Programming in Python

Module 5

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

April 18, 2024

Outline

OS Module

- The os module in Python provides essential functions for interacting with the operating system.
- It is part of Python's standard utility modules, offering a portable way to use OS-dependent functionality.
- Following are some functions in OS module
  1. os.name
  2. os.getcwd()
  3. os.listdir('.')
  4. os.chdir('..')
  5. os.mkdir(path)
  6. os.rmdir(path)
  7. os.remove(path)
  8. os.rename(old, new)

OS Module

os.name

- Syntax: os.name
- Returns the name of the operating system-dependent module imported.
- Example:

```
import os
print(os.name)  # Output: 'nt' (for Windows)
```

os.getcwd()

- Syntax: os.getcwd()
- Returns the current working directory (CWD) of the file used to execute the code.
- Example:

```
import os
current_directory = os.getcwd()
print(current_directory)
```

OS Module

os.listdir('.')

- Syntax: os.listdir('.')
- Lists files and directories in the specified directory (here, the current directory).
- Example:

```
import os
contents = os.listdir('.')
print(contents)
```

os.chdir('..')

- Syntax: os.chdir('..')
- Changes the current working directory to the specified path.
- Example:

```
import os
os.chdir('..')
new_directory = os.getcwd()
print(new_directory)
```

OS Module

os.mkdir(path)

- Syntax: os.mkdir(path)
- Creates a new directory at the specified path.
- Example:

```
import os
os.mkdir('C:\\test')
```

os.rmdir(path)

- Syntax: os.rmdir(path)
- Removes the specified directory.
- Example:

```
import os
os.rmdir('C:\\test')
```

OS Module

os.remove(path)

- Syntax: os.remove(path)
- Removes the specified file.
- Example:

```
import os
os.remove('example.txt')
```

os.rename(old, new)

- Syntax: os.rename(old, new)
- Renames the file or directory from the old name to the new name.
- Example:

```
import os
os.rename('old_name.txt', 'new_name.txt')
```

Sys Module

- The sys module provides functions and variables for manipulating different parts of the Python runtime environment.
- Functions:
  1. sys.argv
  2. sys.exit
  3. sys.maxsize
  4. sys.path
  5. sys.version

Sys Module

sys.argv

- Returns a list of command-line arguments passed to a Python script.
- Example:

```
import sys
# Get the script name
script_name = sys.argv[0]

# Get the command-line arguments (excluding the script name)
arguments = sys.argv[1:]
```

sys.exit

- Causes the script to exit back to either the Python console or the command prompt.
- Used to safely exit from the program in case of an exception.
- Example:

```
import sys
sys.exit()
```

Sys Module

sys.maxsize

- Returns the largest integer a variable can take.
- Example:

```
import sys
max_integer = sys.maxsize
```

sys.path

- An environment variable that is a search path for all Python modules.
- Example:

```
import sys
module_path = sys.path
```

sys.version

- Displays a string containing the version number of the current Python interpreter.

```
import sys
python_version = sys.version
```

Introduction to NumPy

Introduction to NumPy

- NumPy is a powerful library in Python for numerical computing.
- It provides multidimensional array objects and a collection of routines for processing these arrays.
- Enables mathematical and logical operations on arrays efficiently.

Key Operations Using NumPy:

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra.
- In-built functions for linear algebra and random number generation.

ndarray Object

- The primary object in NumPy is the ndarray (N-dimensional array).
- Represents a collection of items of the same type.
- Accessed using zero-based indexing.
- Each item takes the same size block in memory.
- **Creating Arrays:**

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([(1, 2, 3), (4, 5, 6)], dtype=float)
c = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)])
```

ndarray Object

ndarray Object Parameters:

- Attributes:
  1. ndarray.ndim: Number of dimensions (axes) of the ndarray.
  2. ndarray.shape: Tuple representing the size of the ndarray in each dimension.
  3. ndarray.size: Total number of elements in the ndarray.
  4. ndarray.dtype: Data type of elements (all elements have the same type).
  5. ndarray.itemsize: Size (in bytes) of each element in the ndarray.

```
import numpy as np
a = np.array([[[1, 2, 3], [4, 3, 5]], [[3, 6, 7], [2, 1, 0]]])
print("Dimension:", a.ndim)            #Dimension: 3
print("Size:", a.shape)                #Size: (2, 2, 3)
print("Total Elements:", a.size)       #Total Elements: 12
print("Data Type:", a.dtype)           #Data Type: int32
print("Size of Each Element:", a.itemsize) #Size of Each Element: 4
```

Indexing and Slicing

- One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences
- One-Dimensional Arrays:

```
import numpy as np
A = np.arange(10)
```

- Indexing:

```
print(A[0])   # Output: 0
print(A[6])   # Output: 6
print(A[-1])  # Output: 9
```

Indexing and Slicing

- Slicing:

```
print(A[0:3])   # Output: [0 1 2]
```

- Assigning a Scalar Value to a Slice:

```
A[0:3] = 100
A[3] = 200
print(A)
# Output: [100 100 100 200 4 5 6 7 8 9]
```

- When a scalar value is assigned to a slice (e.g., A[0:3] = 100), the value is broadcasted to the entire selection.

Indexing and Slicing

- Array Slices as Views:

```
slice = A[5:9]
print(slice)   # Output: [5 6 7 8]
slice[:] = 200
print(slice)    # Output: [200 200 200 200]
```

  - Array slices are views on the original array, meaning modifications to the view are reflected in the source array.

Indexing and Slicing

Example:

```
import numpy as np

# Create a NumPy array with values ranging from 0 to 9
B = np.arange(10)

# Print the original array
print("Original Array:")
print(B)
print()

# Select elements from index 0 to 7 (exclusive) with a step of 2
print("Select every second element from index 0 to 7:")
print(B[0:8:2])
print()
```

Indexing and Slicing

```
# Select elements from index 8 to 1 with a step of -2
print("Select every second element in reverse order from index 8 to 1:")
print(B[8:0:-2])
print()

# Select elements from the beginning up to index 3 (exclusive)
print("Select the first four elements:")
print(B[:4])
print()

# Select elements from index 5 to the end of the array
print("Select elements from index 5 onwards:")
print(B[5:])
print()

# Reverse the entire array
print("Reverse the array:")
print(B[::-1])
```

Indexing and Slicing

Output:

```
Original Array:
[0 1 2 3 4 5 6 7 8 9]

Select every second element from index 0 to 7:
[0 2 4 6]

Select every second element in reverse order from index 8 to 1:
[8 6 4 2]

Select the first four elements:
[0 1 2 3]

Select elements from index 5 onwards:
[5 6 7 8 9]

Reverse the array:
[9 8 7 6 5 4 3 2 1 0]
```

The reshape() function

The reshape() function: The reshape() function allows you to change the dimensions or shape of this container without altering the data itself.

```
import numpy as np

arr = np.array([0, 1, 2, 3, 4, 5])

reshaped_arr = arr.reshape((3, 2))

#Output
[[0 1]
[2 3]
[4 5]]
```

The reshape() function

Example:

```
import numpy as np

# Creating a 3x2 array with values from 0 to 5
arr1 = np.arange(6).reshape((3, 2))
print(arr1)

#Output
[[0 1]
[2 3]
[4 5]]

# Creating another 3x2 array with values from 0
arr2 = np.arange(6).reshape((3, 2))   to 5
print(arr2)

#Output
[[0 1]
[2 3]
[4 5]]
```

The reshape() function

Example cont...

```
# Adding the first row of arr2 to arr1
arr3 = arr1 + arr2[0].reshape((1, 2))

print(arr3)

#Output
[[0 2]
[2 4]
[4 6]]
```

Arithmetic Operations with NumPy Array:

- Element-Wise Operations:
  - Arithmetic operations with NumPy arrays perform element-wise operations.
  - Operators are applied only between corresponding elements.
  - Operations require arrays to have the same structure and dimensions.
  - Basic Operations with Scalars:

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
b = a + 1
c = 2 ** a
```

  - Output:

```
[2 3 4 5 6]
[ 2 4 8 16 32]
```

Matrix Addition

```
import numpy as np

# Define matrices A and B
A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])

# Perform matrix addition (element-wise)
C = A + B

# Print the result
print("Matrix Addition:")
print(C)
# Output:
# [[11  1]
#  [ 8  0]]
```

Matrix Subtraction

```
import numpy as np

# Define matrices A and B
A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])

# Perform matrix subtraction (element-wise)
C = A - B

# Print the result
print("Matrix Subtraction:")
print(C)
# Output:
# [[ -7   7]
#  [  2 -12]]
```

Element-wise Matrix Multiplication (Hadamard product)

```
import numpy as np

# Define matrix A
A = np.array([[2, 4], [5, -6]])

# Perform element-wise matrix multiplication (Hadamard product)
D = A * A

# Print the result
print("Element-wise Matrix Multiplication:")
print(D)
# Output:
# [[ 4 16]
#  [25 36]]
```

Scalar Multiplication

```
import numpy as np

# Define matrix A
A = np.array([[1, 2], [3, 4]])

# Perform scalar multiplication
E = A * 2

# Print the result
print("Scalar Multiplication:")
print(E)
# Output:
# [[2 4]
#  [6 8]]
```

Matrix Operations

Matrix Transpose:

```
import numpy as np

# Define matrix A
A = np.array([[2, 4], [5, -6]])

# Perform matrix transpose
F = A.T  # or np.transpose(A)

# Print the result
print("Matrix Transpose:")
print(F)
# Output:
# [[ 2  5]
#  [ 4 -6]]
```

Matrix Inversion

- Matrix inversion is a mathematical operation that finds another matrix, denoted as A^{-1} , such that when multiplied with the original matrix A, it results in the identity matrix I. Not all matrices have inverses, and a square matrix that lacks an inverse is referred to as singular.

Matrix Operations

Matrix Inversion

```
from numpy import array
from numpy.linalg import inv
# Define a 2x2 matrix
A = array([[1.0, 2.0], [3.0, 4.0]])

# Print the original matrix
print("Original Matrix A:")
print(A)

# Invert the matrix
B = inv(A)
print("\nInverted Matrix B:")
print(B)

# Multiply A and B to get the identity matrix
I = A.dot(B)
print("\nIdentity Matrix I:")
print(I)
```

Matrix Operations

Output:

```
Original Matrix A:
[[1. 2.]
[3. 4.]]

Inverted Matrix B:
[[-2.   1. ]
[ 1.5 -0.5]]

Identity Matrix I:
[[1. 0.]
[0. 1.]]
```

Matrix Operations

Matrix Trace

- The trace of a square matrix is the sum of its diagonal elements from the top-left to the bottom-right.

```
from numpy import array
from numpy import trace

# Define a 3x3 matrix
A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Print the original matrix
print("Original Matrix A:")
print(A)

# Calculate the trace of the matrix
B = trace(A)
print("\nTrace of the Matrix:")
print(B)
```

Output

```
Original Matrix A:
[[1 2 3]
[4 5 6]
[7 8 9]]

Trace of the Matrix:
15
```

In this example, the original matrix A is a 3×3 matrix. The trace is calculated by summing the diagonal elements $(1 + 5 + 9)$, resulting in a trace value of 15. The trace is a useful measure often employed in various mathematical and engineering applications.

Matrix Operations

Determinant

- The determinant of a square matrix is a scalar value that provides information about the volume scaling factor of the linear transformation described by the matrix.

```
from numpy import array
from numpy.linalg import det

# Define a 3x3 matrix
A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Print the original matrix
print("Original Matrix A:")
print(A)

# Calculate the determinant of the matrix
B = det(A)
print("\nDeterminant of the Matrix:")
print(B)
```

Matrix Operations

Output:

```
Original Matrix A:
[[1 2 3]
[4 5 6]
[7 8 9]]

Determinant of the Matrix:
0.0
```

In this example, the original matrix A is a 3×3 matrix. The determinant is calculated, and in this case, it is 0.0. When the determinant is zero, it indicates that the matrix is singular, and it doesn't have a unique inverse. Determinants play a crucial role in various mathematical and geometric concepts.

Matrix Operations

Matrix-Matrix Multiplication

- Matrix multiplication, also known as the matrix dot product, is a more complex operation compared to element-wise operations.
- It involves a specific rule, and not all matrices can be multiplied together.

Rule for Matrix Multiplication:

- The number of columns (n) in the first matrix (A) must equal the number of rows (m) in the second matrix (B).

Matrix-Matrix Multiplication

```
from numpy import array

# Define the first matrix
A = array([[1, 2], [3, 4], [5, 6]])
print("Matrix A:")
print(A)

# Define the second matrix
B = array([[1, 2], [3, 4]])
print("\nMatrix B:")
print(B)

# Multiply matrices A and B
C = A.dot(B)
print("\nResult of Matrix Multiplication (C = A.dot(B)):")
print(C)
```

Matrix Operations

Output:

```
Matrix A:
[[1 2]
[3 4]
[5 6]]

Matrix B:
[[1 2]
[3 4]]

Result of Matrix Multiplication (C = A.dot(B)):
[[ 7 10]
[15 22]
[23 34]]
```

Random numbers

- Randomness is crucial in computing, but computers follow definitive instructions.
- True randomness is challenging; instead, we often use pseudo-random numbers generated by algorithms.
- NumPy's random module provides tools for pseudo-random number generation.

Random Integer Generation with randint()

- The randint() method in NumPy's random module is used to generate random integers within a specified range.
- Syntax:

```
numpy.random.randint(low, high=None, size=None, dtype='l')
```

  - low: The lowest (inclusive) integer in the range.
  - high: The highest (exclusive) integer in the range. If not specified, it will generate integers from 0 to low.
  - size: The shape of the output. If None, a single random integer is returned.
  - dtype: Data type of the output. Default is 'l' (long).

Random numbers

randint() Method:

```
import numpy as np
# Generate a random integer between 0 and 100
x = np.random.randint(100)
print(x)
# Output: 64

# Generate an array of 5 random integers between 0 and 100
x = np.random.randint(100, size=5)
print(x)
# Output: [25 62 24 81 39]
# Generate a 2-D array with 3 rows, each containing 5 random integers
    from 0 to 100
x = np.random.randint(100, size=(3, 5))
print(x)
# Output:
# [[ 2 96 40 43 85]
#  [81 81 4 48 29]
#  [80 31 6 10 24]]
```

Random Float Generation with rand()

- The rand() method in NumPy's random module is used to generate random float values from a uniform distribution over the interval[0, 1).
- Syntax:

```
numpy.random.rand(d0, d1, ..., dn)
```

  - d0, d1, ..., dn: Optional. The dimensions of the returned array. If no argument is provided, a single random float is returned.

