ACE Engineering College

Department of CSE(AI&ML)

Machine Learning Lab Manual

III CSM A&B

List of Experiments

Week-1

1. Write a python program to compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation

Aim: To compute Central Tendency Measures: Mean, Median, Mode Measure of Dispersion: Variance, Standard Deviation

Description:

Mean: The mean, often referred to as the average, is calculated by summing up all the values in a dataset and dividing by the total number of values. It provides a measure of the "center" of the data.

Median: The median is the middle value of a dataset when it is sorted in ascending or descending order. If there is an odd number of observations, the median is the middle value. If there is an even number of observations, the median is the average of the two middle values.

Mode: The mode is the value that appears most frequently in a dataset. A dataset can have one mode, more than one mode (multimodal), or no mode (if all values occur with the same frequency).

Variance: Variance is a measure of how spread out a set of data points is.

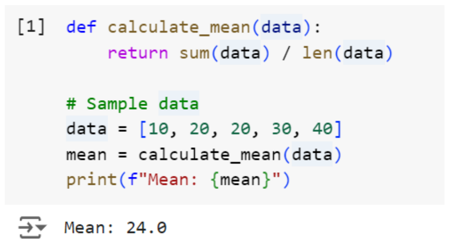
To calculate variance:

1. Find the mean (average) of the data points.
2. Subtract the mean from each data point and square the result.
3. Take the average of these squared differences.

Standard deviation: Standard Deviation is the square root of the variance. It provides a measure of the average distance between each data point and the mean. A higher standard deviation indicates more spread-out data.

Programs:

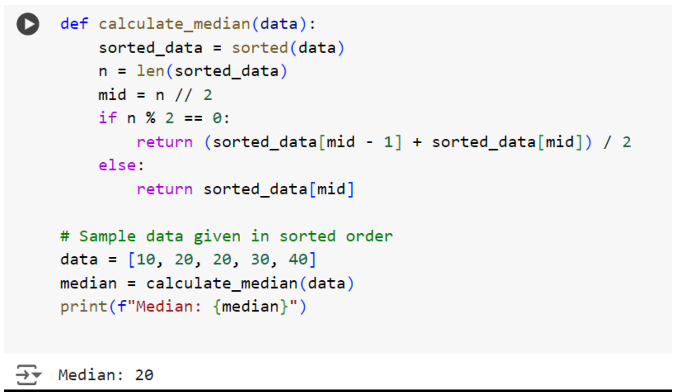
Mean:



```
[1] def calculate_mean(data):
        return sum(data) / len(data)

    # Sample data
    data = [10, 20, 20, 30, 40]
    mean = calculate_mean(data)
    print(f"Mean: {mean}")
```

```
Mean: 24.0
```

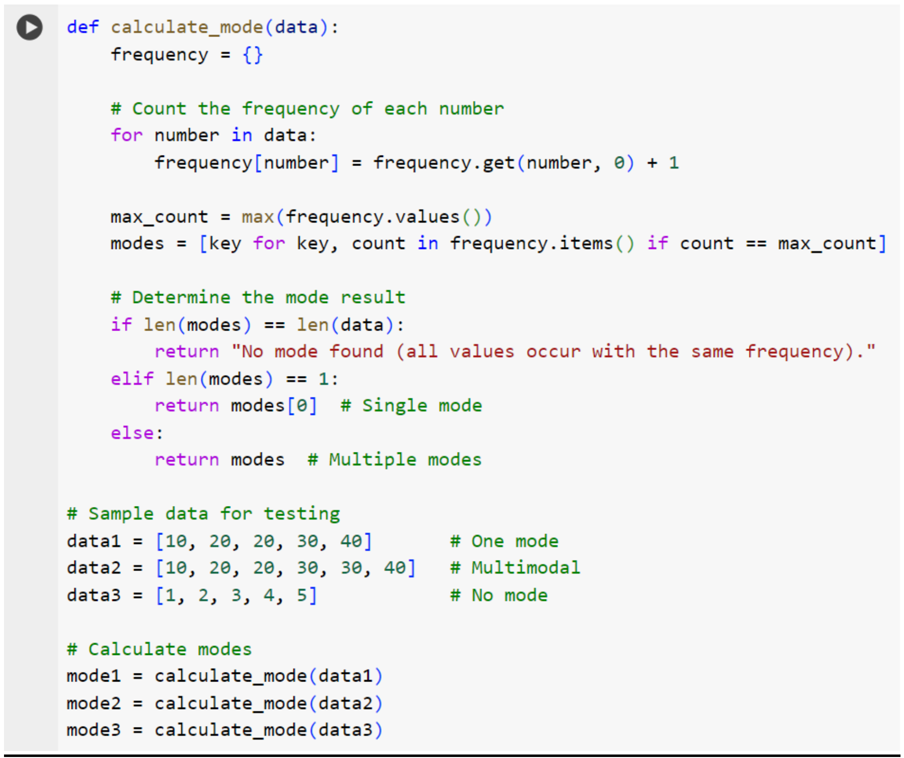
Median:



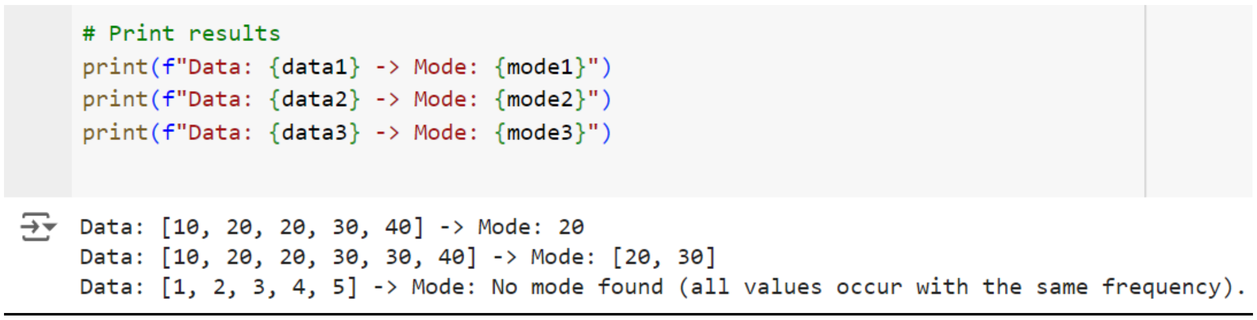
```
def calculate_median(data):
    sorted_data = sorted(data)
    n = len(sorted_data)
    mid = n // 2
    if n % 2 == 0:
        return (sorted_data[mid - 1] + sorted_data[mid]) / 2
    else:
        return sorted_data[mid]

# Sample data given in sorted order
data = [10, 20, 20, 30, 40]
median = calculate_median(data)
print(f"Median: {median}")
```

```
Median: 20
```

Mode:



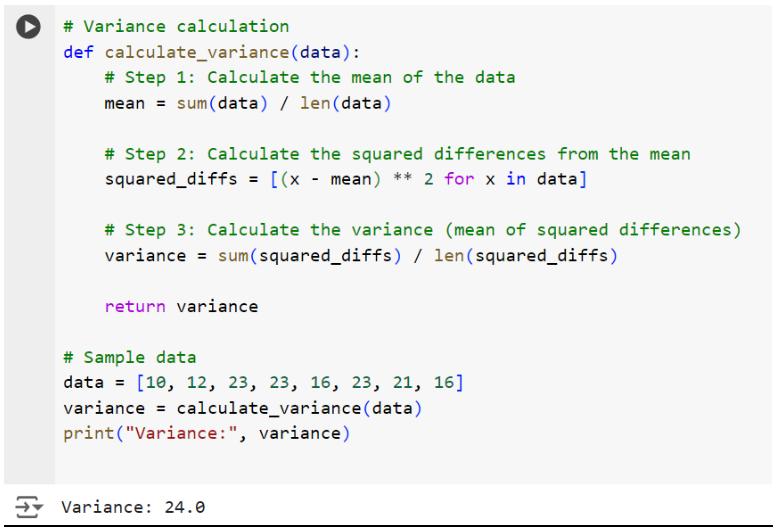
```
def calculate_mode(data):
    frequency = {}

    # Count the frequency of each number
    for number in data:
        frequency[number] = frequency.get(number, 0) + 1

    max_count = max(frequency.values())
    modes = [key for key, count in frequency.items() if count == max_count]

    # Determine the mode result
    if len(modes) == len(data):
        return "No mode found (all values occur with the same frequency)."
    elif len(modes) == 1:
        return modes[0]  # Single mode
    else:
        return modes  # Multiple modes

# Sample data for testing
data1 = [10, 20, 20, 30, 40]       # One mode
data2 = [10, 20, 20, 30, 30, 40]   # Multimodal
data3 = [1, 2, 3, 4, 5]            # No mode

# Calculate modes
mode1 = calculate_mode(data1)
mode2 = calculate_mode(data2)
mode3 = calculate_mode(data3)
```



```python
# Print results
print(f"Data: {data1} -> Mode: {mode1}")
print(f"Data: {data2} -> Mode: {mode2}")
print(f"Data: {data3} -> Mode: {mode3}")
```

```
Data: [10, 20, 20, 30, 40] -> Mode: 20
Data: [10, 20, 20, 30, 30, 40] -> Mode: [20, 30]
Data: [1, 2, 3, 4, 5] -> Mode: No mode found (all values occur with the same frequency).
```

