓ Sci → (78+92+88)/3 = 86.0

Q18. Tkinter – Small window with a label

```
import tkinter as tk
root = tk.Tk()
root.title("Hello Tkinter")
tk.Label(root, text="Welcome!").pack()
root.mainloop()
```

Output: A small window showing "Welcome!".

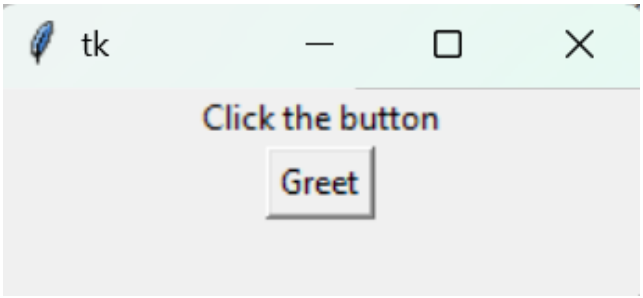


Explanation: Minimal GUI: create window, add label, start loop.

Q19. Tkinter – Button that updates label

```
import tkinter as tk
def greet():
   lbl.config(text="Hello, Python!")
root = tk.Tk()
lbl = tk.Label(root, text="Click the button"); lbl.pack()
tk.Button(root, text="Greet", command=greet).pack()
root.mainloop()
```

Output: On click, label text changes to "Hello, Python!".

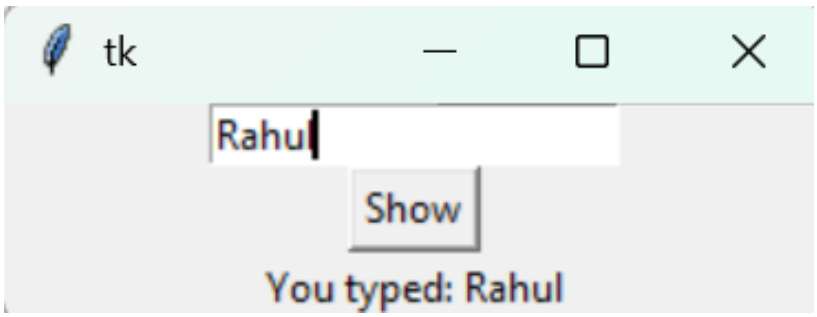


Explanation: command binds the button to a function.

Q20. Tkinter – Entry box and show text

```
import tkinter as tk
def show():
   out.config(text="You typed: " + ent.get())
root = tk.Tk()
ent = tk.Entry(root); ent.pack()
tk.Button(root, text="Show", command=show).pack()
out = tk.Label(root, text=""); out.pack()
root.mainloop()
```

Output: After typing and clicking "Show", label displays entered text.



Explanation: entry.get() reads user input; label displays it.