Random numbers

rand() Method:

```
import numpy as np
# Generate a random float between 0 and 1
x = np.random.rand()
print(x)
# Output: 0.2733166576024767

# Generate an array of 10 random floats between 0 and 1
x = np.random.rand(10)
print(x)
# Output: [0.82536563 0.46789636 0.28863107 0.83941914 0.24424812
    0.25816291 0.72567413 0.80770073 0.32845661 0.34451507]

# Generate a 2-D array with size (3, 5)
x = np.random.rand(3, 5)
print(x)
# Output:
# [[0.16220086 0.80935717 0.97331357 0.60975199 0.48542906]
#  [0.68311884 0.27623475 0.73447814 0.29257476 0.27329666]
#  [0.62625815 0.0069779  0.21403868 0.49191027 0.4116709 ]]
```

Random Value Selection with choice()

- The choice() method in NumPy's random module is used to generate random values by selecting elements randomly from a given array or list.
- Syntax:

```
numpy.random.choice(a, size=None, replace=True, p=None)
```

  - a: The array or list from which random samples will be drawn.
  - size: Optional. The shape of the output. If not provided, a single random element is returned.
  - replace: Optional. Whether the sampling is done with or without replacement. Default is True.
  - p: Optional. The probabilities associated with each entry in the array. If not provided, the sample is equally likely for each entry.

Random numbers

choice() Method:

```
import numpy as np

# Get a random value from an array of values
x = np.random.choice([3, 5, 6, 7, 9, 2])
print(x)
# Output: 3

# Generate a 2-D array with size (3, 5) from the given values
x = np.random.choice([3, 5, 6, 7, 9, 2], size=(3, 5))
print(x)
# Output:
# [[3 2 5 2 6]
#  [5 9 3 6 9]
#  [5 6 9 3 3]]
```

Random Permutations: Shuffling Arrays with shuffle()

- In NumPy's random module, the shuffle() method is used for shuffling the elements of an array, effectively changing their arrangement in-place.
- Syntax:

```
numpy.random.shuffle(x)
```

- x: The array or list to be shuffled.

Random numbers

Example:

```
import numpy as np

# Create an array
x = np.array([1, 2, 3, 4, 5])

# Shuffle the array in-place
np.random.shuffle(x)
print(x)
# Output: [4 1 3 5 2]
```

The shuffle() method modifies the original array directly, and it rearranges the elements randomly. This is useful when you want to create a random order of elements for various applications like shuffling a deck of cards, randomizing training data for machine learning, etc.

Generating Permutation of Arrays with permutation()

- The permutation() method in NumPy's random module generates a random permutation of a sequence or returns a permuted range.
- Syntax:

```
numpy.random.permutation(x)
```

- x: The array or list to be permuted.

Random numbers

Example:

```
import numpy as np

# Create an array
x = np.array([1, 2, 3, 4, 5])

# Generate a permuted array (leaving the original array unchanged)
y = np.random.permutation(x)
print(y)
# Output: [3 1 5 2 4]
```

Note: Unlike shuffle(), the permutation() method does not modify the original array; instead, it returns a permuted version of the array. This is useful when you want to keep the original order intact while creating a random arrangement for a specific use case.

Random numbers

Random Permutations:

```
import numpy as np

# Shuffle an array in-place
x = np.array([1, 2, 3, 4, 5])
np.random.shuffle(x)
print(x)
# Output: [4 1 3 5 2]

# Generate a permutation of an array without changing the original array
x = np.array([1, 2, 3, 4, 5])
y = np.random.permutation(x)
print(y)
# Output: [3 1 5 2 4]
```

Matplotlib

Matplotlib

- Matplotlib is a widely used Python library for data visualization.
- It facilitates the creation of 2D plots from data stored in arrays.
- Being cross-platform, Matplotlib is compatible with various operating systems.
- It is typically used in conjunction with NumPy for numerical operations.

Importing Matplotlib

- The pyplot module is the primary interface for creating plots.

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

The plot() function

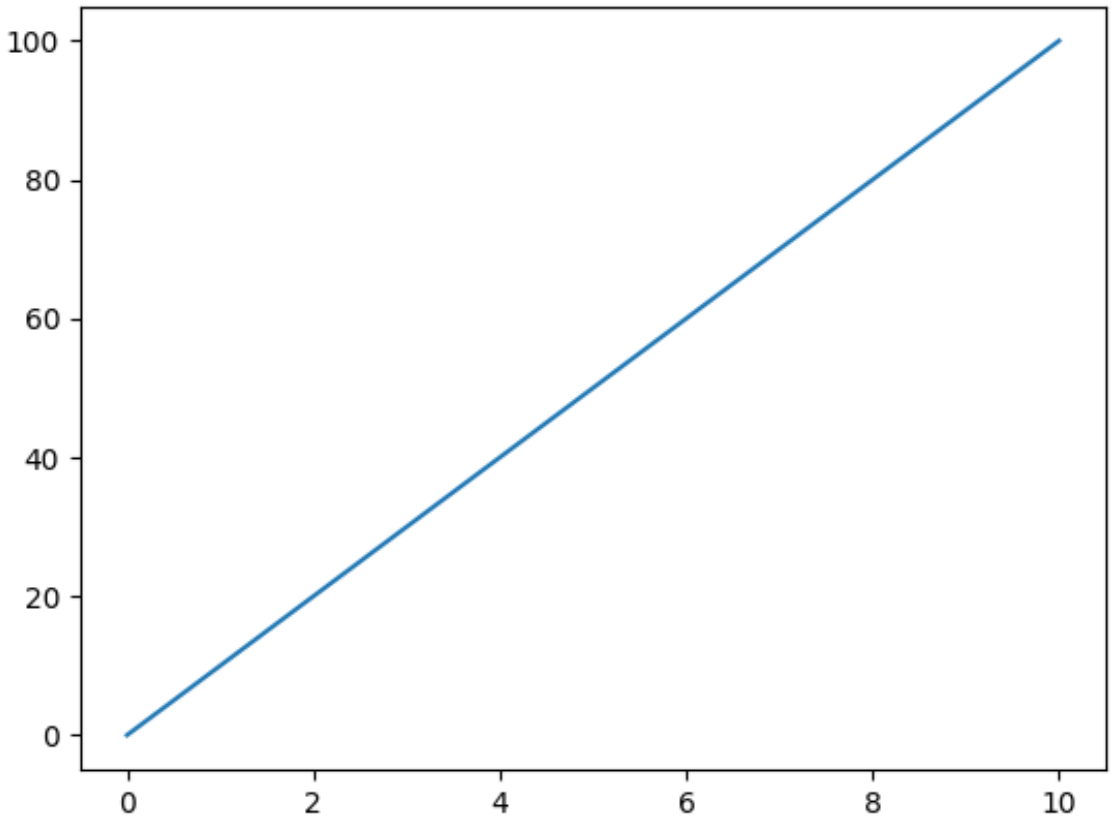
- The plot() function is used to draw points (markers) in a diagram.
- By default, the plot() function draws a line from point to point.
- The function takes parameters for specifying points in the diagram.
  - Parameter 1 is an array containing the points on the x-axis.
  - Parameter 2 is an array containing the points on the y-axis.

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

xpoints=np.array([0,10])
ypoints=np.array([0,100])

plt.plot(xpoints,ypoints)
plt.show()
```

Matplotlib



Matplotlib

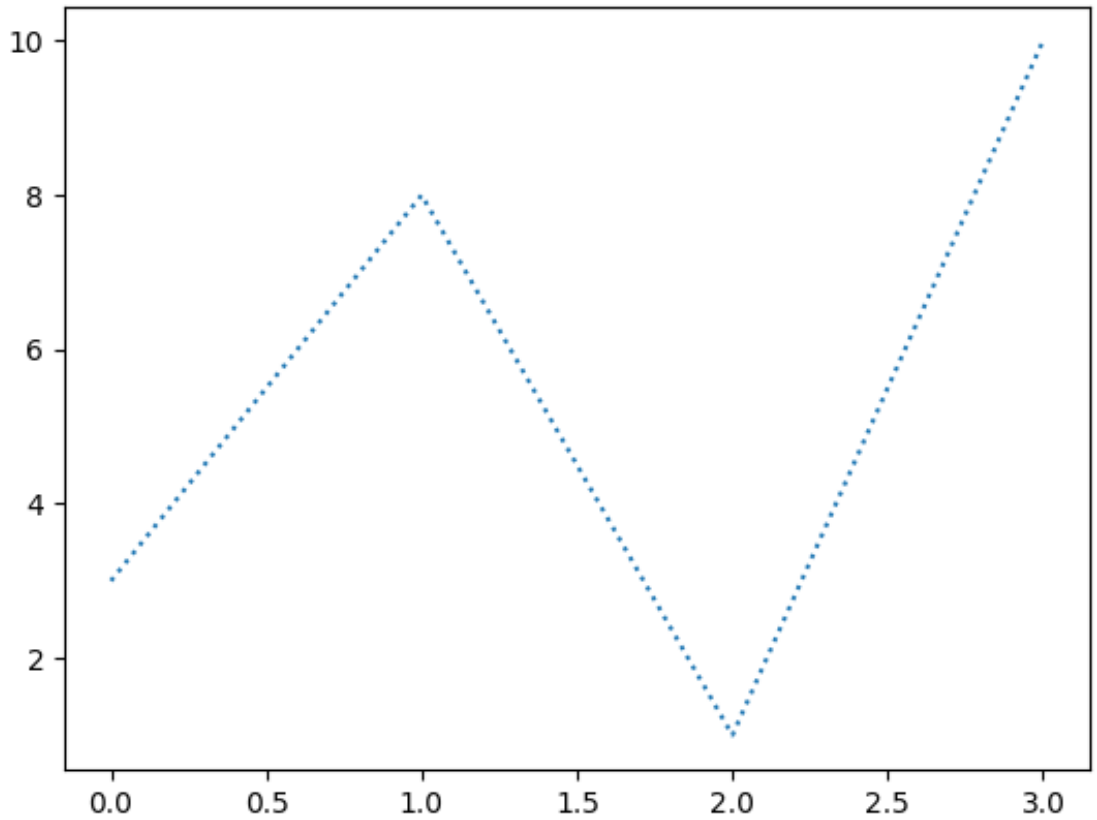
- When you provide only a single array of y-values to plt.plot(), matplotlib assumes that the x-values are the indices of the array. In other words, the x-values are implicitly taken as range(len(ypoints)).
- So, for the ypoints array [3, 8, 1, 10], the x-values would be [0, 1, 2, 3] (indices of the array).
- This results in the points (0, 3), (1, 8), (2, 1), and (3, 10) being plotted.

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

ypoints=np.array([3,8,1,10])

plt.plot(ypoints,linestyle='dotted')
plt.show()
```

Matplotlib



Matplotlib

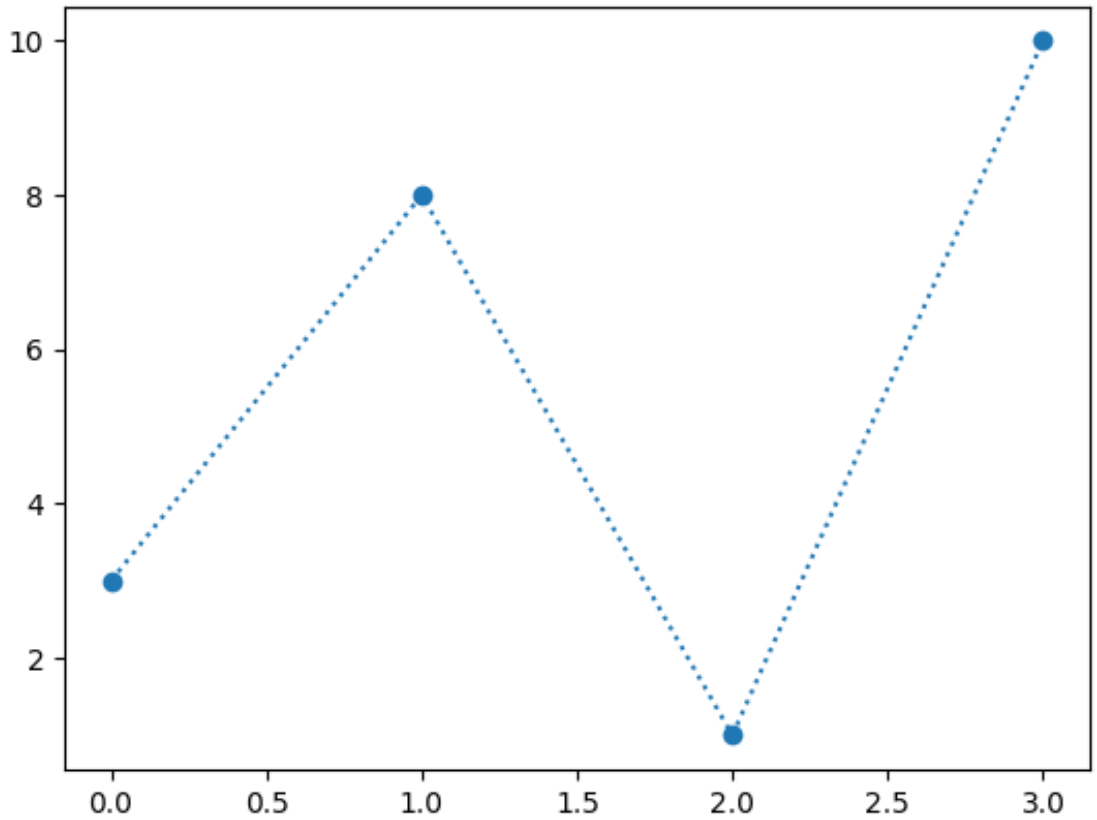
- You can use the marker parameter in plt.plot() to specify a marker style for each point.