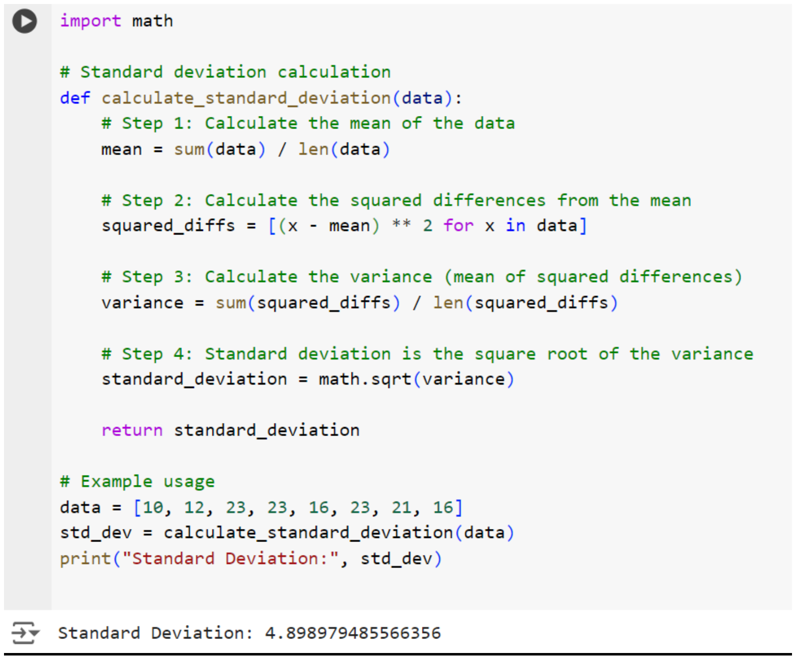
Variance:



```python
# Variance calculation
def calculate_variance(data):
    # Step 1: Calculate the mean of the data
    mean = sum(data) / len(data)

    # Step 2: Calculate the squared differences from the mean
    squared_diffs = [(x - mean) ** 2 for x in data]

    # Step 3: Calculate the variance (mean of squared differences)
    variance = sum(squared_diffs) / len(squared_diffs)

    return variance

# Sample data
data = [10, 12, 23, 23, 16, 23, 21, 16]
variance = calculate_variance(data)
print("Variance:", variance)
```

```
Variance: 24.0
```

Standard Deviation:



```python
import math

# Standard deviation calculation
def calculate_standard_deviation(data):
    # Step 1: Calculate the mean of the data
    mean = sum(data) / len(data)

    # Step 2: Calculate the squared differences from the mean
    squared_diffs = [(x - mean) ** 2 for x in data]

    # Step 3: Calculate the variance (mean of squared differences)
    variance = sum(squared_diffs) / len(squared_diffs)

    # Step 4: Standard deviation is the square root of the variance
    standard_deviation = math.sqrt(variance)

    return standard_deviation

# Example usage
data = [10, 12, 23, 23, 16, 23, 21, 16]
std_dev = calculate_standard_deviation(data)
print("Standard Deviation:", std_dev)
```

```
Standard Deviation: 4.898979485566356
```

Week2:

Study of Python Basic Libraries such as Statistics, Math, Numpy and Scipy

Aim:

To write a python program to Study various Python Basic Libraries such as Statistics, Math, NumPy and SciPy

Description:

A Python library is a collection of modules and packages that provide reusable code to perform specific tasks.

Libraries are generally structured as:

- **Modules:** Individual files containing Python code (functions, classes, etc.) that provide specific functionalities.
- **Packages:** Directories that can contain multiple modules and sub-packages, organized in a hierarchical manner.

Types of Python Libraries

1. Standard Libraries:

Python's Standard Library is a collection of modules and packages that come pre-installed with Python. They provide a wide range of functionalities, including file handling, mathematical operations, and data manipulation. Ex: math, os, sys, datetime, pip, random, csv, statistics, re

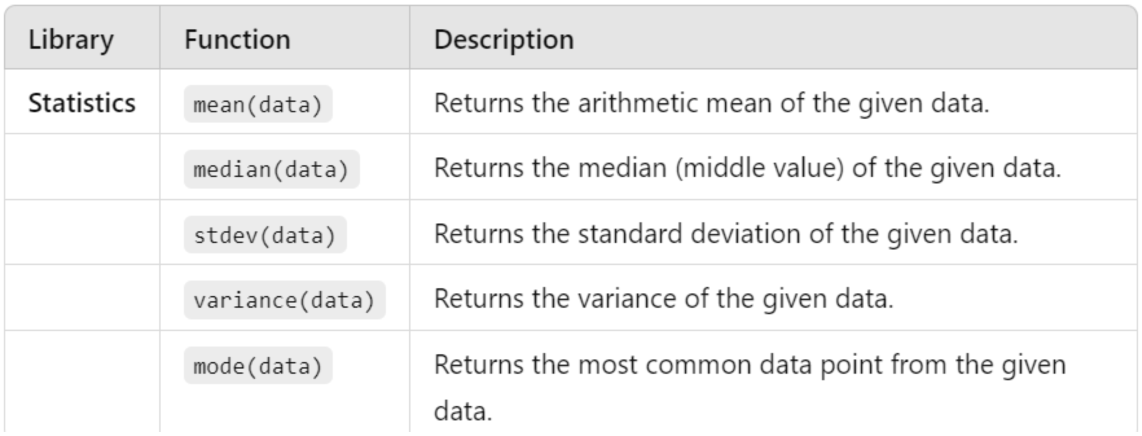
2. Other Libraries:

These are not included with the standard Python installation. They can be installed via package managers like pip and offer extended functionalities.

Ex: NumPy, Pandas, SciPy, Matplotlib, Scikit-learn, Seaborn, TensorFlow, Keras, Pytorch, NLTK, spaCy,

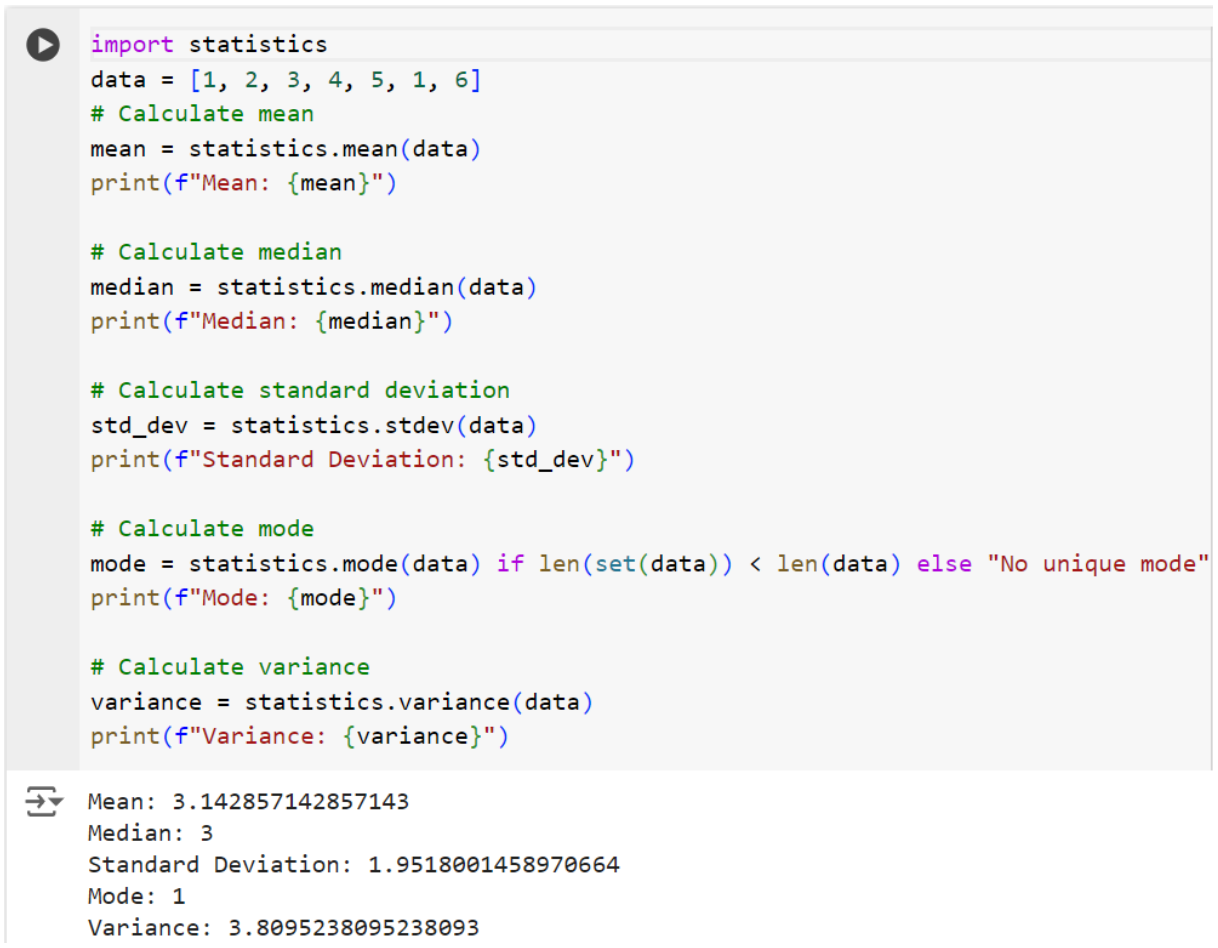
Some of the basic python libraries:

<u>Statistics:</u> This library provides functions for calculating basic statistical measures such as mean, median, mode, variance, and standard deviation.



Library	Function	Description
Statistics	`mean(data)`	Returns the arithmetic mean of the given data.
	`median(data)`	Returns the median (middle value) of the given data.
	`stdev(data)`	Returns the standard deviation of the given data.
	`variance(data)`	Returns the variance of the given data.
	`mode(data)`	Returns the most common data point from the given data.

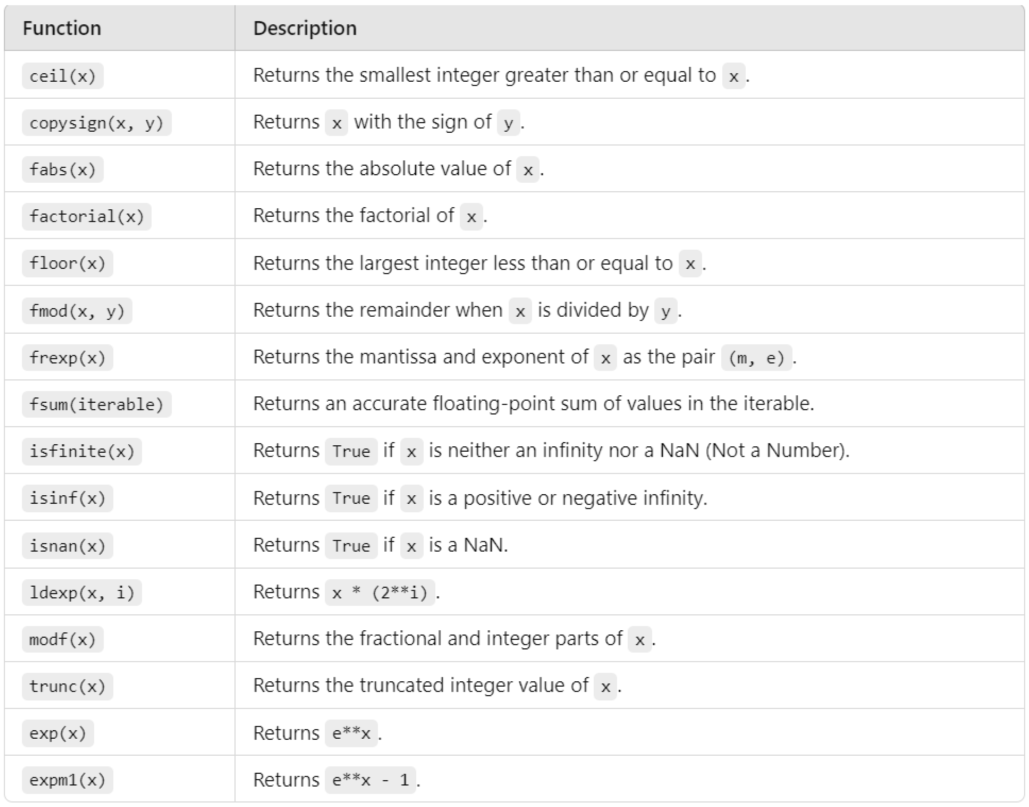
Program:



```python
import statistics
data = [1, 2, 3, 4, 5, 1, 6]
# Calculate mean
mean = statistics.mean(data)
print(f"Mean: {mean}")

# Calculate median
median = statistics.median(data)
print(f"Median: {median}")

# Calculate standard deviation
std_dev = statistics.stdev(data)
print(f"Standard Deviation: {std_dev}")

# Calculate mode
mode = statistics.mode(data) if len(set(data)) < len(data) else "No unique mode"
print(f"Mode: {mode}")

# Calculate variance
variance = statistics.variance(data)
print(f"Variance: {variance}")
```

```
Mean: 3.142857142857143
Median: 3
Standard Deviation: 1.9518001458970664
Mode: 1
Variance: 3.8095238095238093
```

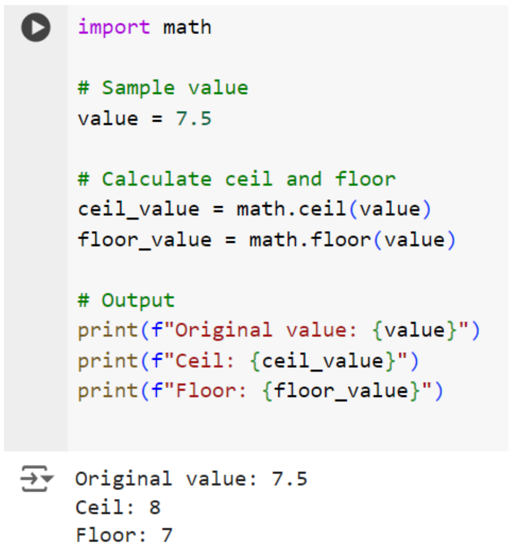
math: The math library offers mathematical functions for operations like square root, factorial, logarithm, and trigonometric functions.



Function	Description
`ceil(x)`	Returns the smallest integer greater than or equal to `x`.
`copysign(x, y)`	Returns `x` with the sign of `y`.
`fabs(x)`	Returns the absolute value of `x`.
`factorial(x)`	Returns the factorial of `x`.
`floor(x)`	Returns the largest integer less than or equal to `x`.
`fmod(x, y)`	Returns the remainder when `x` is divided by `y`.
`frexp(x)`	Returns the mantissa and exponent of `x` as the pair `(m, e)`.
`fsum(iterable)`	Returns an accurate floating-point sum of values in the iterable.
`isfinite(x)`	Returns `True` if `x` is neither an infinity nor a NaN (Not a Number).
`isinf(x)`	Returns `True` if `x` is a positive or negative infinity.
`isnan(x)`	Returns `True` if `x` is a NaN.
`ldexp(x, i)`	Returns `x * (2**i)`.
`modf(x)`	Returns the fractional and integer parts of `x`.
`trunc(x)`	Returns the truncated integer value of `x`.
`exp(x)`	Returns `e**x`.
`expm1(x)`	Returns `e**x - 1`.

Programs on math library:

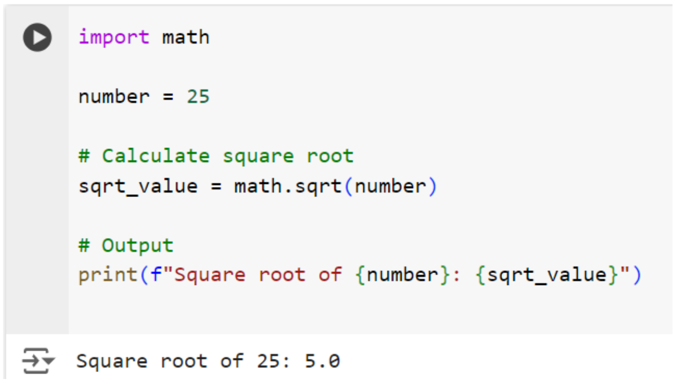
1. Program illustrating ceil and floor



```python
import math

# Sample value
value = 7.5

# Calculate ceil and floor
ceil_value = math.ceil(value)
floor_value = math.floor(value)

# Output
print(f"Original value: {value}")
print(f"Ceil: {ceil_value}")
print(f"Floor: {floor_value}")
```

```
Original value: 7.5
Ceil: 8
Floor: 7
```

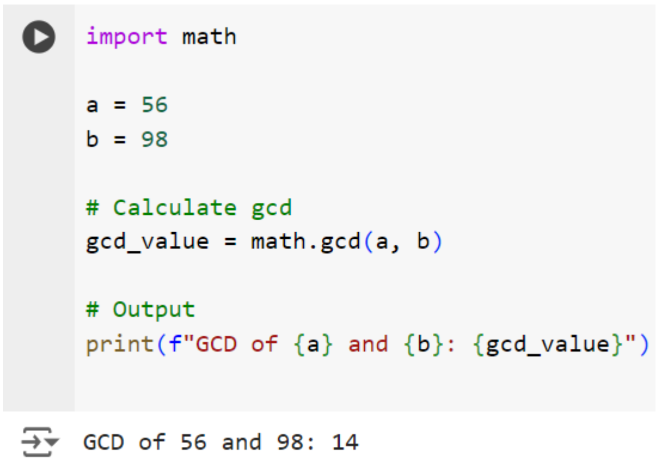
2. Program to find the square root of a given number



```python
import math

number = 25

# Calculate square root
sqrt_value = math.sqrt(number)

# Output
print(f"Square root of {number}: {sqrt_value}")
```

```
Square root of 25: 5.0
```

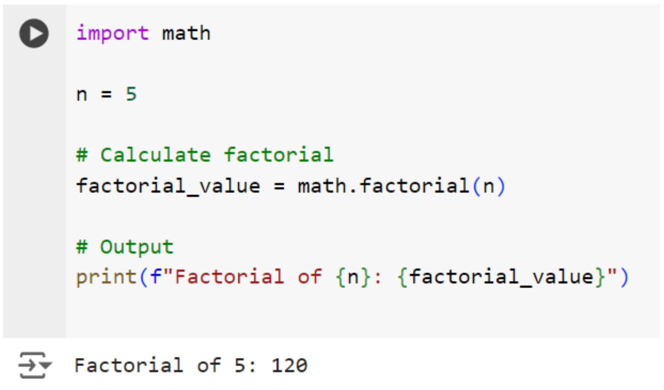
3. Program to find the gcd of 2 given numbers



```python
import math

a = 56
b = 98

# Calculate gcd
gcd_value = math.gcd(a, b)

# Output
print(f"GCD of {a} and {b}: {gcd_value}")
```

```
GCD of 56 and 98: 14
```