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle='dotted', marker='o')
plt.show()
```

Matplotlib



Matplotlib

Step-by-step instructions to plot a simple sine wave using Matplotlib:

1. Import Matplotlib
2. Import NumPy and Create Data
3. Calculate Sine Values
4. Plot the Sine Wave.
5. Customize the Plot
6. Display the Plot

Importing Matplotlib

- The pyplot module is the primary interface for creating plots.

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

Importing NumPy and Setting up Data

```
import numpy as np
import math

x = np.arange(0, math.pi * 2, 0.05)
```

- NumPy is often used with Matplotlib for handling numerical data.
- Here, np.arange() generates an array of numbers from 0 to 2π with a step of 0.05.

Calculating Sine Values

```
y = np.sin(x)
```

- The np.sin() function is applied to the array x to obtain corresponding sine values.

Plotting with Matplotlib

- Using the plot() Function

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

  - The plot() function is used to plot the values from the arrays x and y.

- Customizing the Plot

```
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('Sine Wave')
```

  - xlabel(), ylabel(), and title() are used to set labels for the x and y axes, and a title for the plot, respectively.

Displaying the Plot

- Invoking the Plot Viewer

```
plt.show()
```

  - The show() function is called to display the plot.

Matplotlib

```
from matplotlib import pyplot as plt
import numpy as np
import math

x = np.arange(0, math.pi * 2, 0.05)
y = np.sin(x)

plt.plot(x, y)
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('Sine Wave')
plt.show()
```

In this example, we import the necessary libraries, generate data for the sine wave, plot it, and customize the plot before displaying it.

Matplotlib

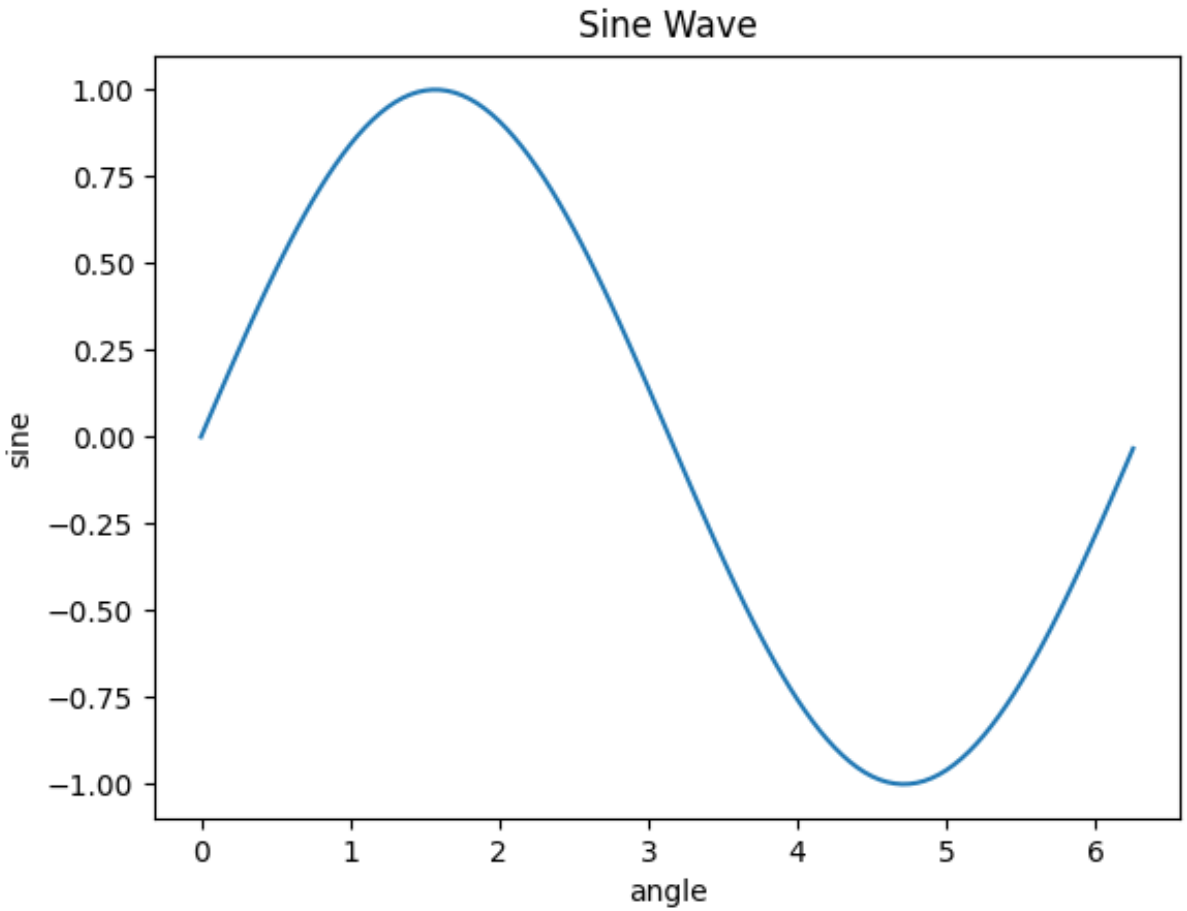


Figure: sine wave

Introduction to PyLab

- PyLab was a Python package that aimed to make plotting and numerical computations similar to what users experienced in MATLAB.
- It provided a convenient way to combine the functionality of two powerful Python libraries, NumPy and Matplotlib, into a single namespace.

Basic Plotting with PyLab

- Importing Libraries

```
from numpy import *
from pylab import *
```

  - Import NumPy for numerical operations and PyLab for plotting.

- Creating Data

```
x = linspace(-3, 3, 30)
y = x**2
```

  - Use NumPy's linspace to generate 30 evenly spaced points between -3 and 3.
  - Calculate y as the square of x.

- Plotting and Displaying

```
plot(x, y)
show()
```

  - Use plot(x, y) to create a plot.
  - Use show() to display the plot.

Example: Basic Plot

```
from numpy import *
from pylab import *

# Create data
x = linspace(-3, 3, 30)
y = x**2

# Plot and display
plot(x, y)
show()
```

This example generates a basic plot of a quadratic function.

Matplotlib

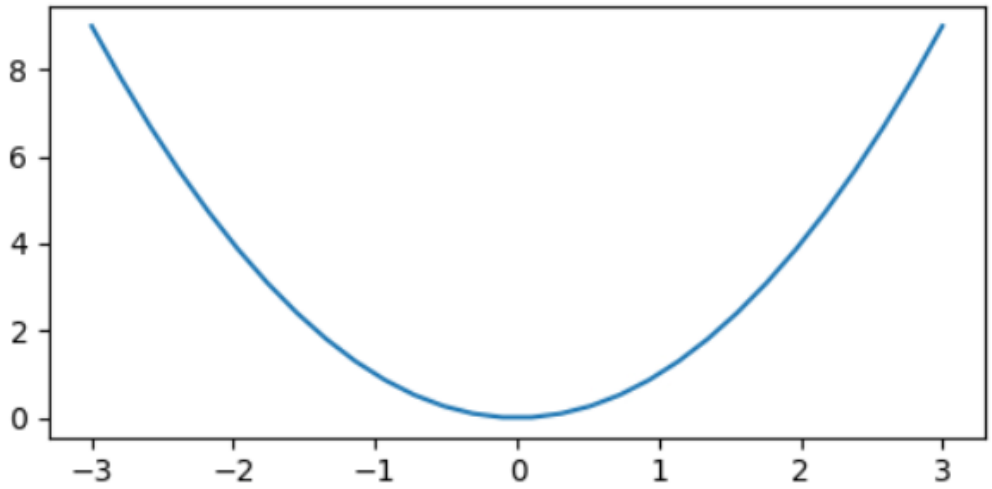


Figure: Basic plot of a quadratic function

Multiple Plots Example

```
import matplotlib.pyplot as plt
import numpy as np
import math  # Import the math module

x = np.arange(0, math.pi*2, 0.05)

# Plot sin(x) in blue
plt.plot(x, np.sin(x))

# Plot cos(x) in red with a solid line
plt.plot(x, np.cos(x), 'r-')

# Plot -sin(x) in green with a dashed line
plt.plot(x, -np.sin(x), 'g--')

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

This example shows multiple plots on the same graph.

Matplotlib

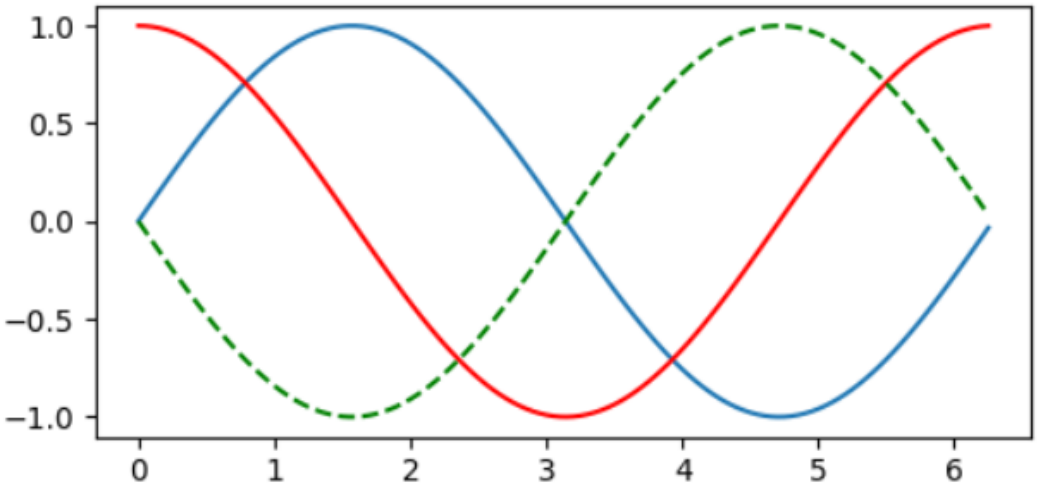


Figure: Multiple Plots Example

Color code

```
'b': blue
'g': green
'r': red
'c': cyan
'm': magenta
'y': yellow
'k': black
'w': white
```

Point Markers:

```
'.': Point
',': Pixel
```

Line Markers

```
'-': Solid line
':': Dotted line
'--': Dashed line
'-.': Dash-dot line
```

Marker Styles

```
'o': Circle
's': Square
'D': Diamond
'^': Upward triangle
'v': Downward triangle
'<': Leftward triangle
'>': Rightward triangle
'p': Pentagon
'P': Larger pentagon
'*': Star
'H': Larger hexagon
'h': Hexagon
'x': Cross
'+': Plus
```

Matplotlib

Example code that uses various markers in a Matplotlib plot

```
import matplotlib.pyplot as plt
import numpy as np
# Data
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
# Plot with different markers
plt.plot(x, y, marker='o', label='Circle')
plt.plot(x, -y, marker='s', label='Square')
plt.plot(x, np.cos(x), marker='^', label='Upward Triangle')
plt.plot(x, -np.cos(x), marker='v', label='Downward Triangle')
plt.plot(x, np.sin(2*x), marker='D', label='Diamond')
# Set labels and legend
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()
# Display the plot
plt.show()
```

Matplotlib

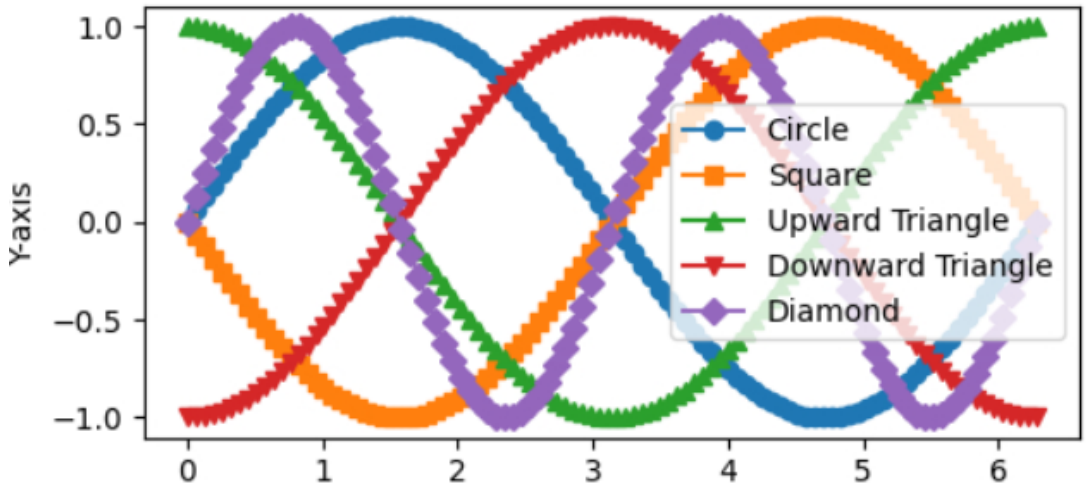


Figure: Example code that uses various markers in a Matplotlib plot

Adding Grids and Legend to the Plot

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

# Generate data
x = np.arange(0, math.pi*2, 0.05)

# Plot sin(x) in blue
plt.plot(x, np.sin(x), label='sin')

# Plot cos(x) in red with a solid line
plt.plot(x, np.cos(x), 'r-', label='cos')

# Plot -sin(x) in green with a dashed line
plt.plot(x, -np.sin(x), 'g--', label='-sin')

# Add a grid
plt.grid(True)
```

Matplotlib

```
# Set title and labels
plt.title('Waves')
plt.xlabel('x')
plt.ylabel('sin, cos, -sin')

# Display the legend in the upper right corner
plt.legend(loc='upper right')

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

This code creates a plot with sine, cosine, and negative sine waves, applies a grid, sets a title, and adds labels to the x and y axes. The legend is placed in the upper right corner to identify each curve

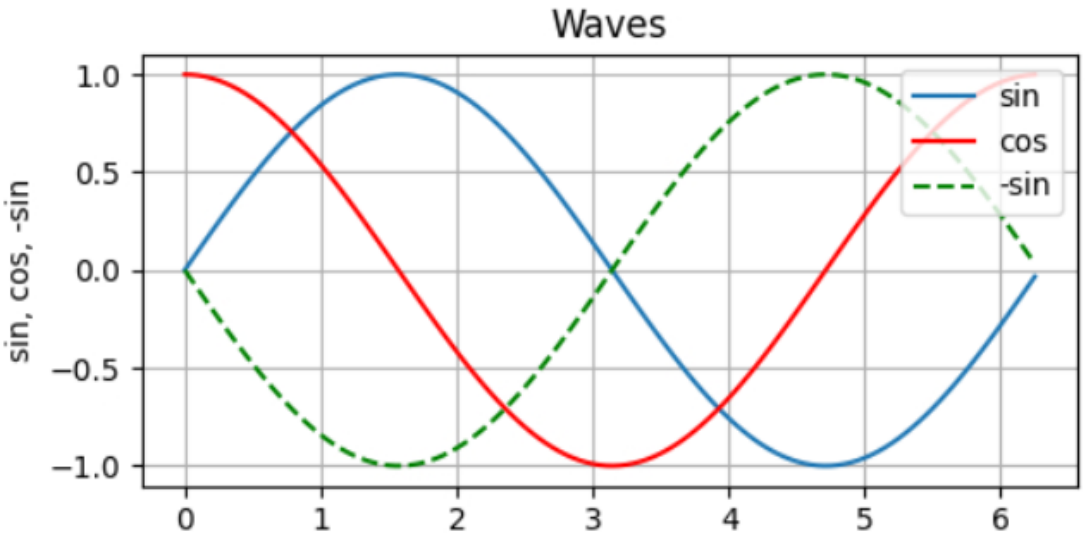


Figure: Adding Grids and Legend to the Plot

Creating a bar plot

```
from matplotlib import pyplot as plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4, 2]

# Plotting the bar chart
plt.bar(x, y)

# Adding labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Bar Plot Example')

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

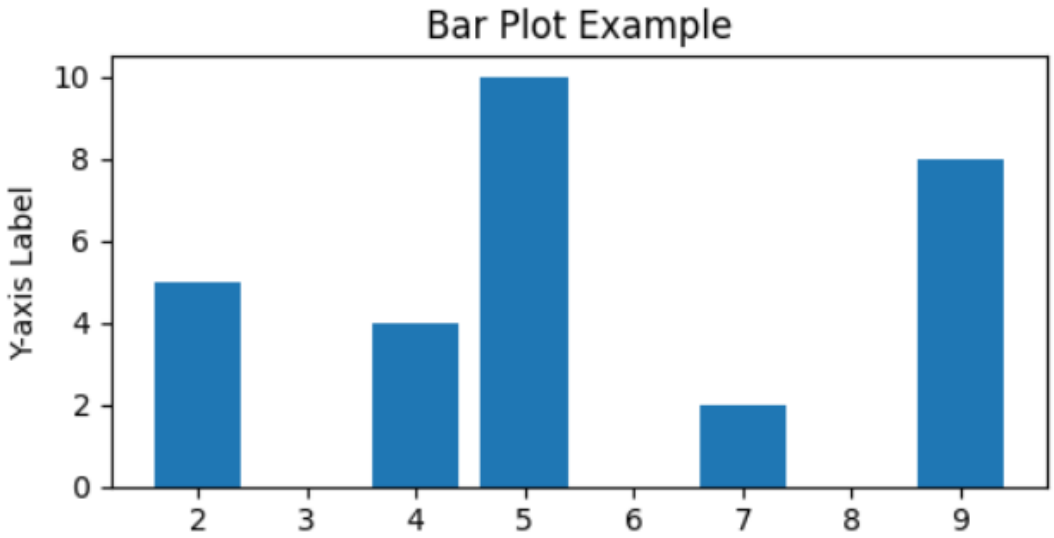


Figure: Creating a bar plot

Creating a histogram