4. Program to find the factorial of a given number



```python
import math

n = 5

# Calculate factorial
factorial_value = math.factorial(n)

# Output
print(f"Factorial of {n}: {factorial_value}")
```

```
Factorial of 5: 120
```

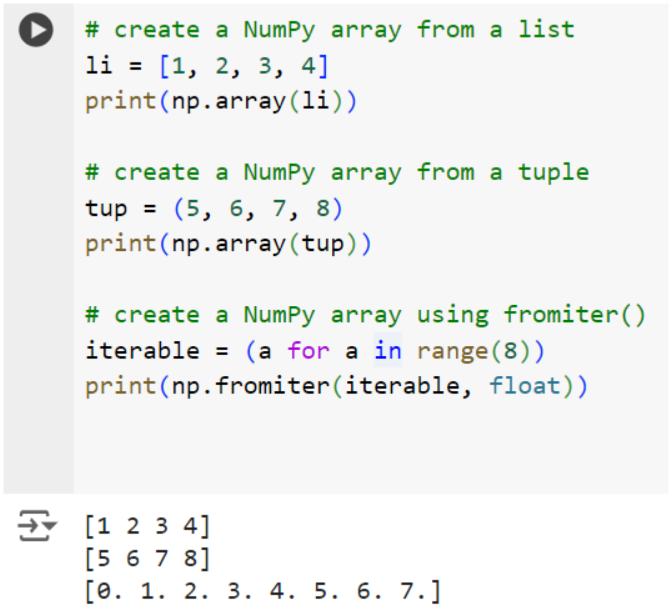
NumPy: NumPy stands for Numerical Python. NumPy is an open-source library in Python designed for mathematical, scientific, engineering, and data science applications. It provides an extensive set of mathematical and statistical functions that facilitate complex calculations. NumPy excels in handling N-dimensional arrays, linear algebra, random number, Fourier transform, etc.

NumPy can be integrated with C/C++ and Fortran, allowing for optimized performance in scientific applications.

Libraries like TensorFlow and Scikit-learn utilize NumPy arrays for efficient matrix computations.

<u>Programs</u>

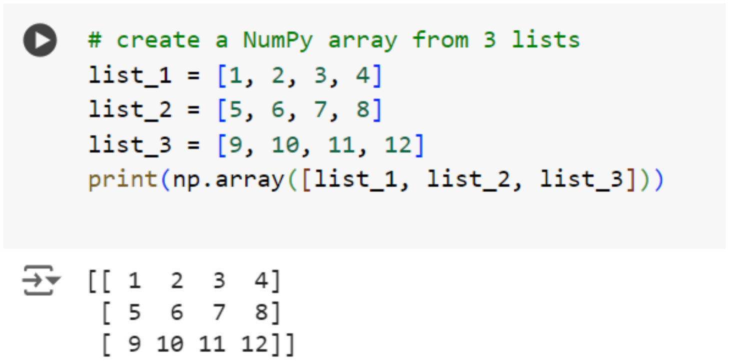
1. Creating arrays



```python
# create a NumPy array from a list
li = [1, 2, 3, 4]
print(np.array(li))

# create a NumPy array from a tuple
tup = (5, 6, 7, 8)
print(np.array(tup))

# create a NumPy array using fromiter()
iterable = (a for a in range(8))
print(np.fromiter(iterable, float))
```

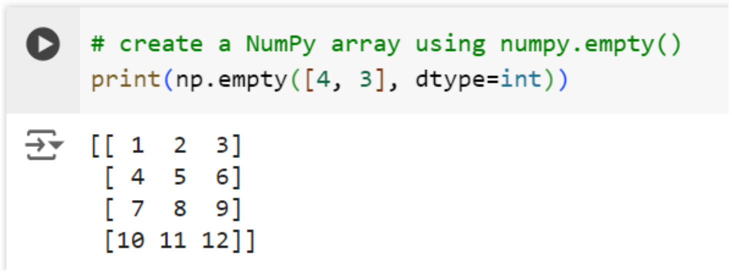
```
[1 2 3 4]
[5 6 7 8]
[0. 1. 2. 3. 4. 5. 6. 7.]
```

2. Creating Multi-Dimensional Array



```python
# create a NumPy array from 3 lists
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
list_3 = [9, 10, 11, 12]
print(np.array([list_1, list_2, list_3]))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

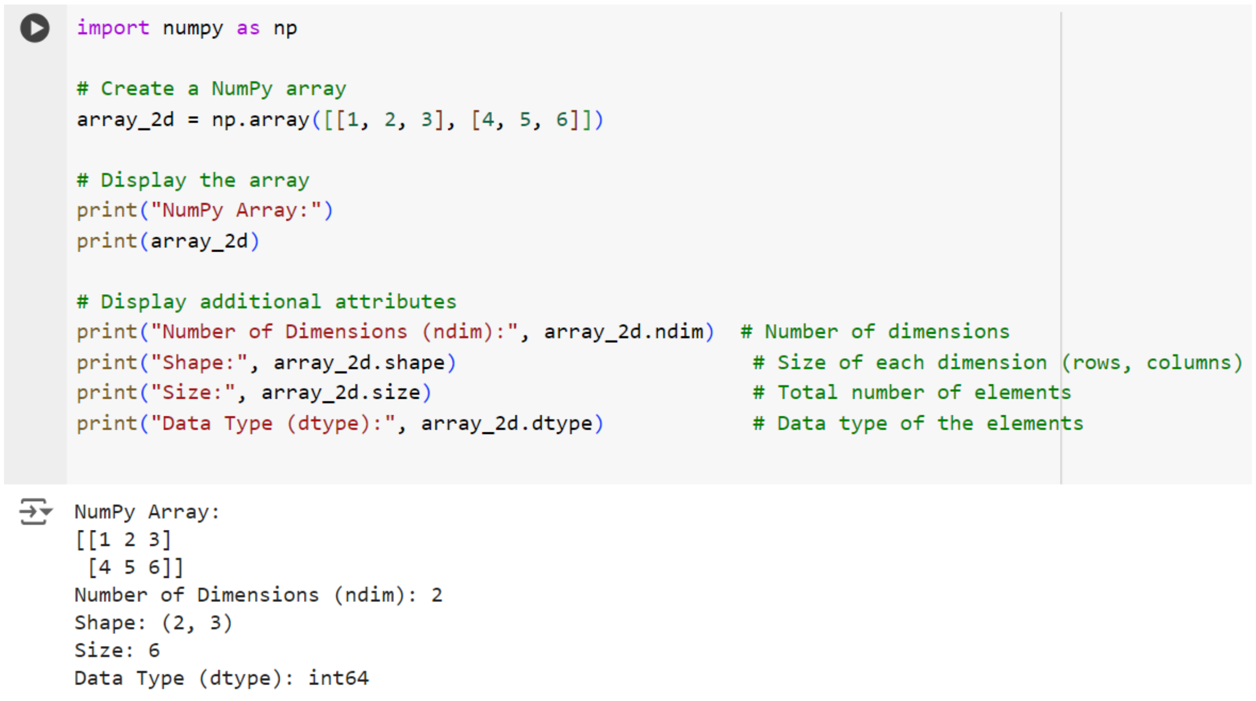


```python
# create a NumPy array using numpy.empty()
print(np.empty([4, 3], dtype=int))
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

The `np.empty()` function is used to create an uninitialized array. This means that the contents of the array are random and based on whatever values already exist in that memory location.