```
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7, 5, 5, 5, 4, 9, 9, 9, 9, 9, 9, 9, 9, 9]

# Plotting the histogram
plt.hist(x, bins='auto', color='blue', alpha=0.7, rwidth=0.85)

# Adding labels and title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram Example')

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

Matplotlib

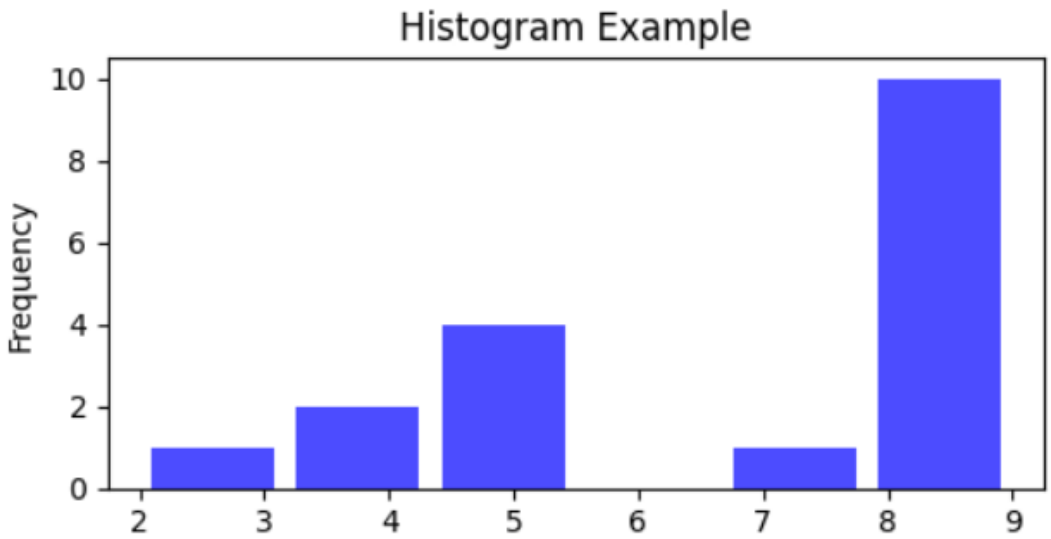


Figure: Creating a histogram

Scatter Plot

```
from matplotlib import pyplot as plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4, 2]

# Plotting the scatter plot
plt.scatter(x, y)

# Adding labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Scatter Plot Example')

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

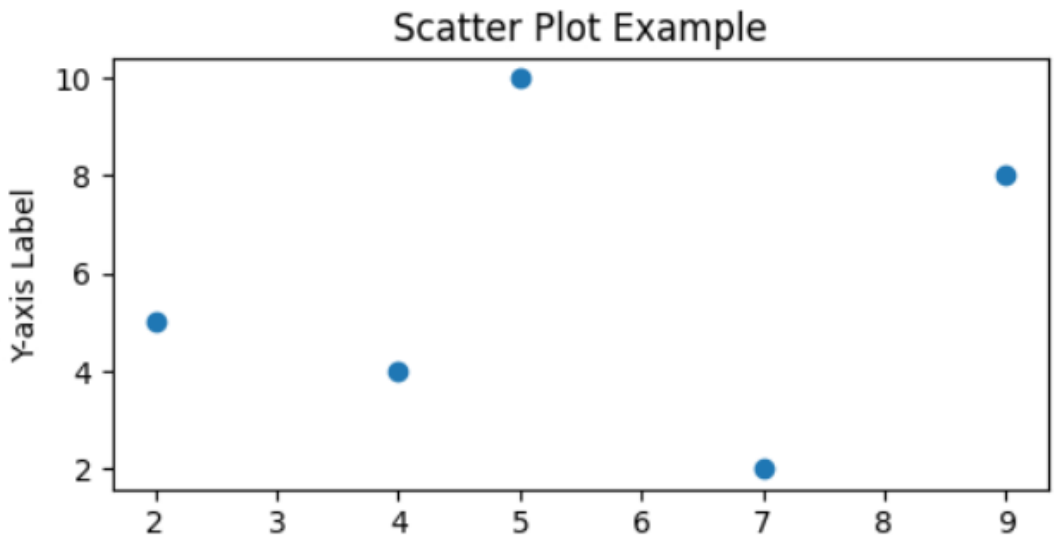


Figure: Scatter Plot

Stem plot

```
from matplotlib import pyplot as plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4, 2]

# Plotting the stem plot
plt.stem(x, y)

# Adding labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Stem Plot Example')

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

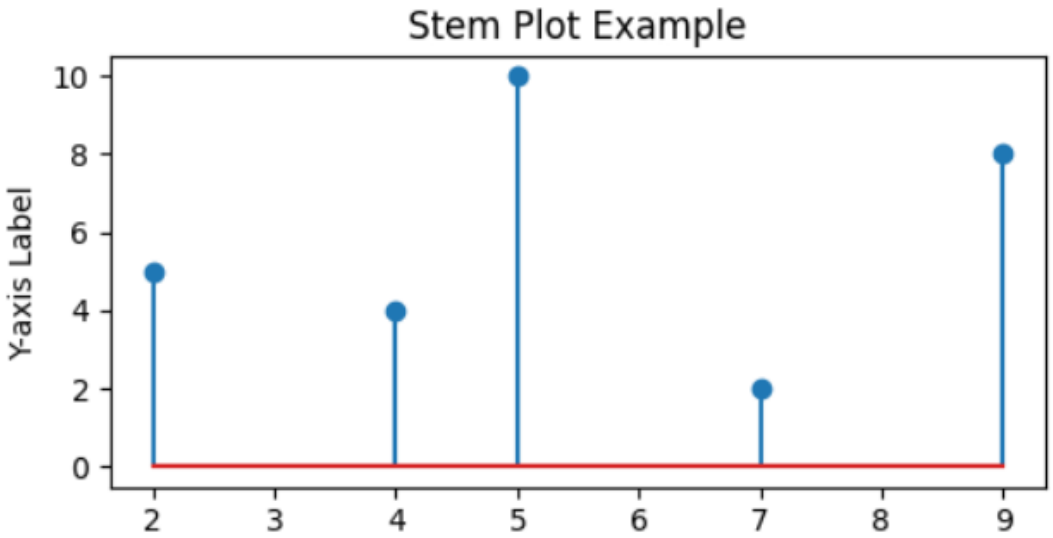


Figure: Stem plot

Pie Plot

```
from pylab import *

data = [20, 30, 10, 50]
labels = ['Label 1', 'Label 2', 'Label 3', 'Label 4']

# Plotting the pie chart with labels
pie(data, labels=labels)

# Showing the plot
show()
```

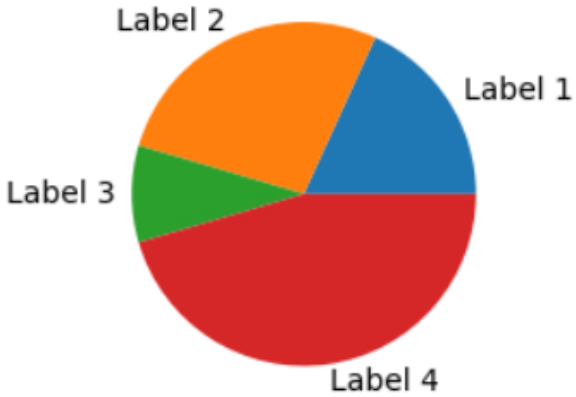


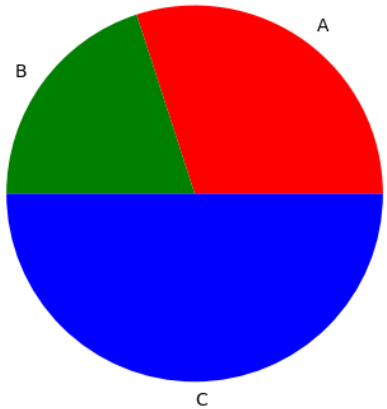
Figure: Pie Plot

Matplotlib

- It's possible to customize the colors of the slices in a pie chart using the colors parameter in Matplotlib's pie() function

```
import matplotlib.pyplot as plt
data = [30, 20, 50]
labels = ['A', 'B', 'C']
colors = ['red', 'green', 'blue']

plt.pie(data, labels=labels, colors=colors)
plt.show()
```



Subplots with in the same plot

- The subplot() function in Matplotlib is used to create subplots within a figure. It takes three arguments:

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

  - nrows: The number of rows of subplots.
  - ncols: The number of columns of subplots.
  - index: The index of the current subplot, starting from 1 and increasing row-wise and then column-wise.

Subplots with in the same plot

```
from pylab import *
import numpy as np
import math

x = np.arange(0, math.pi*2, 0.05)

subplot(2, 2, 1)
plot(x, np.sin(x), label='sin')
xlabel('x')
ylabel('sin')
legend(loc='upper right')
grid(True)

subplot(2, 2, 2)
plot(x, np.cos(x), 'r-', label='cos')
xlabel('x')
ylabel('cos')
legend(loc='upper right')
grid(True)
```

Matplotlib

```
subplot(2, 2, 3)
xlabel('x')
ylabel('-sin')
plot(x, -np.sin(x), 'g--', label='-sin')
legend(loc='upper right')
grid(True)

subplot(2, 2, 4)
xlabel('x')
ylabel('tan')
plot(x, np.tan(x), 'y-', label='tan')
legend(loc='upper right')
grid(True)

show()
```

Matplotlib

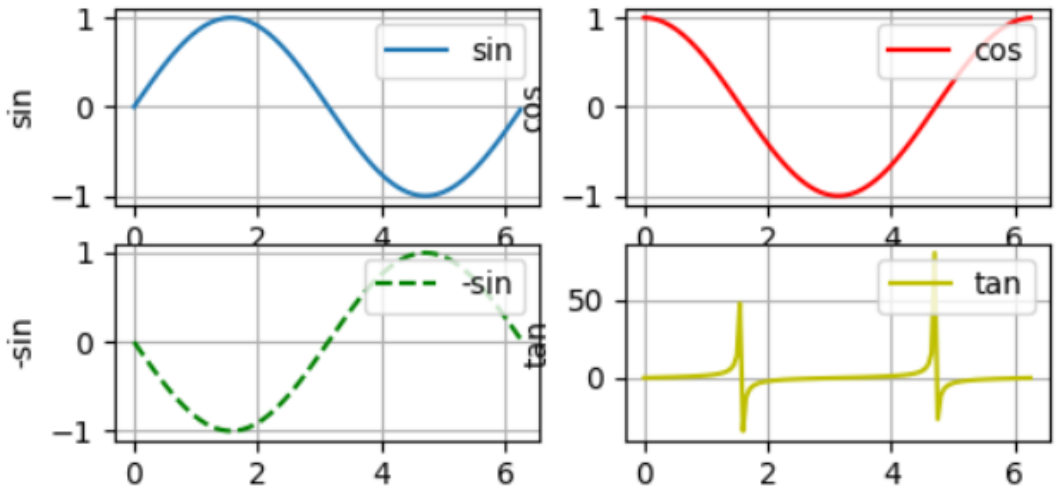


Figure: Subplots with in the same plot

Ticks in Plot

- Ticks in a plot represent specific points on the coordinate axis, and they can be either numbers or strings.
- By default, Matplotlib provides ticks on the axes when we create a graph. While the default ticks are suitable for many situations, there are cases where customization is needed.
- In those instances, we can modify the ticks to better suit our specific requirements.

Ticks in Plot

- Ticks in a plot are the marks or indicators along the axis that represent specific points or intervals.
- They help in understanding the scale of the plot and interpreting the values being displayed.
- In simpler terms, ticks are the little marks you see along the x-axis and y-axis of a graph that show you where specific values are located.

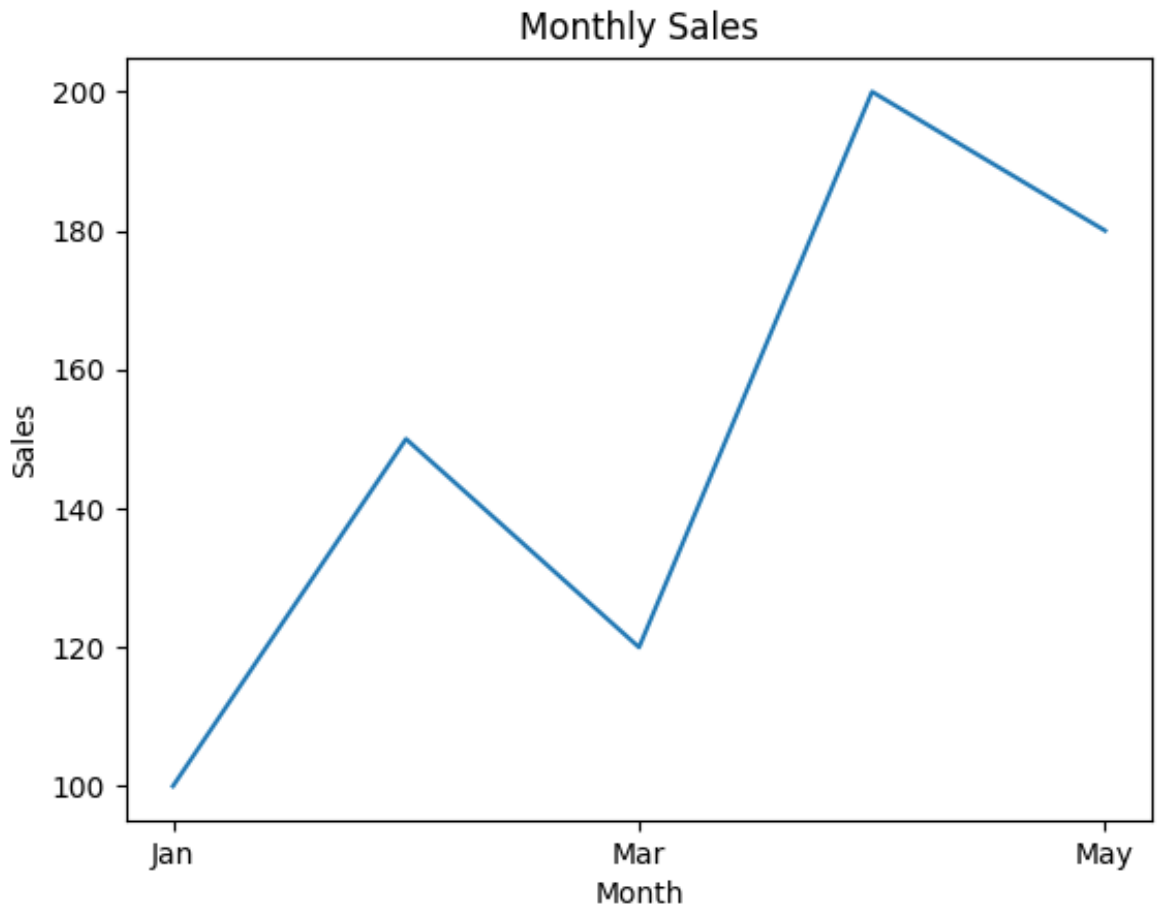
The following program shows the default ticks and customized ticks

```
import matplotlib.pyplot as plt

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May']
sales = [100, 150, 120, 200, 180]

plt.plot(months, sales)
plt.xticks(months[::2])  # Show ticks for every other month
plt.xlabel('Month')
plt.ylabel('Sales')
plt.title('Monthly Sales')
plt.show()
```