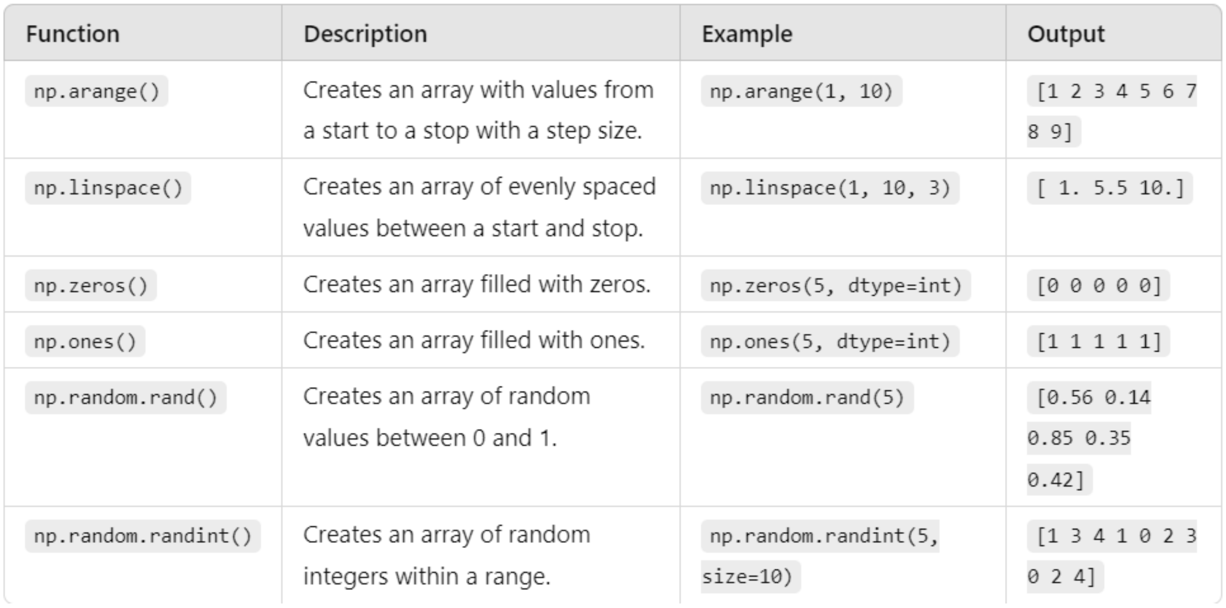
3. Displaying additional attributes



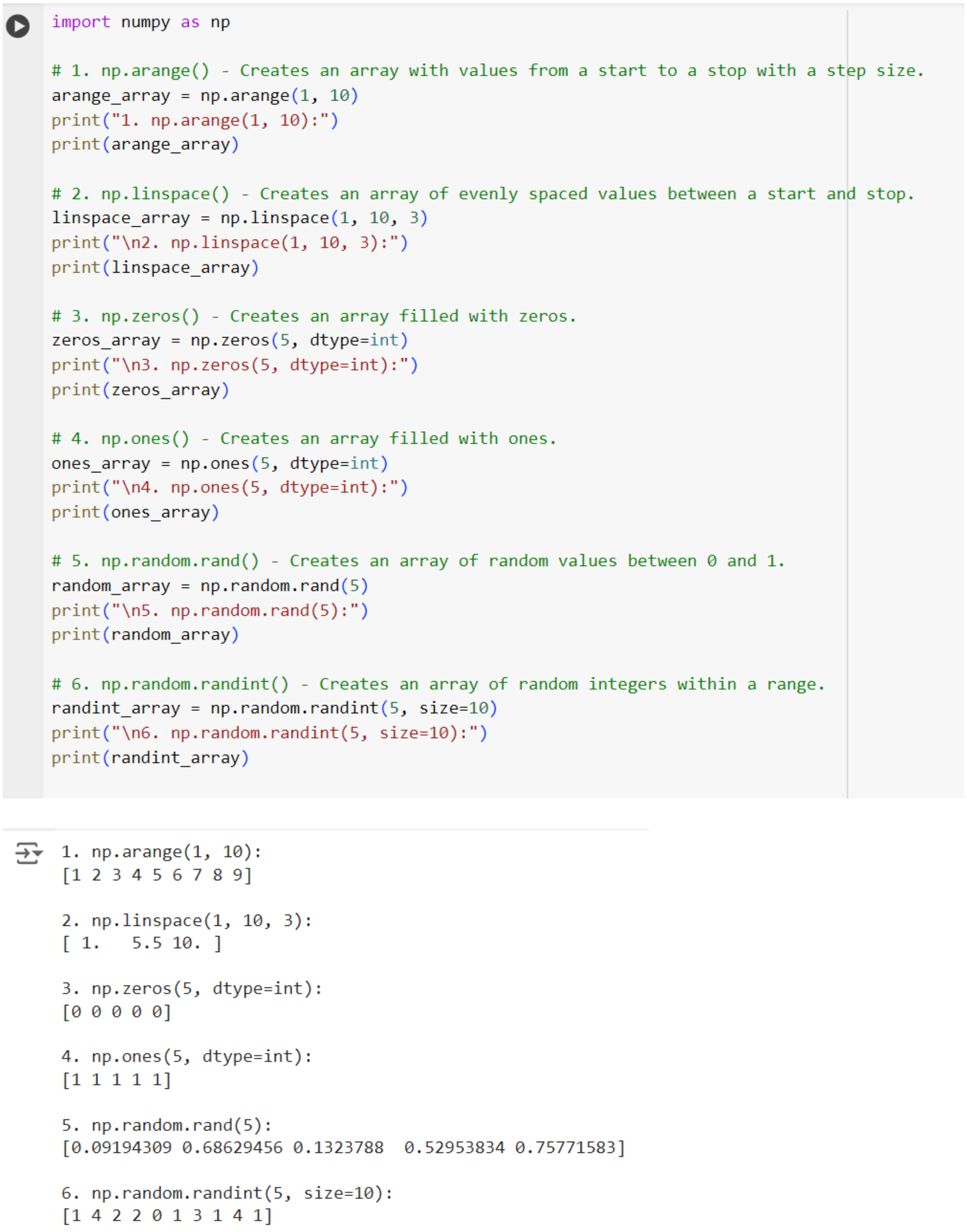
```python
import numpy as np

# Create a NumPy array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Display the array
print("NumPy Array:")
print(array_2d)

# Display additional attributes
print("Number of Dimensions (ndim):", array_2d.ndim)  # Number of dimensions
print("Shape:", array_2d.shape)                        # Size of each dimension (rows, columns)
print("Size:", array_2d.size)                          # Total number of elements
print("Data Type (dtype):", array_2d.dtype)            # Data type of the elements
```

```
NumPy Array:
[[1 2 3]
 [4 5 6]]
Number of Dimensions (ndim): 2
Shape: (2, 3)
Size: 6
Data Type (dtype): int64
```

4. NumPy functions for initializing placeholders for 1D arrays



Function	Description	Example	Output
`np.arange()`	Creates an array with values from a start to a stop with a step size.	`np.arange(1, 10)`	`[1 2 3 4 5 6 7 8 9]`
`np.linspace()`	Creates an array of evenly spaced values between a start and stop.	`np.linspace(1, 10, 3)`	`[ 1. 5.5 10.]`
`np.zeros()`	Creates an array filled with zeros.	`np.zeros(5, dtype=int)`	`[0 0 0 0 0]`
`np.ones()`	Creates an array filled with ones.	`np.ones(5, dtype=int)`	`[1 1 1 1 1]`
`np.random.rand()`	Creates an array of random values between 0 and 1.	`np.random.rand(5)`	`[0.56 0.14 0.85 0.35 0.42]`
`np.random.randint()`	Creates an array of random integers within a range.	`np.random.randint(5, size=10)`	`[1 3 4 1 0 2 3 0 2 4]`

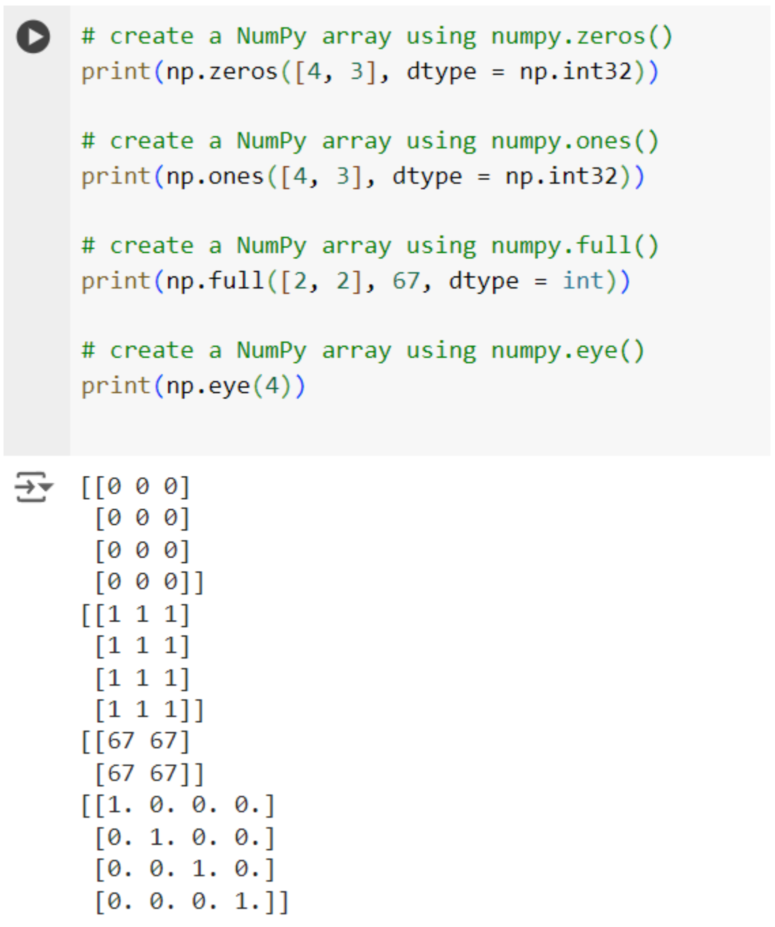


```python
import numpy as np

# 1. np.arange() - Creates an array with values from a start to a stop with a step size.
arange_array = np.arange(1, 10)
print("1. np.arange(1, 10):")
print(arange_array)

# 2. np.linspace() - Creates an array of evenly spaced values between a start and stop.
linspace_array = np.linspace(1, 10, 3)
print("\n2. np.linspace(1, 10, 3):")
print(linspace_array)

# 3. np.zeros() - Creates an array filled with zeros.
zeros_array = np.zeros(5, dtype=int)
print("\n3. np.zeros(5, dtype=int):")
print(zeros_array)

# 4. np.ones() - Creates an array filled with ones.
ones_array = np.ones(5, dtype=int)
print("\n4. np.ones(5, dtype=int):")
print(ones_array)

# 5. np.random.rand() - Creates an array of random values between 0 and 1.
random_array = np.random.rand(5)
print("\n5. np.random.rand(5):")
print(random_array)

# 6. np.random.randint() - Creates an array of random integers within a range.
randint_array = np.random.randint(5, size=10)
print("\n6. np.random.randint(5, size=10):")
print(randint_array)
```

```
1. np.arange(1, 10):
[1 2 3 4 5 6 7 8 9]

2. np.linspace(1, 10, 3):
[ 1.   5.5 10. ]

3. np.zeros(5, dtype=int):
[0 0 0 0 0]

4. np.ones(5, dtype=int):
[1 1 1 1 1]

5. np.random.rand(5):
[0.09194309 0.68629456 0.1323788  0.52953834 0.75771583]

6. np.random.randint(5, size=10):
[1 4 2 2 0 1 3 1 4 1]
```