Matplotlib



Matplotlib

The following program shows the default ticks and customized ticks

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

# Values for x and y axes
x = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
y = [1, 4, 3, 2, 7, 6, 9, 8, 10, 5]

# Figure 1: Default ticks (blue)
plt.figure(1)
plt.plot(x, y, 'b')
plt.xlabel('x')
plt.ylabel('y')

# Figure 2: Customized ticks (red)
plt.figure(2)
plt.plot(x, y, 'r')
plt.xlabel('x')
plt.ylabel('y')
```

Matplotlib

```
# Customizing ticks on the x and y axes
plt.xticks(np.arange(0, 51, 5))  # x-axis ticks from 0 to 50 with
    intervals of 5
plt.yticks(np.arange(0, 11, 1))  # y-axis ticks from 0 to 10 with
    intervals of 1

# Customizing y-axis tick labels to appear in red and rotated by 45
    degrees
plt.tick_params(axis='y', colors='red', rotation=45)

# Displaying the figures
plt.show()
```

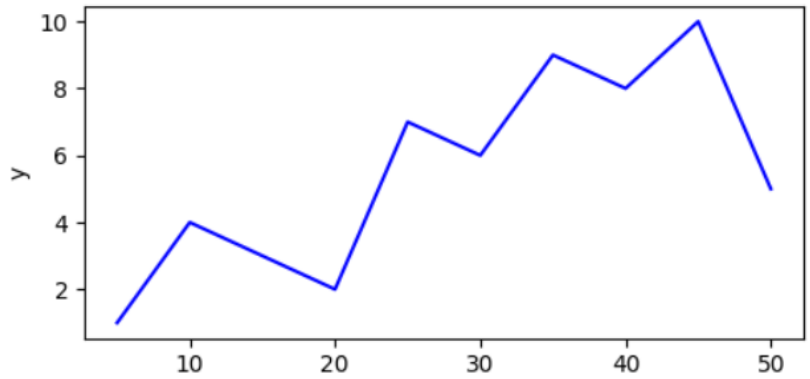


Figure: Default ticks (blue)

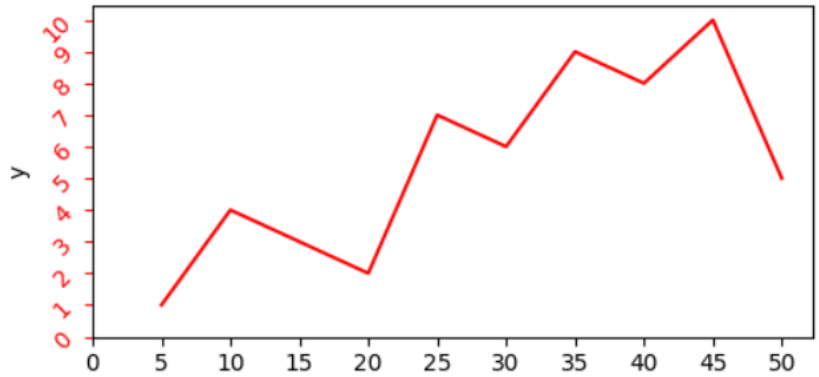


Figure: Customized ticks (red)

Labels in Matplotlib

- In Matplotlib, labels are often associated with axes, titles, or data points.
- **Axis Labels:**

```
import matplotlib.pyplot as plt

plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.show()
```

This sets labels for the X and Y axes in a plot.

- **Title Label:**

```
import matplotlib.pyplot as plt

plt.title("Title of the Plot")
plt.show()
```

Sets a title for the entire plot.

- **Data Point Labels:**

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 25, 30]

plt.scatter(x, y, label="Data Points")
plt.legend()
plt.show()
```

Adds a label to a set of data points in a scatter plot.

Legends in Matplotlib

- Legends are used to explain the meaning of different elements in the plot, such as lines or markers.
- **Basic Legend:**

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y1 = [10, 20, 15, 25, 30]
y2 = [5, 15, 10, 20, 25]

plt.plot(x, y1, label="Line 1")
plt.plot(x, y2, label="Line 2")
plt.legend()
plt.show()
```

Creates a legend for multiple lines in a line plot.

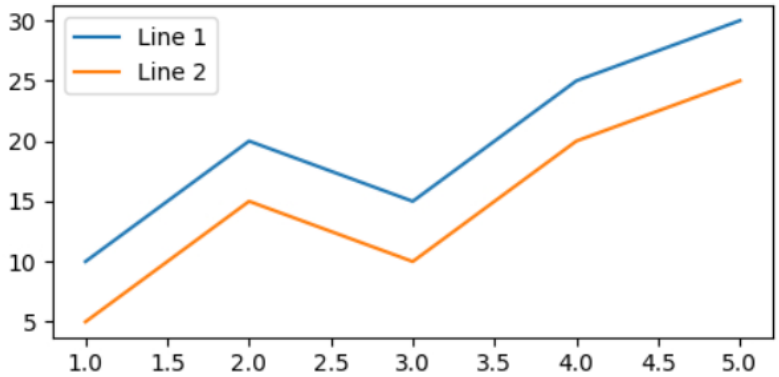


Figure: Basic Legend

- **Customizing Legend Location:**

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 25, 30]

plt.plot(x, y, label="Data Points")
plt.legend(loc="upper right")
plt.show()
```

Places the legend in the upper-right corner of the plot.

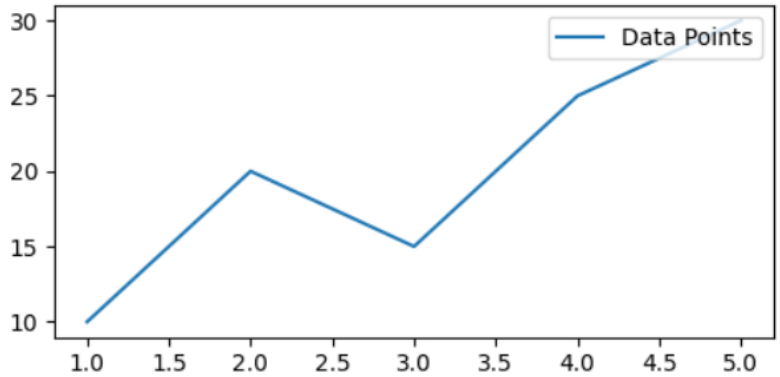


Figure: Customizing Legend Location

Tick Parameters

Parameter	Description	Example
axis	Specifies the axis to apply the changes ('x', 'y', 'both').	`plt.tick_params(axis='x', ...)`
reset	If `True`, set all parameters to default values.	`plt.tick_params(reset=True)`
length	Length of tick marks in points.	`plt.tick_params(length=10.0)`
width	Width of tick marks in points.	`plt.tick_params(width=2.0)`
rotation	Rotation angle of tick labels in degrees.	`plt.tick_params(rotation=45.0)`
color	Color of tick marks and labels.	`plt.tick_params(color='red')`
pad	Distance between tick marks and labels.	`plt.tick_params(pad=5.0)`
direction	Direction of tick marks ('in', 'out', 'inout').	`plt.tick_params(direction='in')`
flat	If `True`, applies changes to all axes.	`plt.tick_params(axis='both', flat=True)`
False	Used to specify boolean values where required.	`plt.tick_params(reset=False)`

Working with CSV Files

- CSV Format Overview:
  - CSV (Comma-Separated Values) is a data format where fields/columns are separated by commas, and records/rows are terminated by newlines.
  - No specific character encoding or byte order is required. Records end at line terminators.
  - All records must have the same number of fields in the same order.
  - Data within fields is treated as a sequence of characters.
- CSV File Characteristics
  - One line for each record.
  - Fields are separated by commas.
  - Space-characters adjacent to commas are ignored.
  - Fields with built-in commas are enclosed in double quote characters.
  - Fields with double quote characters must be surrounded by double quotes. Each in-built double quote must be represented by a pair of consecutive quotes.

Working with CSV Files

Consider a CSV file data.csv with the following content:

```
Name,Age,Occupation
John,30,Engineer
Jane,25,Teacher
Bob,40,Doctor
```

Working with CSV Files

In Python, you can use the csv module to read and write CSV files:

```
import csv

# Reading CSV File
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

# Writing to CSV File
data_to_write = [
    ['Alice', 28, 'Data Scientist'],
    ['Charlie', 35, 'Artist']
]

with open('new_data.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data_to_write)
```

Working with CSV Files

In this example, the csv.reader is used to read the content of data.csv, and csv.writer is used to write new data to new_data.csv. The newline=" parameter is important to ensure correct newline handling.

Panal Data and Python Data Analysis

- Pandas is a tool for data processing which helps in data analysis
- It provides functions and methods to efficiently manipulate large datasets
- Pandas is an open-source library built on top of NumPy

Advantages:

- Fast and efficient for data manipulation and analysis.
- Supports loading data from various file formats.
- Easy handling of missing data (represented as NaN).
- Size mutability allows dynamic column operations in DataFrames.
- Facilitates data set merging, joining, reshaping, and pivoting.

Pandas Data Structures:

- Pandas provides two main data structures:
  1. Series
  2. DataFrame

Series:

- A one-dimensional labeled array capable of holding data of any type.
- Index labels collectively form the index.
- Created using the pd.Series() constructor.
- Syntax:

```
import pandas as pd
series_data = pd.Series(data, index=index)
```

Example:

```
import pandas as pd

# Creating a simple Series
obj = pd.Series([3, 5, -8, 7, 9])
print(obj)
```

Output:

```
0    3
1    5
2   -8
3    7
4    9
dtype: int64
```

```
import pandas as pd

# Creating a Series with specified index
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
print(obj2)
```

Output:

```
d    4
b    7
a   -5
c    3
dtype: int64
```

```
import pandas as pd

# Creating a Series from a Python dict
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
print(obj3)
```

Output:

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Checking for Missing Data:

- The isnull() and notnull() functions detect missing data in a Series.
- Syntax:

```
import pandas as pd
result = pd.isnull(series)
result = pd.notnull(series)
```

- Example:

```
import pandas as pd
# Checking for missing data
missing_data = pd.isnull(obj3)
print(missing_data)
```

Output:

```
Ohio      False
Texas     False
Oregon    False
Utah      False
dtype: bool
```

Pandas Series Attributes and Methods

Attribute/Method	Description	Example
`axes`	Returns a list of the row axis labels.	`s.axes`
`dtype`	Returns the data type of the values in the Series.	`s.dtype`
`empty`	Returns `True` if the Series is empty; otherwise, `False`.	`s.empty`
`ndim`	Returns the number of dimensions (always 1 for a Series).	`s.ndim`
`size`	Returns the number of elements in the Series.	`s.size`
`values`	Returns the actual data as a NumPy array.	`s.values`
`head(n)`	Returns the first n elements from the Series (default n=5).	`s.head()`
`tail(n)`	Returns the last n elements from the Series (default n=5).	`s.tail()`

Pandas Series Attributes and Methods

Pandas DataFrame

- A Pandas DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).
- Data is organized in rows and columns, forming a table-like structure.
- Basic Operations on Pandas DataFrame:
  - Creating a DataFrame
  - Dealing with Rows and Columns
  - Indexing and Selecting Data
  - Working with Missing Data

Pandas DataFrame

Creating a DataFrame:

- Real-world data is often loaded into a DataFrame from existing storage like SQL Database, CSV file, or Excel file.
- DataFrames can be created from lists, dictionaries, or a list of dictionaries.

```
import pandas as pd

# Creating DataFrame from a list
lst = ['mec', 'minor', 'stud', 'eee', 'bio']
df = pd.DataFrame(lst)
print(df)

# Creating DataFrame from a dictionary of lists
data = {'Name': ['Tom', 'nick', 'krish', 'jack'], 'Age': [20, 21,
    19, 18]}
df = pd.DataFrame(data)
print(df)
```

Pandas DataFrame

- Output

```
       0
0    mec
1  minor
2   stud
3    eee
4    bio

    Name  Age
0    Tom   20
1   nick   21
2  krish   19
3   jack   18
```

Pandas DataFrame

Dealing with Rows and Columns:

- A DataFrame in pandas is like a table with rows and columns. You can easily work with its data by selecting, deleting, adding, or renaming rows and columns.
- To pick a specific column, you can simply use its name.
- Pandas offers a special way to get rows from a DataFrame using DataFrame.loc[], and you can also use iloc[] by giving it the row's number.

Pandas DataFrame

```
import pandas as pd
# Create a sample DataFrame with two columns 'A' and 'B', and index
    labels 'a' to 'e'
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50]},
index=['a', 'b', 'c', 'd', 'e'])

print("DataFrame:")
print(df) # Print the DataFrame

# Using loc: Select and print the values of row 'a' (labeled indexing)
print("Using loc:")
print(df.loc['a'])  # Selects the values in row 'a' for both columns 'A'
     and 'B'

# Using iloc: Select and print the values of the first row (integer-
    based indexing)
print("Using iloc:")
print(df.iloc[0])  # Selects the values in the first row for both
    columns 'A' and 'B'
```

Pandas DataFrame

```
DataFrame:
   A   B
a  1  10
b  2  20
c  3  30
d  4  40
e  5  50
Using loc:
A     1
B    10
Name: a, dtype: int64
Using iloc:
A     1
B    10
Name: a, dtype: int64
```

Indexing and Selecting Data:

- Indexing involves selecting particular rows and columns.

Pandas DataFrame

In this example, loc['a', 'A'] selects the value in row 'a' and column 'A', while iloc[0, 0] selects the value in the first row and first column (row 0 and column 0).

```
import pandas as pd

# Create a sample DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50]},
index=['a', 'b', 'c', 'd', 'e'])

# Using loc
print(df.loc['a', 'A'])  # Selects the value at row 'a' and column 'A'

# Using iloc
print(df.iloc[0, 0])  # Selects the value at row 0 (first row) and
    column 0 (first column)
#Output
1
1
```

Pandas DataFrame

- This code will use the loc method to select columns 'Name' and 'Qualification'.

```
import pandas as pd
# Your data dictionary
data = {'Name': ['John', 'Jai', 'Emma'],
    'Age': [25, 30, 35],
    'Qualification': ['B.Tech', 'MBA', 'Ph.D']}
# Creating a DataFrame from the dictionary
df = pd.DataFrame(data)

# Now you can use loc method on df
selected_data = df.loc[:, ['Name', 'Qualification']]
print(selected_data)

#Output
   Name Qualification
0  John         B.Tech
1   Jai            MBA
2  Emma          Ph.D
```

Working with Missing Data:

- Missing Data can occur, and functions like isnull() and notnull() help check and handle missing values.

```
import pandas as pd
import numpy as np
dict = {'First Score':[100, 90, np.nan, 95],
    'Second Score': [30, 45, 56, np.nan],
    'Third Score':[np.nan, 40, 80, 98]}
df = pd.DataFrame(dict)
print(df.isnull())
#nan stands for "Not a Number." NaN is used to represent missing or
     undefined values.
```

Output

```
   First Score  Second Score  Third Score
0        False         False         True
1        False         False        False
2         True         False        False
3        False          True        False
```

Pandas DataFrame

```
import pandas as pd
import numpy as np
dict = {'First Score':[100, 90, np.nan, 95],
    'Second Score': [30, 45, 56, np.nan],
    'Third Score':[np.nan, 40, 80, 98]}
df = pd.DataFrame(dict)
print(df.notnull())
```

Output

```
   First Score  Second Score  Third Score
0         True          True        False
1         True          True         True
2        False          True         True
3         True         False         True
```

Pandas DataFrame

```
import pandas as pd
import numpy as np
dict = {'First Score':[100, 90, np.nan, 95],
    'Second Score': [30, 45, 56, np.nan],
    'Third Score':[np.nan, 40, 80, 98]}
df = pd.DataFrame(dict)
print(df.notnull())
```

Output

```
   First Score  Second Score  Third Score
0         True          True        False
1         True          True         True
2        False          True         True
3         True         False         True
```

Filling missing values using fillna(), replace() and interpolate() :

- fillna():
  - This method is used to fill NaN (Not a Number) or missing values with a specified value or a value derived from nearby data.

```
df.fillna(value=0, inplace=True)
```

- replace():
  - The replace() method allows you to replace specified values with another value. You can use it to replace missing values with a specific value.

```
df.replace(to_replace=np.nan, value=0, inplace=True)
```

- interpolate():
  - This method is used to fill missing values by computing intermediate values based on the existing data. It can be useful for time series data where missing values can be estimated based on surrounding values.

```
df.interpolate(method='linear', inplace=True)
```

Filling missing values using fillna(), replace() and interpolate() :

```
import pandas as pd
import numpy as np

# Dictionary of lists
data = {'First Score': [100, 90, np.nan, 95],
    'Second Score': [30, 45, 56, np.nan],
    'Third Score': [np.nan, 40, 80, 98]}

# Creating a DataFrame from the dictionary
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)

# Filling missing values using fillna()
filled_df = df.fillna(0)
print("\nDataFrame after filling NaN with 0:")
print(filled_df)
```

Pandas DataFrame

```
# Filling NaN values by interpolation
interpolated_df = df.interpolate()
print("\nDataFrame after interpolating NaN values:")
print(interpolated_df)

# Replacing the NaN values with -1
replaced_df = df.replace(np.nan, -1)
print("\nDataFrame after replacing NaN with -1:")
print(replaced_df)

# Dropping the rows containing null values
dropped_df = df.dropna()
print("\nDataFrame after dropping rows with null values:")
print(dropped_df)
```

Pandas DataFrame

Output

```
Original DataFrame:
    First Score  Second Score  Third Score
0         100.0          30.0          NaN
1          90.0          45.0         40.0
2           NaN          56.0         80.0
3          95.0           NaN         98.0
```

```
DataFrame after filling NaN with 0:
    First Score  Second Score  Third Score
0         100.0          30.0          0.0
1          90.0          45.0         40.0
2           0.0          56.0         80.0
3          95.0           0.0         98.0
```

Pandas DataFrame

```
DataFrame after interpolating NaN values:
    First Score  Second Score  Third Score
0         100.0          30.0          NaN
1          90.0          45.0         40.0
2          92.5          56.0         80.0
3          95.0          56.0         98.0
```

```
DataFrame after replacing NaN with -1:
    First Score  Second Score  Third Score
0         100.0          30.0         -1.0
1          90.0          45.0         40.0
2          -1.0          56.0         80.0
3          95.0          -1.0         98.0
```

```
DataFrame after dropping rows with null values:
    First Score  Second Score  Third Score
1          90.0          45.0         40.0
```

DataFrame basic functionality

DataFrame Basic Functionality - Part 1

Attribute/Method	Description
`T`	Transpose the DataFrame (swap rows and columns).
`axes`	Returns a list representing the axes of the DataFrame.
`dtype`	Returns the data type of the DataFrame.
`empty`	Returns `True` if the DataFrame is empty, `False` otherwise.

DataFrame Basic Functionality - Part 2

Attribute/Method	Description
`ndim`	Returns the number of dimensions (always 2 for DataFrames).
`shape`	Returns a tuple representing the dimensions (number of rows, number of columns) of the DataFrame.
`size`	Returns the total number of elements in the DataFrame.
`values`	Returns a 2D array (NumPy array) of the DataFrame's data.
`head(n)`	Returns the first n rows of the DataFrame.
`tail(n)`	Returns the last n rows of the DataFrame.

Pandas: Reading & Writing functions

read_csv() Function:

- read_csv() is used to read data from a CSV file and create a DataFrame.

```
import pandas as pd

# Reading data from a CSV file into a DataFrame
df = pd.read_csv('file.csv')
```

  - Parameters:
    - filepath_or_buffer: Path or object to the CSV file.
    - Many other optional parameters like sep (separator/delimiter), header, index_col, etc.

- Example:

```
df = pd.read_csv('data.csv', sep=',', header=0)
```

Pandas: Reading & Writing functions

to_csv() Function:

- to_csv() is used to write a DataFrame to a CSV file.

```
import pandas as pd

# Writing a DataFrame to a CSV file
df.to_csv('output_file.csv', index=False)
```

  - Parameters:
    - path_or_buf: File path or object.
    - Many other optional parameters like sep (separator/delimiter), header, index, etc.

- Example:

```
df.to_csv('output_data.csv', sep='\t', index=False)
```

In this example, the DataFrame df is written to a file named 'output_data.csv' using a tab ('\t') as the separator, and the index is excluded from the output.

Pandas: Reading & Writing functions

```
import pandas as pd
# dictionary of lists
data = {'name': ["aparna", "pankaj", "sudhir", "Geeku"],
    'degree': ["MBA", "BCA", "M.Tech", "MBA"],
    'score': [90, 40, 80, 98]}
# creating a DataFrame from a dictionary
df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)

# Exporting DataFrame to a CSV file
df.to_csv('studdata.csv', index=False)

# Reading DataFrame from the CSV file
read_df = pd.read_csv('studdata.csv')

# Displaying the DataFrame read from the CSV file
print(read_df)
```

Functions related to data manipulation and analysis

functions related to data manipulation and analysis

- agg() (Aggregate)
- append()
- apply()
- groupby

Functions related to data manipulation and analysis

agg() (Aggregate):

- The agg() function in pandas is used to perform aggregation on a DataFrame.(such as sum, mean, median, min, max, etc.,)
- It allows you to apply different aggregation functions to different columns or the same aggregation function to multiple columns.

```
import pandas as pd
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': [9, 10, 11, 12]
})
result = df.agg({'A': 'sum', 'B': 'mean'})
print(result)
```

Output

```
A    10.0
B     6.5
dtype: float64
```

Functions related to data manipulation and analysis

append():

- The append() method is used to concatenate two DataFrames along a particular axis.
- It creates a new DataFrame by appending rows from the second DataFrame to the end of the first DataFrame.

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

result = df1.append(df2, ignore_index=True)
print(result)
```

- output

```
   A  B
0  1  3
1  2  4
2  5  7
3  6  8
```

Functions related to data manipulation and analysis

apply():

- The apply() function in pandas is used to apply a function along the axis of a DataFrame or Series.
- It can be used to apply a custom function to each element, row, or column of the DataFrame.

```
import pandas as pd
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]})
def custom_function(x):
    return x * 2
result = df.apply(custom_function)
#output
   A   B
0  2  10
1  4  12
2  6  14
3  8  16
```

Functions related to data manipulation and analysis

groupby:

- The groupby() function is used for grouping data based on some criteria.
- It is often followed by an aggregation function to perform some operation on each group of data.

```
import pandas as pd
df = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'B'],
    'Value': [10, 20, 30, 40]
})
grouped_data = df.groupby('Category')
result = grouped_data.sum()
```

- output

```
          Value
Category
A            40
B            60
```

Descriptive Statistics Functions

Pandas Descriptive Statistics: functions to compute descriptive statistics on your data.

Function	Description
`count()`	Returns the number of non-null values for each column.
`sum()`	Computes the sum of values for each column.
`mean()`	Calculates the mean for each column.
`median()`	Computes the median for each column.
`mode()`	Calculates the mode for each column.
`std()`	Calculates the standard deviation for each column.
`min()`	Returns the minimum value for each column.
`max()`	Returns the maximum value for each column.

Descriptive Statistics Functions (contd.)

Function	Description
`abs()`	Computes the absolute values of each element in the DataFrame.
`prod()`	Calculates the product of values for each column.
`cumsum()`	Computes the cumulative sum for each column.
`cumprod()`	Computes the cumulative product for each column.

Pandas Descriptive Statistics Functions with Examples

- Below are some commonly used Pandas functions for descriptive statistics with examples.

Sample Input DataFrame

- Let's define a sample input DataFrame 'df' for demonstration purposes.

```
import pandas as pd
# Creating a sample DataFrame
data = {'column1': [1, 2, 3, 4, 5],
        'column2': [3, 4, 5, 6, 7]}
df = pd.DataFrame(data)
print("Sample Input DataFrame 'df':")
print(df)

#Output:
Sample Input DataFrame 'df':
        column1  column2
0         1        3
1         2        4
2         3        5
3         4        6
4         5        7
```

1. count()

- Returns the number of non-null values for each column.
- Example:

```
count_values = df.count()
print("Count of non-null values for each column:")
print(count_values)
```

Output:

```
count_values:
column1    5
column2    5
dtype: int64
```

2. sum()

- Computes the sum of values for each column.
- Example:

```
sum_values = df.sum()
print("Sum of values for each column:")
print(sum_values)
```

Output:

```
Sum of values for each column:
column1    15
column2    25
dtype: int64
```

3. mean()

- Calculates the mean for each column.
- Example:

```
mean_values = df.mean()
print("Mean for each column:")
print(mean_values)
```

Output:

```
Mean for each column:
column1    3.0
column2    5.0
dtype: float64
```

4. median()

- Computes the median for each column.
- Example:

```
median_values = df.median()
print("Median for each column:")
print(median_values)
```

Output:

```
Median for each column:
column1    3.0
column2    5.0
dtype: float64
```

5. mode()

- Calculates the mode for each column.
- Example:

```
mode_values = df.mode().iloc[0]
print("Mode for each column:")
print(mode_values)
```

Output:

```
Mode for each column:
column1    1
column2    3
Name: 0, dtype: int64
```

6. std()

- Calculates the standard deviation for each column.
- Example:

```
std_dev = df.std()
print("Standard deviation for each column:")
print(std_dev)
```

Output:

```
Standard deviation for each column:
column1    1.581139
column2    1.581139
dtype: float64
```

7. min()

- Returns the minimum value for each column.
- Example:

```
min_values = df.min()
print("Minimum value for each column:")
print(min_values)
```

Output:

```
Minimum value for each column:
column1    1
column2    3
dtype: int64
```

8. max()

- Returns the maximum value for each column.
- Example:

```
max_values = df.max()
print("Maximum value for each column:")
print(max_values)
```

Output:

```
Maximum value for each column:
column1    5
column2    7
dtype: int64
```

9. abs()

- Computes the absolute values of each element in the DataFrame.
- Example:

```
abs_values = df.abs()
print("Absolute values of each element:")
print(abs_values)

#Output:
Absolute values of each element:
    column1  column2
0         1        3
1         2        4
2         3        5
3         4        6
4         5        7
```

10. prod()

- Calculates the product of values for each column.
- Example:

```
prod_values = df.prod()
print("Product of values for each column:")
print(prod_values)
```

Output:

```
Product of values for each column:
column1     120
column2    2520
dtype: int64
```

11. cumsum()

- Computes the cumulative sum for each column.
- Example:

```
cumsum_values = df.cumsum()
print("Cumulative sum for each column:")
print(cumsum_values)

#Output:
Cumulative sum for each column:
    column1  column2
0         1        3
1         3        7
2         6       12
3        10       18
4        15       25
```

12. cumprod()

- Computes the cumulative product for each column.
- Example:

```
cumprod_values = df.cumprod()
print("Cumulative product for each column:")
print(cumprod_values)

#Output:
Cumulative product for each column:
    column1  column2
0         1        3
1         2       12
2         6       60
3        24      360
4       120     2520
```

Load CSV files to Python Pandas

- To load CSV files into Python using Pandas, you can use the read_csv function provided by Pandas.