5. NumPy functions for initializing placeholders for 2D arrays

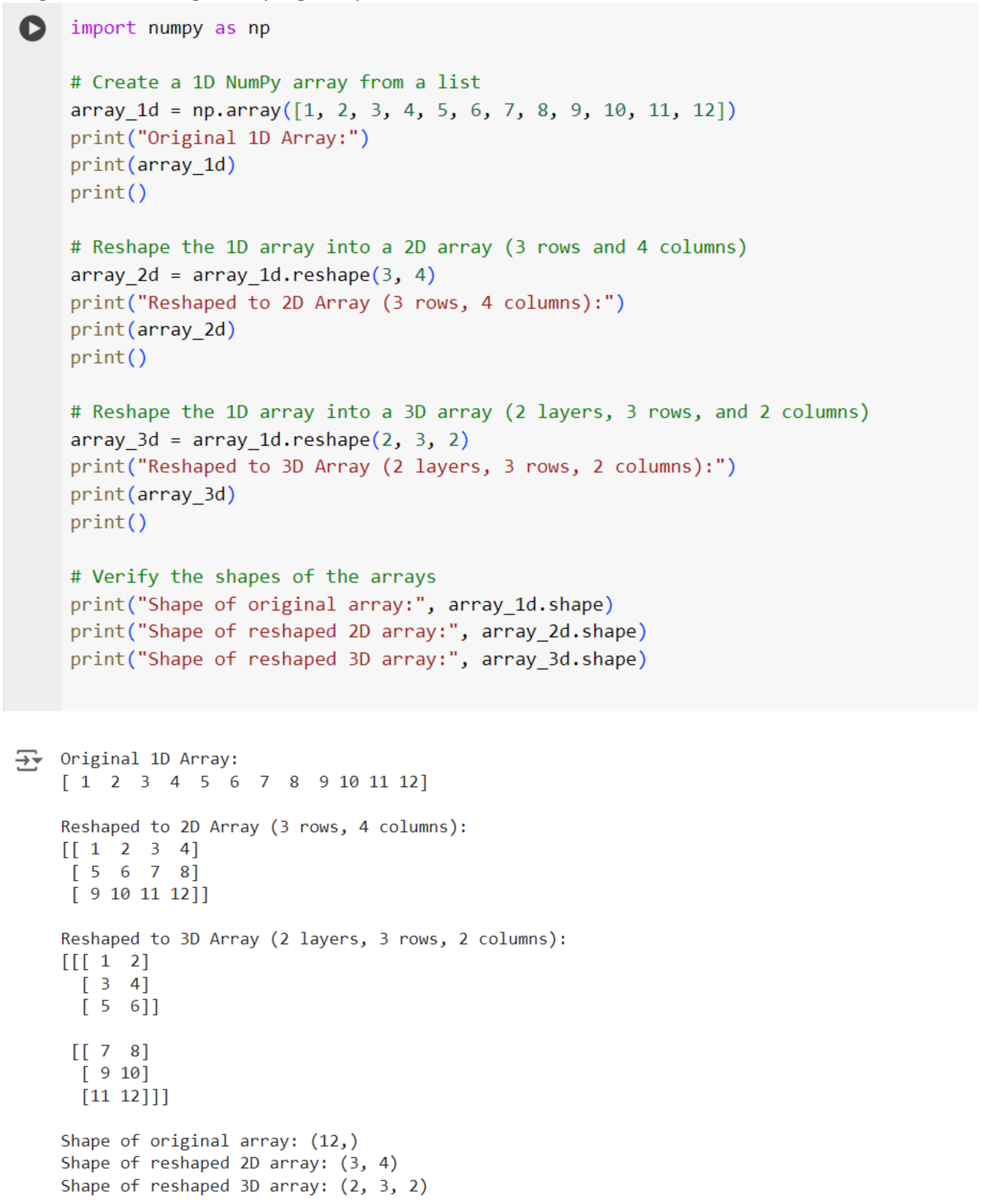
Function	Description	Example	Output Example
`np.zeros(shape, dtype)`	Creates a 2D array filled with zeros.	`np.zeros([4, 3], dtype=np.int32)`	`[[0 0 0] [0 0 0] [0 0 0] [0 0 0]]`
`np.ones(shape, dtype)`	Creates a 2D array filled with ones.	`np.ones([4, 3], dtype=np.int32)`	`[[1 1 1] [1 1 1] [1 1 1] [1 1 1]]`
`np.full(shape, fill_value, dtype)`	Creates a 2D array filled with a specified value.	`np.full([2, 2], 67, dtype=int)`	`[[67 67] [67 67]]`
`np.eye(N)`	Creates a 2D identity matrix of size N x N.	`np.eye(4)`	`[[1. 0. 0. 0.] [0. 1. 0. 0.] [0. 0. 1. 0.] [0. 0. 0. 1.]]`



```python
# create a NumPy array using numpy.zeros()
print(np.zeros([4, 3], dtype = np.int32))

# create a NumPy array using numpy.ones()
print(np.ones([4, 3], dtype = np.int32))

# create a NumPy array using numpy.full()
print(np.full([2, 2], 67, dtype = int))

# create a NumPy array using numpy.eye()
print(np.eye(4))
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
[[1 1 1]
 [1 1 1]
 [1 1 1]
 [1 1 1]]
[[67 67]
 [67 67]]
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

6. Program illustrating reshaping array

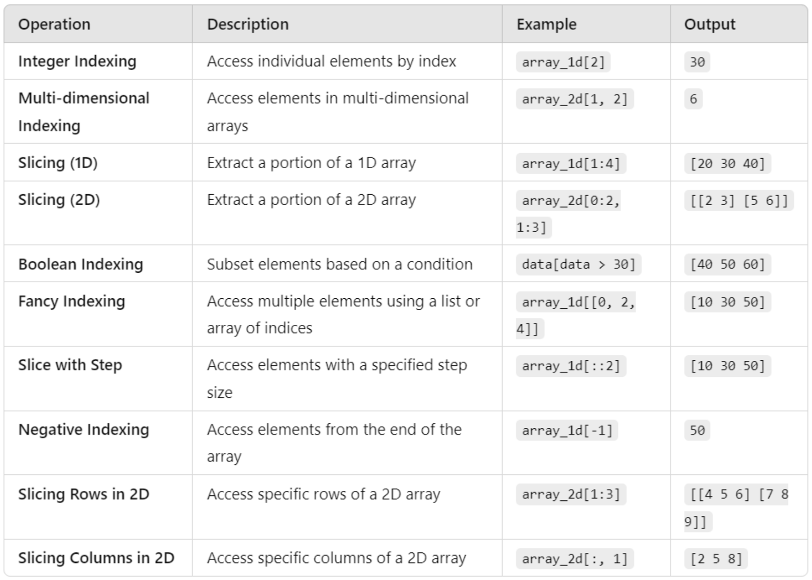


```python
import numpy as np

# Create a 1D NumPy array from a list
array_1d = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
print("Original 1D Array:")
print(array_1d)
print()

# Reshape the 1D array into a 2D array (3 rows and 4 columns)
array_2d = array_1d.reshape(3, 4)
print("Reshaped to 2D Array (3 rows, 4 columns):")
print(array_2d)
print()

# Reshape the 1D array into a 3D array (2 layers, 3 rows, and 2 columns)
array_3d = array_1d.reshape(2, 3, 2)
print("Reshaped to 3D Array (2 layers, 3 rows, 2 columns):")
print(array_3d)
print()

# Verify the shapes of the arrays
print("Shape of original array:", array_1d.shape)
print("Shape of reshaped 2D array:", array_2d.shape)
print("Shape of reshaped 3D array:", array_3d.shape)
```

```
Original 1D Array:
[ 1  2  3  4  5  6  7  8  9 10 11 12]

Reshaped to 2D Array (3 rows, 4 columns):
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

Reshaped to 3D Array (2 layers, 3 rows, 2 columns):
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]