```
import pandas as pd

# Replace 'your_file.csv' with the actual path to your CSV file
file_path = 'your_file.csv'

# Load CSV file into a Pandas DataFrame
df = pd.read_csv(file_path)

# Display the DataFrame
print("DataFrame loaded from CSV:")
print(df)
```

Pandas read_csv() Function with Parameters

- The `read_csv()` function in Pandas is used to read CSV files into a DataFrame. It has several parameters for customization.

Pandas read_csv() Function with Parameters

- Sample input CSV file:

```
# Sample CSV content (saved as 'sample_input.csv')
column1,column2,column3
    1,      2,      3
    4,      5,      6
    7,      8,      9
```

1. filepath_or_buffer

- Description: Location of the file to be retrieved (string path or URL).
- Example:

```
file_path = 'sample_input.csv'
df = pd.read_csv(file_path)

#Output:
DataFrame loaded from CSV:
    column1  column2  column3
0        1        2        3
1        4        5        6
2        7        8        9
```

1. head() method

- If you want to display the first few rows of your DataFrame to get a quick overview, you can use the head() method.
- Example:

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('sample_input.csv')

# Display the first few rows using the head() method
print(df.head())

#Output:
   column1  column2  column3
0        1        2        3
1        4        5        6
2        7        8        9
```

2. sep

- Description: Separator for the CSV file (default is ‘,‘ for comma-separated values).
- Example:

```
df = pd.read_csv(file_path, sep=',')  # Default separator

#Output:
DataFrame loaded from CSV:
    column1  column2  column3
0         1        2        3
1         4        5        6
2         7        8        9
```

3. header

- Description: Row numbers to use as column names and start of the data.
- Example:

```
df = pd.read_csv(file_path, header=0)  # Use the first row as column
    names

#Output:
    DataFrame loaded from CSV:
    column1  column2  column3
0         1         2         3
1         4         5         6
2         7         8         9
```

4. usecols

- Description: Retrieve only selected columns from the CSV file.
- Example:

```
df = pd.read_csv(file_path, usecols=['column1', 'column3'])  # Retrieve
    specific columns

#Output:
DataFrame loaded from CSV:
    column1  column3
0         1         3
1         4         6
2         7         9
```

5. nrows

- Description: Number of rows to be displayed from the dataset.
- Example:

```
df = pd.read_csv(file_path, nrows=2)  # Display only the first 2 rows

#Output:
    DataFrame loaded from CSV:
    column1  column2  column3
0         1        2        3
1         4        5        6
```

6. index_col

- Description: If `None`, there are no index numbers displayed along with records.
- Example:

```
df = pd.read_csv(file_path, index_col='column1')  # Set 'column1' as the
     index

#Output:
DataFrame loaded from CSV with 'column1' as index:
         column2  column3
column1
1              2        3
4              5        6
7              8        9
```

7. squeeze

- Description: If `True` and only one column is passed, it returns a Pandas Series instead of a DataFrame.
- Example:

```
series = pd.read_csv(file_path, usecols=['column1'], squeeze=True)  #
    Returns a series
```

Output:

```
Series loaded from CSV:
0    1
1    4
2    7
Name: column1, dtype: int64
```

8. skiprows

- Description: Skips passed rows in the new DataFrame.
- Example:

```
df = pd.read_csv(file_path, skiprows=[1])  # Skip the second row

#Output:
DataFrame loaded from CSV with the second row skipped:
    column1  column2  column3
0        4        5        6
1        7        8        9
```

9. names

- Description: Allows you to retrieve columns with new names.
- Example:

```
df = pd.read_csv(file_path, names=['new_column1', 'new_column2', '
    new_column3'])  # Rename columns
```

Output:

```
DataFrame loaded from CSV with renamed columns:
        new_column1  new_column2  new_column3
0             1            2            3
1             4            5            6
2             7            8            9
```

10. rename - Renaming Column Headers

- Description: Rename one or more columns in the DataFrame.
- Example:

```
# Original DataFrame
df = pd.read_csv(file_path)
print("Original DataFrame:")
print(df)

# Rename columns
df = df.rename(columns={'column1': 'new_column1', 'column2': '
    new_column2', 'column3': 'new_column3'})
```

Output:

```
        Original DataFrame:
        column1  column2  column3
0        1        2        3
1        4        5        6
2        7        8        9
```

10. rename - Renaming Column Headers

```
DataFrame after renaming columns:
    new_column1  new_column2  new_column3
0             1            2            3
1             4            5            6
2             7            8            9
```

Writing to CSV File

- Description: Pandas DataFrame provides the `to_csv()` method to write/export DataFrame to a CSV file.
- Example:

```
# Original DataFrame
df = pd.read_csv(file_path)
print("Original DataFrame:")
print(df)

# Write DataFrame to CSV with header and index
df.to_csv('output_file.csv')
```

Writing to CSV Without Header

- Description: Write DataFrame to CSV without including column headers.
- Example:

```
# Write DataFrame to CSV without header
df.to_csv('output_file_no_header.csv', header=False)
```

Writing Using Custom Delimiter

- Description: Write DataFrame to CSV using a custom delimiter (e.g., pipe '|') without including column headers.
- Example:

```
# Write DataFrame to CSV with custom delimiter and without header
df.to_csv('output_file_custom_delimiter.csv', header=False, sep='|')
```

Writing to CSV Ignoring Index

- Description: Write DataFrame to CSV without including the index column.
- Example:

```
# Write DataFrame to CSV without index
df.to_csv('output_file_no_index.csv', index=False)
```

Export Selected Columns to CSV File

- Description: Write only selected columns to a CSV file while ignoring the index.
- Example:

```
# Export selected columns to CSV without index
column_names = ['Courses', 'Fee', 'Discount']
df.to_csv('output_file_selected_columns.csv', index=False, columns=
    column_names)
```

Change Header Column Names While Writing

- Description: Write DataFrame to CSV with custom header column names and without including the index.
- Example:

```
# Write DataFrame to CSV with custom header and without index
column_names = ['Courses', 'Course_Fee', 'Course_Duration', '
    Course_Discount']
df.to_csv('output_file_custom_header.csv', index=False, header=
    column_names)
```

Write DataFrame to CSV by Encoding

- Description: Write DataFrame to CSV using a specific delimiter (tab in this case) and encoding (UTF-8).
- Example:

```
# Write DataFrame to CSV with tab delimiter and UTF-8 encoding
df.to_csv('output_file_encoding.csv', sep='\t', encoding='utf-8')
```

Append DataFrame to Existing CSV File

- Description: Append DataFrame to an existing CSV file without including column headers and using a custom delimiter (pipe '|').
- Example:

```
# Append DataFrame to an existing CSV file with pipe delimiter and
    without header
df.to_csv("existing_file.csv", header=False, sep='|', index=False, mode=
    'a')
```

Writing to CSV file

```
import pandas as pd
# Create a sample DataFrame
data = {
    'Name': ['John', 'Alice', 'Bob'],
    'Age': [25, 30, 22],
    'City': ['New York', 'San Francisco', 'Seattle']
}

df = pd.DataFrame(data)

# Print the original DataFrame
print("Original DataFrame:")
print(df)

# Write DataFrame to CSV file
output_file = 'output_file.csv'
df.to_csv(output_file, index=False)

# Print a message indicating the operation is completed
print(f"\nDataFrame has been written to '{output_file}'.")
```

CSV Data Cleaning - Missing Values

- In Pandas, missing values are usually denoted by NaN (NumPy's special floating-point NaN).
- Cleaning missing values involves dropping or replacing NaN values.

1. Dropping Missing Values

- `df.dropna()`: Drop all rows that have any NaN values.
- `df.dropna(how='all')`: Drop only if ALL columns are NaN.
- `df.dropna(thresh=2)`: Drop rows with fewer than two non-NaN values.
- `df.dropna(subset=[1])`: Drop rows with NaN in a specific column.

```
# Original DataFrame with missing values
df = pd.read_csv('input_file.csv')
print("Original DataFrame:")
print(df)

# Drop rows with any NaN values
df_dropped = df.dropna()
# Drop rows only if ALL columns are NaN
df_dropped_all = df.dropna(how='all')
# Drop rows with fewer than two non-NaN values
df_thresh = df.dropna(thresh=2)
# Drop rows with NaN in a specific column (e.g., column with index 1)
df_subset = df.dropna(subset=[1])
```

2. Replacing Missing Values

- `df.fillna(value)`: Replace NaN values with a specified value.
- Example:

```
# Replace NaN values with mean of the column
mean_value = df[1].mean()
df_filled = df.fillna(value=mean_value)
```

Exercises

- Consider the above student.csv file with fields Name, Branch, Year, CGPA .Write python code using pandas to
  1. To find the average CGPA of the students
  2. To display the details of all students having $CGPA > 9$
  3. To display the details of all CSE students with $CGPA > 9$
  4. To display the details of student with maximum CGPA
  5. To display average CGPA of each branch

Exercises

```
import pandas as pd

# Load CSV file into a DataFrame
df = pd.read_csv('student.csv')

# Task 1: To find the average CGPA of the students
average_cgpa = df['CGPA'].mean()
print("Average CGPA of the students:", average_cgpa)

# Task 2: To display the details of all students having CGPA > 9
high_cgpa_students = df[df['CGPA'] > 9]
print("Details of students having CGPA > 9:")
print(high_cgpa_students)

# Task 3: To display the details of all CSE students with CGPA > 9
cse_high_cgpa_students = df[(df['Branch'] == 'CSE') & (df['CGPA'] > 9)]
print("Details of CSE students with CGPA > 9:")
print(cse_high_cgpa_students)
```

Exercises

```
# Task 4: To display the details of student with maximum CGPA
max_cgpa_student = df[df['CGPA'] == df['CGPA'].max()]
print("Details of student with maximum CGPA:")
print(max_cgpa_student)

# Task 5: To display average CGPA of each branch
average_cgpa_per_branch = df.groupby('Branch')['CGPA'].mean()
print("Average CGPA of each branch:")
print(average_cgpa_per_branch)
```

Exercises

- Write Python program to write the data given below to a CSV file named student.csv

$$
\begin{aligned}
fields &= [\text{'Name'},\text{' Branch'},\text{' Year'},\text{' CGPA'}] \\
rows &= [[\text{'Nikhil'},\text{' CSE'},\text{' 2'},\text{' 8.0'}], \\
&\quad [\text{'Sanchit'},\text{' CSE'},\text{' 2'},\text{' 9.1'}], \\
&\quad [\text{'Aditya'},\text{' IT'},\text{' 2'},\text{' 9.3'}], \\
&\quad [\text{'Sagar'},\text{' IT'},\text{' 1'},\text{' 9.5'}]]
\end{aligned}
$$

Exercises

```
import pandas as pd

# Data to be written to the CSV file
fields = ['Name', 'Branch', 'Year', 'CGPA']
rows = [
['Nikhil', 'CSE', '2', '8.0'],
['Sanchit', 'CSE', '2', '9.1'],
['Aditya', 'IT', '2', '9.3'],
['Sagar', 'IT', '1', '9.5']
]

# Create DataFrame
df = pd.DataFrame(rows, columns=fields)

# Write DataFrame to CSV file
df.to_csv('student.csv', index=False)

print("Data has been written to student.csv")
```

Exercises

Consider a CSV file 'employee.csv' with the following columns(name, gender, start_date ,salary, team). Write commands to do the following using panda library.

- print first 7 records from employees file
- print all employee names in alphabetical order
- find the name of the employee with highest salary
- list the names of male employees
- to which all teams employees belong

Exercises

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('employee.csv')

# Task 1: Print first 7 records from employees file
print("First 7 records from employees file:")
print(df.head(7))

# Task 2: Print all employee names in alphabetical order
print("\nAll employee names in alphabetical order:")
print(sorted(df['name']))

# Task 3: Find the name of the employee with the highest salary
highest_salary_employee = df.loc[df['salary'].idxmax()]['name']
print("\nName of the employee with the highest salary:",
    highest_salary_employee)
```

Exercises

```
# Task 4: List the names of male employees
male_employee_names = df[df['gender'] == 'Male']['name'].tolist()
print("\nNames of male employees:", male_employee_names)

# Task 5: List all teams employees belong to
teams = df['team'].unique()
print("\nTeams employees belong to:", teams)
```

Exercises

Given the sales information of a company as CSV file with the following fields month_number, facecream, facewash, toothpaste, bathingsoap, shampoo, moisturizer, total_units, total_profit. Write Python codes to visualize the data as follows

- Toothpaste sales data of each month and show it using a scatter plot
- Face cream and face wash product sales data and show it using the bar chart
- Calculate totalsale data for last year for each product and show it using a Pie chart.

Exercises

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

# Read the CSV file
sales_data = pd.read_csv('sales_data.csv')

# 1) Toothpaste sales data of each month and show it using a scatter
    plot
plt.figure(figsize=(10, 6))
plt.scatter(sales_data['month_number'], sales_data['toothpaste'], color=
    'blue')
plt.title('Toothpaste Sales Data')
plt.xlabel('Month Number')
plt.ylabel('Toothpaste Sales')
plt.grid(True)
plt.show()
```

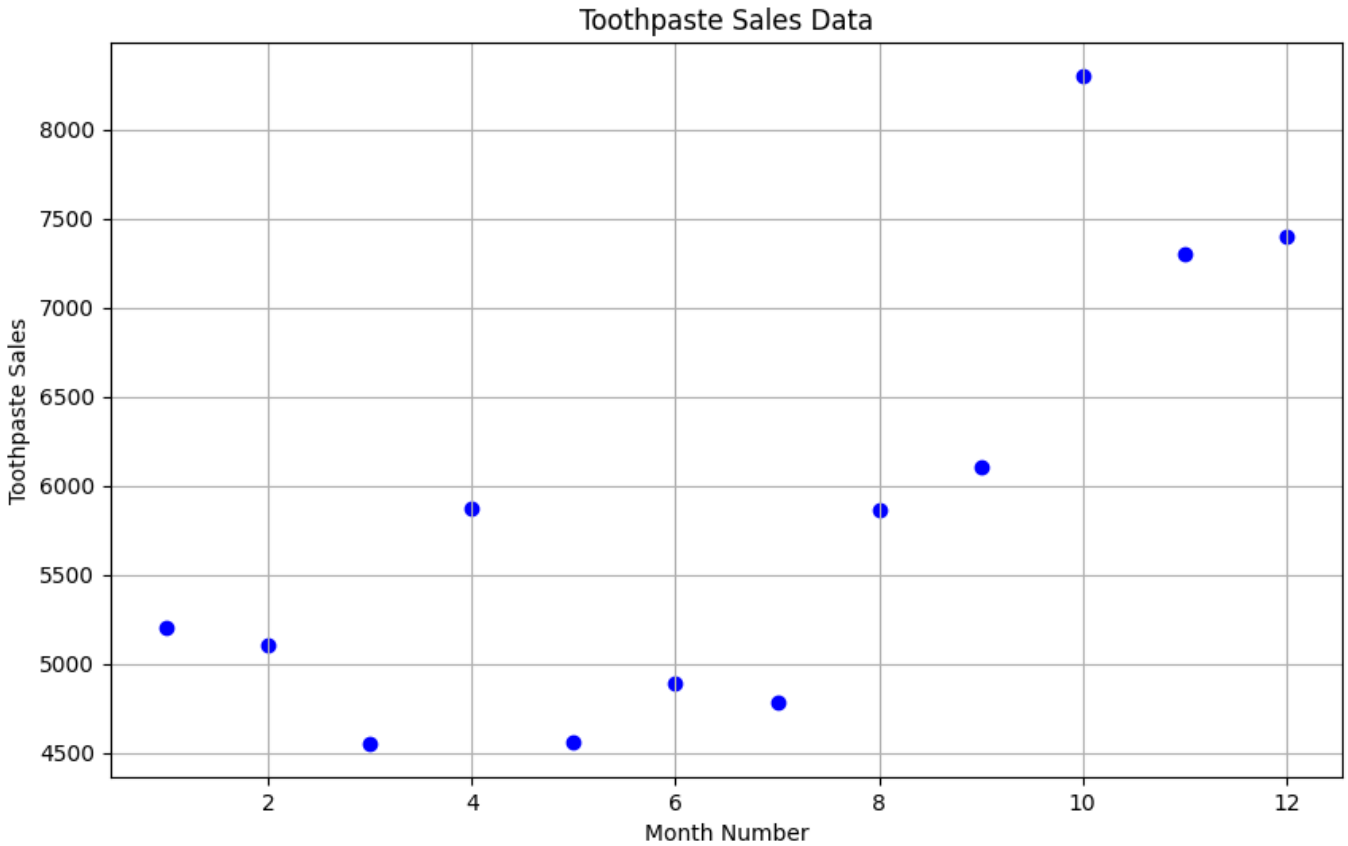
Exercises

```
# 2) Face cream and face wash product sales data and show it using a bar
     chart
plt.figure(figsize=(10, 6))
plt.bar(sales_data['month_number'], sales_data['facecream'], color='
    skyblue', label='Face Cream')
plt.bar(sales_data['month_number'], sales_data['facewash'], color='
    lightgreen', label='Face Wash', alpha=0.6)
plt.title('Face Cream and Face Wash Sales Data')
plt.xlabel('Month Number')
plt.ylabel('Sales Quantity')
plt.legend()
plt.show()
```

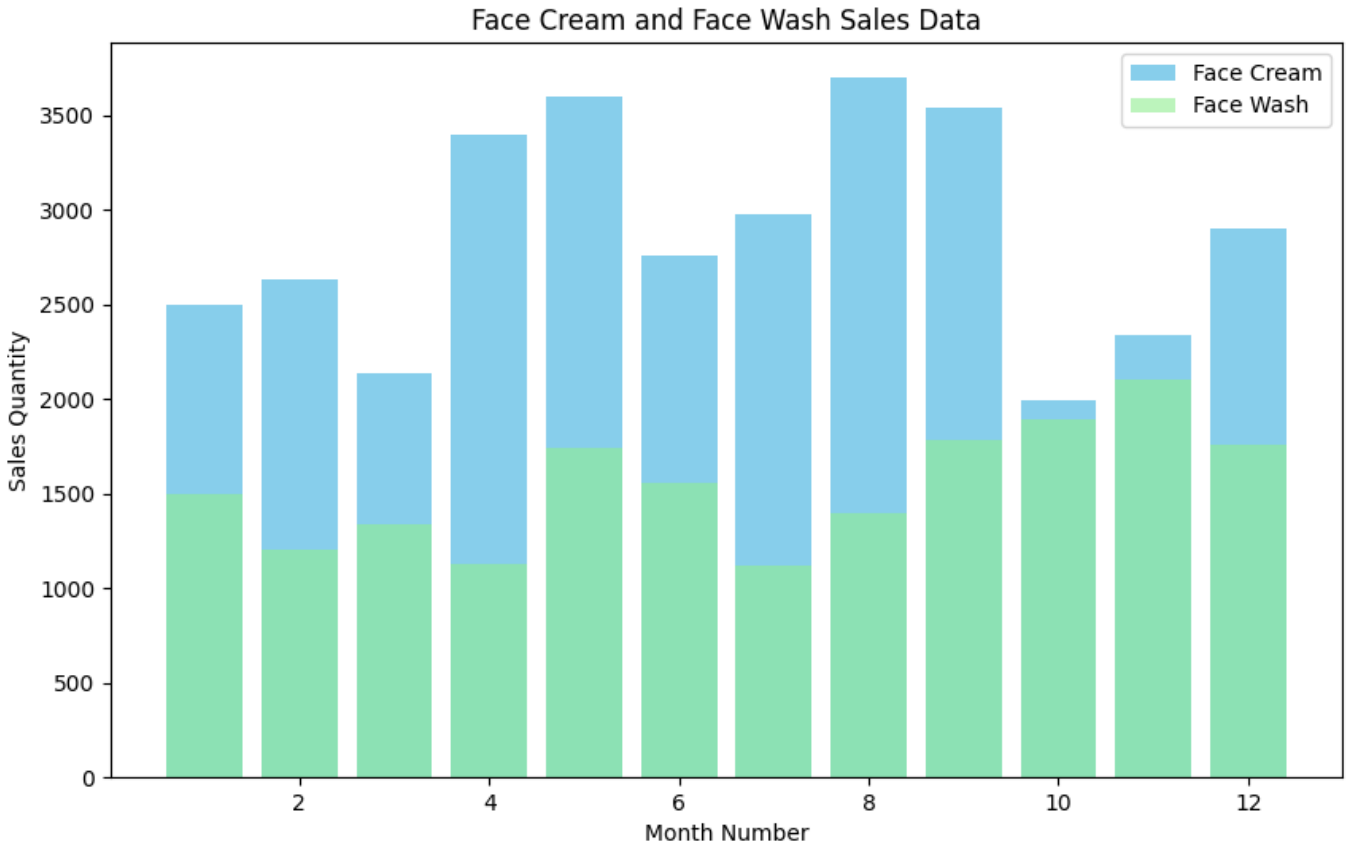
Exercises

```
# 3) Calculate total sale data for last year for each product and show
    it using a Pie chart
total_sales = sales_data[['facecream', 'facewash', 'toothpaste', '
    bathingsoap', 'shampoo', 'moisturizer']].sum()
labels = total_sales.index
plt.figure(figsize=(8, 8))
plt.pie(total_sales, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title('Total Sales Data for Last Year')
plt.show()
```

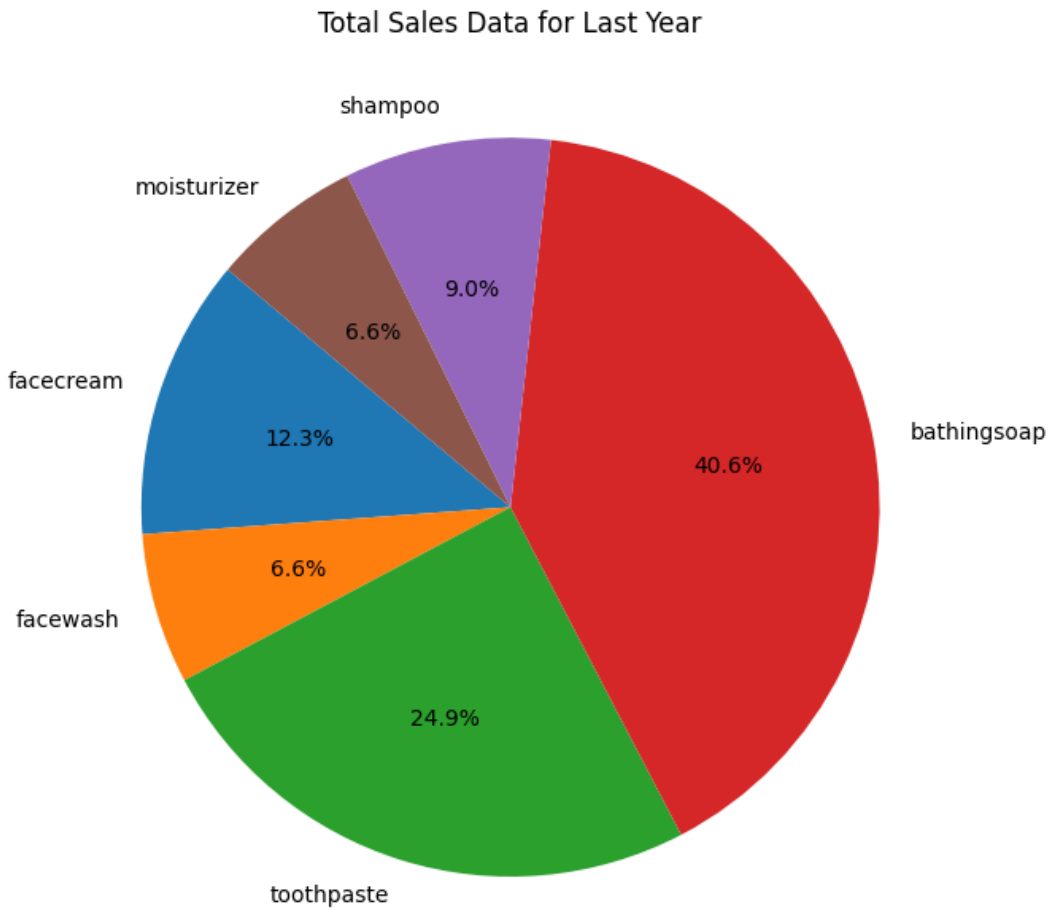
Exercises


Toothpaste Sales Data
Toothpaste Sales
Month Number
8000
7500
7000
6500
6000
5500
5000
4500
2
4
6
8
10
12

Exercises


Face Cream and Face Wash Sales Data
Face Cream
Face Wash
Sales Quantity
0
500
1000
1500
2000
2500
3000
3500
2
4
6
8
10
12
Month Number

Exercises


Total Sales Data for Last Year
shampoo
moisturizer
facecream
facewash
toothpaste
bathingsoap
9.0%
6.6%
12.3%
6.6%
24.9%
40.6%

Introduction to Microservices using Flask

- Flask is a micro web framework written in Python. It is categorized as a microframework due to its minimalistic nature, avoiding the necessity for specific tools or libraries.
- Unlike some other frameworks, Flask does not come with built-in database abstraction layer, form validation, or other components commonly provided by third-party libraries.
- Flask is a Python-based web application framework developed by Armin Ronacher, who leads an international group of Python enthusiasts called Pocco.
- Flask utilizes the Werkzeug WSGI toolkit and Jinja2 template engine as its foundational components. Both Werkzeug and Jinja2 are projects under the Pocco umbrella.

Introduction to Microservices using Flask

- To install virtualenv for creating development environments in Python, you can follow these steps:

```
$sudo apt-get install virtualenv
```

- To activate corresponding environment, on Linux/OS X, use the

```
venv/bin/activate
```

To create a basic Flask application template, you can follow this structure.

- Create a directory for your Flask project:

```
mkdir my_flask_app
cd my_flask_app
```

- Create a Python virtual environment and activate it:

```
virtualenv venv
source venv/bin/activate
```

- Install Flask within your virtual environment:

```
pip install flask
```

- Create a Python script named app.py:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=True)
```

- Run the Flask application:

```
python app.py
```

Introduction to Microservices using Flask

- Importing Flask: from flask import Flask
- Creating the Flask Application Object: app = Flask(__name__)
  - Here, we create an instance of the Flask class and assign it to the variable app.
- Defining a Route: @app.route('/')
  - This is a decorator provided by Flask that associates the URL path '/' with the hello_world function below it. When a request is made to the root URL of the application (i.e., http://localhost:5000/), Flask will invoke the hello_world function and return its result as the response.
- View Function: def hello_world():
  - We define a function called hello_world that contains the logic for generating the response that Flask will send back to the user's web browser. In this case, the function simply returns the string 'Hello, World!', which will be displayed in the browser.

- Running the Application: if __name__ == '__main__': app.run(debug=True)
  - This conditional statement ensures that the Flask development server is started only if the script is executed directly (i.e., not imported as a module). When the script is run directly, the app.run() method starts the Flask development server on localhost (127.0.0.1) and port 5000. The debug=True argument enables debug mode, which provides helpful debugging information in case of errors and automatically reloads the application when code changes are detected during development.

Introduction to Microservices using Flask

Flask application that has two routes: one for displaying a greeting message and another for echoing back a message provided in the URL.

```
#In Flask, the ‘request‘ object lets you easily work with incoming data
    from HTTP requests, like form inputs, query parameters, and uploaded
     files.

from flask import Flask, request

app = Flask(__name__)

# Define a route for the root URL
@app.route('/')
def hello_world():
    return 'Hello, World!'

# Define a route for displaying a personalized greeting
@app.route('/greet/<name>')
def greet(name):
    return f'Hello, {name}!'
```

```
# Define a route for echoing back a message
@app.route('/echo')
def echo():
    message = request.args.get('message', 'No message provided')
    return f'You said: {message}'
# Run the Flask application
if __name__ == '__main__':
    app.run(debug=True)
```

```
http://127.0.0.1:5000/: Displays "Hello, World!"
http://127.0.0.1:5000/greet/John: Displays "Hello, John!"
http://127.0.0.1:5000/echo?message=Hello: Echoes back "You said: Hello"
```

Introduction to Microservices using Flask

This example demonstrates how to use the *request.args.get*() method to retrieve query parameters from the URL.

```
from flask import Flask, request

app = Flask(__name__)
# Define a route for handling GET requests
@app.route('/get-example', methods=['GET'])
def handle_get_request():
    # Retrieve query parameters from the URL
    name = request.args.get('name', 'Anonymous')
    age = request.args.get('age', 'Unknown')

    # Construct the response message
    response = f'Hello, {name}! Your age is {age}.'

    return response

if __name__ == '__main__':
    app.run(debug=True)
```

Introduction to Microservices using Flask

This example demonstrates how to use the *request.form.get*() method to retrieve form data from a POST request.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/post-example', methods=['POST'])
def handle_post_request():
    # Retrieve form data from the request
    name = request.form.get('name', 'Anonymous')
    age = request.form.get('age', 'Unknown')

    response = f'Hello, {name}! Your age is {age}.'

    return response

if __name__ == '__main__':
    app.run(debug=True)
```

Flask Example: Returning JSON Data for Books

```
from flask import Flask, jsonify

app = Flask(__name__)

# Sample data (usually this would come from a database)
books = [
    {"title": "Python Programming", "author": "John Doe"},
    {"title": "Web Development with Flask", "author": "Jane Smith"}
    ]

# Route to get all books
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify(books)

if __name__ == '__main__':
    app.run(debug=True)
```

- In this example:
  - We define a Flask application with one route: /books.
  - The get_books function returns a JSON response containing a list of books.
  - We use the jsonify function from Flask to convert the Python list into a JSON response.

Caching in Flask

```
from flask import Flask
from flask_caching import Cache
from random import randint

app = Flask(__name__)

# Initialize the cache
cache = Cache(app)
app.config['CACHE_TYPE'] = 'simple'

@app.route('/')
@cache.cached(timeout=5)
def index():
    randnum = randint(1,1000)
    return f'<h1>The number is: {randnum}</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```

Caching in Flask

- When a user visits the root URL of the Flask application, the index() function is executed.
- If the response for this function is found in the cache and it's still valid (i.e., within the timeout period), the cached response is returned directly without executing the function again.
- If the response is not found in the cache or it's expired, the function is executed, and the response is stored in the cache for future use.
- This helps in reducing the processing time and improving the performance of the web application.

Caching in Flask using @cache.memoize

```
from flask import Flask
from flask_caching import Cache

app = Flask(__name__)
cache = Cache(app, config={'CACHE_TYPE': 'simple'})

@app.route('/memoize/<int:num>')
@cache.memoize(timeout=30)
def memoized_route(num):
    # Simulating expensive computation
    result = num * num
    return f'The square of {num} is: {result}'

if __name__ == '__main__':
    app.run(debug=True)
```

- The memoized_route function takes an integer parameter num, computes its square, and returns the result.
- The @cache.memoize decorator caches the result of this function based on the num argument for 30 seconds.

@cache.memoize(timeout=50):

- This decorator is used to cache the result of a specific function call based on its input parameters. It memoizes the function's return value based on the arguments passed to it.
- If the same set of arguments is passed to the function within the specified timeout period (50 seconds in this case), the cached result will be returned without re-executing the function.

Flask Web Application: This Flask web application serves static web pages using Flask's render_template function.

```
from flask import Flask, render_template

app = Flask(__name__, template_folder='pages')

@app.route("/")
def home():
    return render_template("home.html")


@app.route("/about")
def about():
    return render_template("about.html")

if __name__ == '__main__':
    app.run(debug=True)
```

Program Description:

- **Import Dependencies:** The program imports the Flask class and the render_template function from the Flask package.
- textbfInitialize Flask App: It initializes a Flask application instance named app, specifying the template folder as 'pages'.
- **Define Routes:**
  - **Home Route ("/"):** When a user navigates to the root URL ("/"), the home() function is triggered. This function renders the "home.html" template using render_template.
  - **About Route ("/about"):** When a user navigates to the "/about" URL, the about() function is triggered. This function renders the "about.html" template using render_template.
- **Run the App:** The program runs the Flask application in debug mode, allowing for easier debugging.

Files Required:

- **app.py:** Contains the Flask application code.
- **pages/home.html:** HTML template for the home page.
- **pages/about.html:** HTML template for the about page.