Shape of original array: (12,)
Shape of reshaped 2D array: (3, 4)
Shape of reshaped 3D array: (2, 3, 2)
```

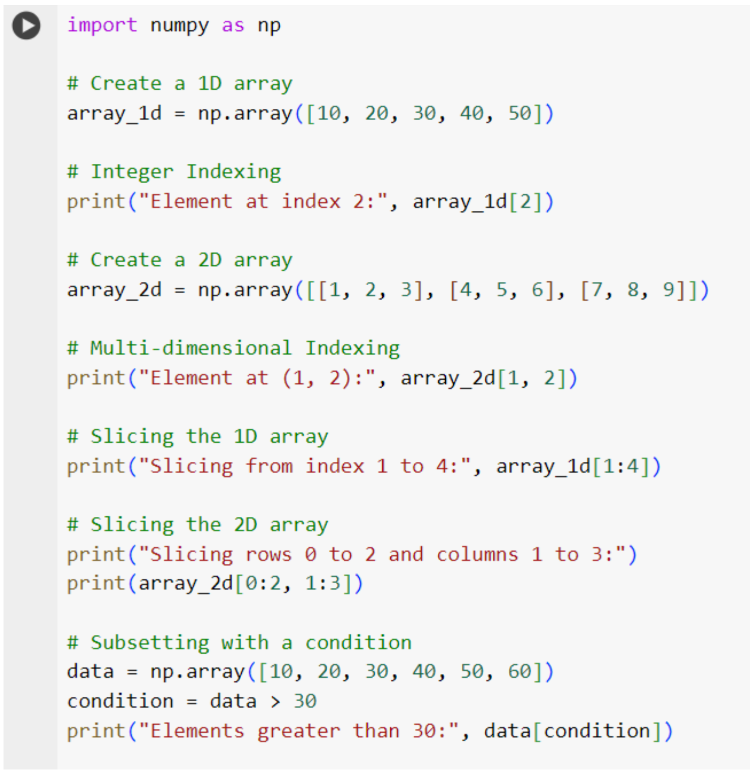
7. Indexing, Slicing and Subsetting functions

array_1d = np.array([10, 20, 30, 40, 50])
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])



Operation	Description	Example	Output
Integer Indexing	Access individual elements by index	`array_1d[2]`	`30`
Multi-dimensional Indexing	Access elements in multi-dimensional arrays	`array_2d[1, 2]`	`6`
Slicing (1D)	Extract a portion of a 1D array	`array_1d[1:4]`	`[20 30 40]`
Slicing (2D)	Extract a portion of a 2D array	`array_2d[0:2, 1:3]`	`[[2 3] [5 6]]`
Boolean Indexing	Subset elements based on a condition	`data[data > 30]`	`[40 50 60]`
Fancy Indexing	Access multiple elements using a list or array of indices	`array_1d[[0, 2, 4]]`	`[10 30 50]`
Slice with Step	Access elements with a specified step size	`array_1d[::2]`	`[10 30 50]`
Negative Indexing	Access elements from the end of the array	`array_1d[-1]`	`50`
Slicing Rows in 2D	Access specific rows of a 2D array	`array_2d[1:3]`	`[[4 5 6] [7 8 9]]`
Slicing Columns in 2D	Access specific columns of a 2D array	`array_2d[:, 1]`	`[2 5 8]`

Program to illustrate indexing, slicing, subsetting



```python
import numpy as np

# Create a 1D array
array_1d = np.array([10, 20, 30, 40, 50])

# Integer Indexing
print("Element at index 2:", array_1d[2])

# Create a 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Multi-dimensional Indexing
print("Element at (1, 2):", array_2d[1, 2])

# Slicing the 1D array
print("Slicing from index 1 to 4:", array_1d[1:4])

# Slicing the 2D array
print("Slicing rows 0 to 2 and columns 1 to 3:")
print(array_2d[0:2, 1:3])

# Subsetting with a condition
data = np.array([10, 20, 30, 40, 50, 60])
condition = data > 30
print("Elements greater than 30:", data[condition])
```

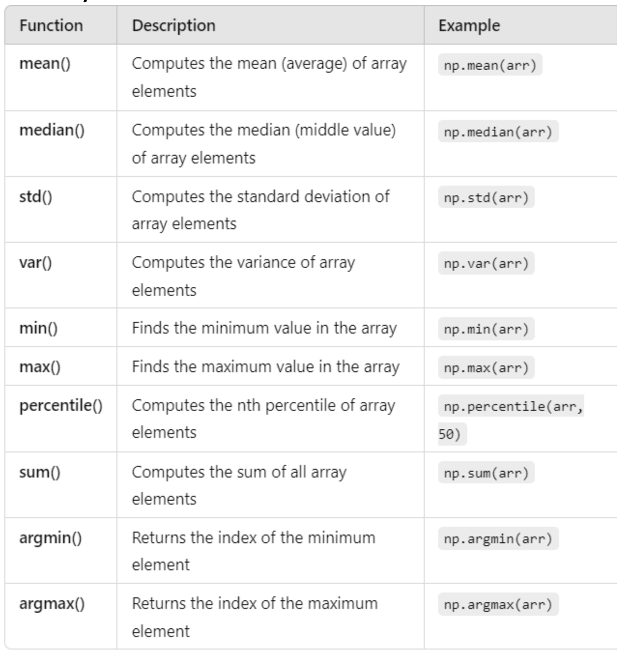
```
Element at index 2: 30
Element at (1, 2): 6
Slicing from index 1 to 4: [20 30 40]
Slicing rows 0 to 2 and columns 1 to 3:
[[2 3]
 [5 6]]
Elements greater than 30: [40 50 60]
```

8. Program to illustrate NumPy arithmetic

```
import numpy as np

# Defining both the matrices
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])

# Performing addition using numpy function
print("Addition:", np.add(a, b))

# Performing subtraction using numpy function
print("Subtraction:", np.subtract(a, b))

# Performing multiplication using numpy function
print("Multiplication:", np.multiply(a, b))

# Performing division using numpy functions
print("Division:", np.divide(a, b))

# Performing mod on two matrices
print("Mod:", np.mod(a, b))

#Performing remainder on two matrices
print("Remainder:", np.remainder(a,b))

# Performing power of two matrices
print("Power:", np.power(a, b))

# Performing Exponentiation
print("Exponentiation:", np.exp(b))
```

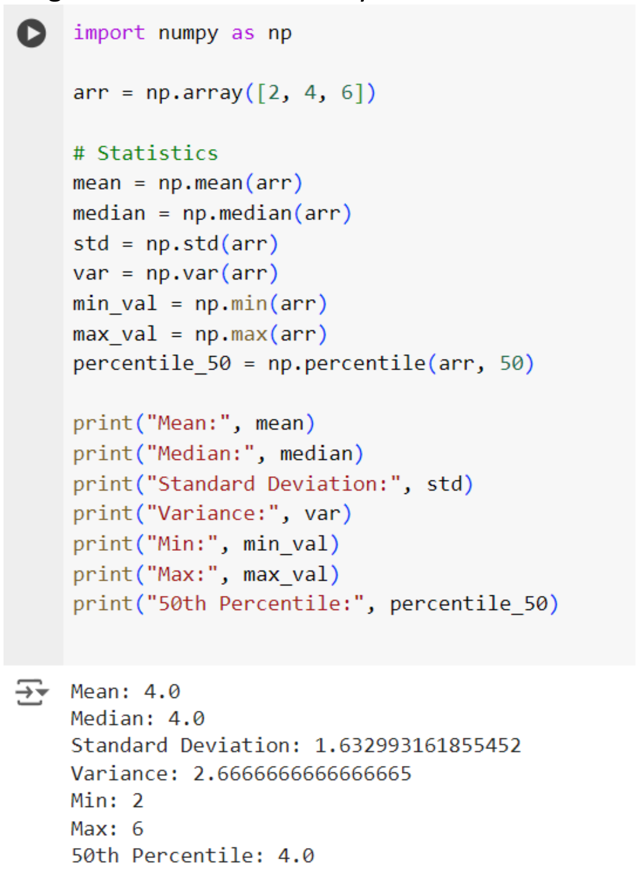
```
Addition: [  7  77  23 130]
Subtraction: [ 3 67  3 70]
Multiplication: [  10  360  130 3000]
Division: [ 2.5        14.4         1.3         3.33333333]
Mod: [ 1  2  3 10]
Remainder: [ 1  2  3 10]
Power: [                 25          1934917632        137858491849
 1152921504606846976]
Exponentiation: [7.38905610e+00 1.48413159e+02 2.20264658e+04 1.06864746e+13]
```

9. NumPy statistics



Function	Description	Example
mean()	Computes the mean (average) of array elements	`np.mean(arr)`
median()	Computes the median (middle value) of array elements	`np.median(arr)`
std()	Computes the standard deviation of array elements	`np.std(arr)`
var()	Computes the variance of array elements	`np.var(arr)`
min()	Finds the minimum value in the array	`np.min(arr)`
max()	Finds the maximum value in the array	`np.max(arr)`
percentile()	Computes the nth percentile of array elements	`np.percentile(arr, 50)`
sum()	Computes the sum of all array elements	`np.sum(arr)`
argmin()	Returns the index of the minimum element	`np.argmin(arr)`
argmax()	Returns the index of the maximum element	`np.argmax(arr)`

Program to illustrate NumPy statistics



```python
import numpy as np

arr = np.array([2, 4, 6])

# Statistics
mean = np.mean(arr)
median = np.median(arr)
std = np.std(arr)
var = np.var(arr)
min_val = np.min(arr)
max_val = np.max(arr)
percentile_50 = np.percentile(arr, 50)

print("Mean:", mean)
print("Median:", median)
print("Standard Deviation:", std)
print("Variance:", var)
print("Min:", min_val)
print("Max:", max_val)
print("50th Percentile:", percentile_50)
```

```
Mean: 4.0
Median: 4.0
Standard Deviation: 1.632993161855452
Variance: 2.6666666666666665
Min: 2
Max: 6
50th Percentile: 4.0
```

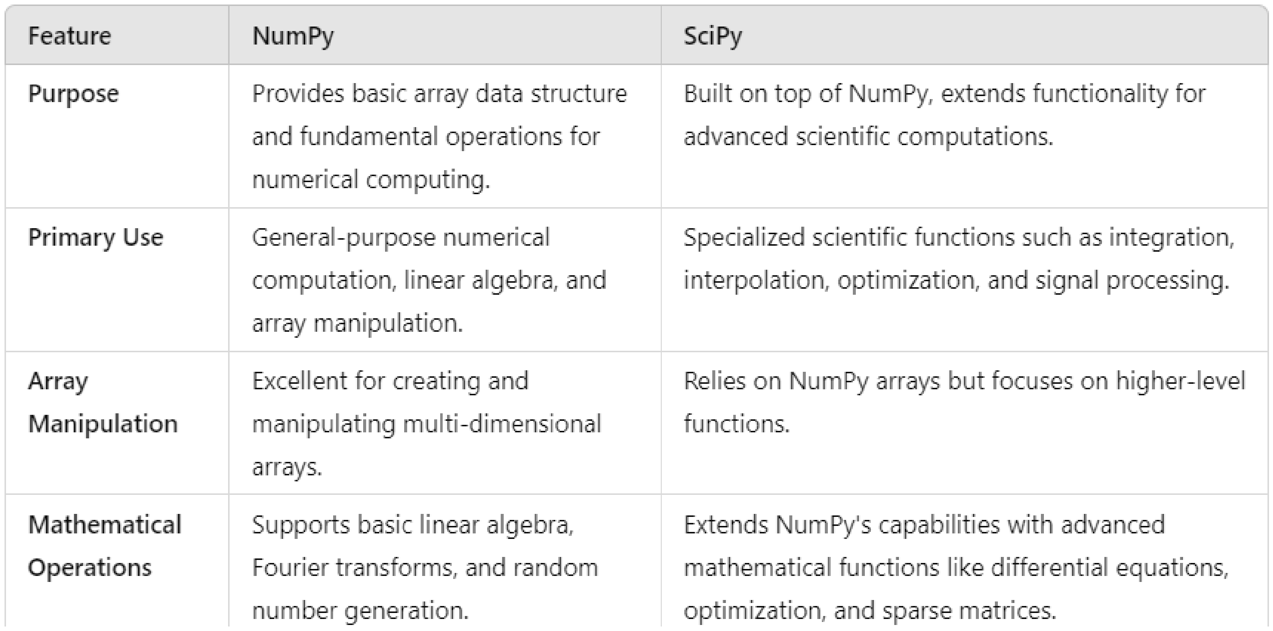
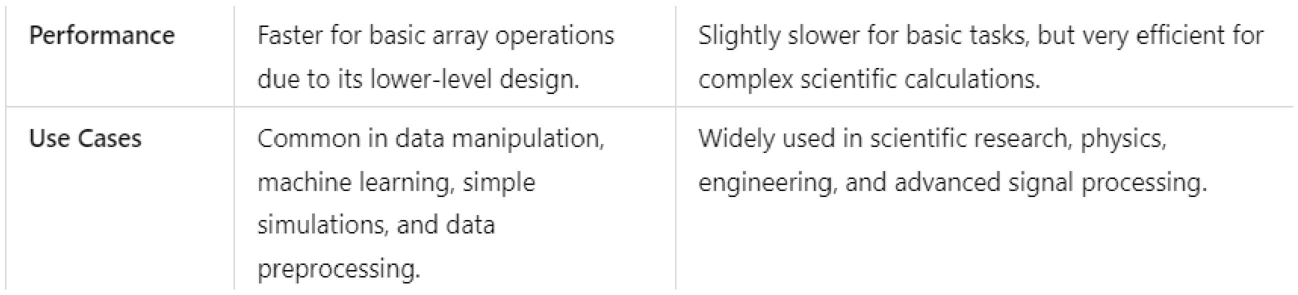
<u>SciPy:</u> SciPy is an Open Source Python-based library, which is used in mathematics, scientific computing, Engineering, and technical computing.

SciPy also pronounced as"SighPi." SciPy contains varieties of sub packages which help to solve the most common issue related to Scientific Computation.

SciPy is the most used Scientific library only second to GNU Scientific Library for C/C++ or Matlab.

Easy to use and understand as well as fast computational power. It can operate on an array of NumPy library.

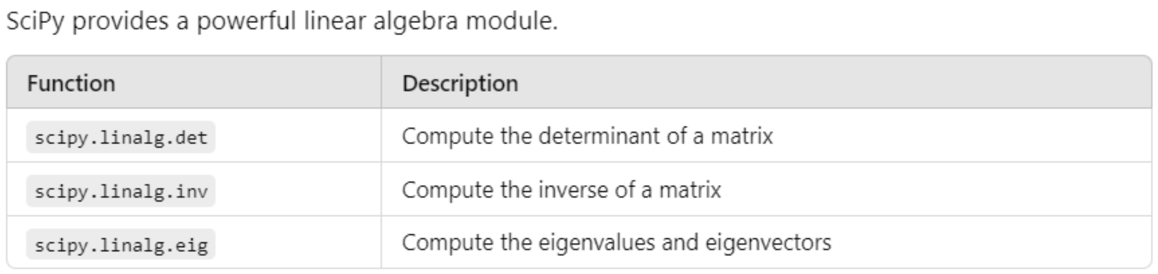
The following are the basic differences between numpy and scipy

Feature	NumPy	SciPy
Purpose	Provides basic array data structure and fundamental operations for numerical computing.	Built on top of NumPy, extends functionality for advanced scientific computations.
Primary Use	General-purpose numerical computation, linear algebra, and array manipulation.	Specialized scientific functions such as integration, interpolation, optimization, and signal processing.
Array Manipulation	Excellent for creating and manipulating multi-dimensional arrays.	Relies on NumPy arrays but focuses on higher-level functions.
Mathematical Operations	Supports basic linear algebra, Fourier transforms, and random number generation.	Extends NumPy's capabilities with advanced mathematical functions like differential equations, optimization, and sparse matrices.
Performance	Faster for basic array operations due to its lower-level design.	Slightly slower for basic tasks, but very efficient for complex scientific calculations.
Use Cases	Common in data manipulation, machine learning, simple simulations, and data preprocessing.	Widely used in scientific research, physics, engineering, and advanced signal processing.

Some of the important features of SciPy across different scientific domains

1. **Linear algebra**

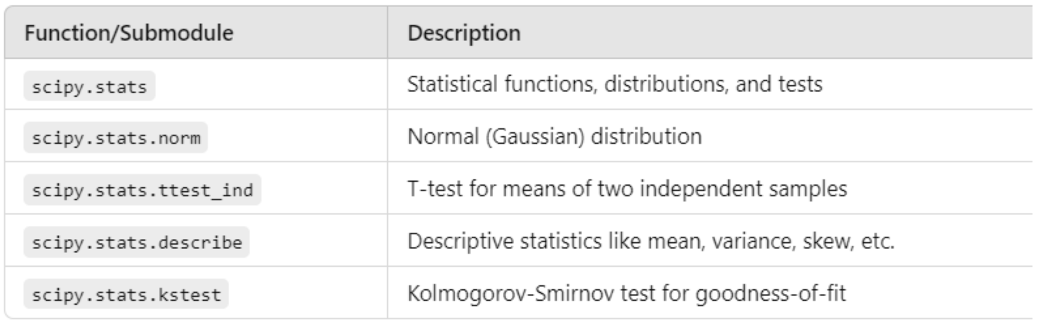


SciPy provides a powerful linear algebra module.

Function	Description
`scipy.linalg.det`	Compute the determinant of a matrix
`scipy.linalg.inv`	Compute the inverse of a matrix
`scipy.linalg.eig`	Compute the eigenvalues and eigenvectors

---

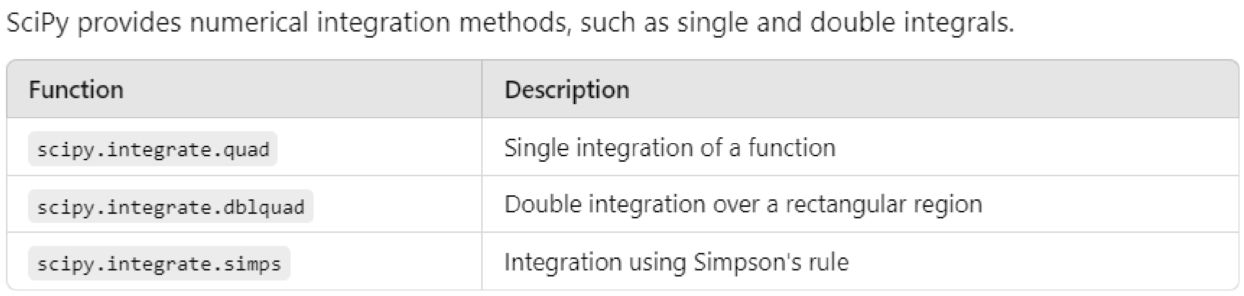
2. **Statistics and probability**



Function/Submodule	Description
`scipy.stats`	Statistical functions, distributions, and tests
`scipy.stats.norm`	Normal (Gaussian) distribution
`scipy.stats.ttest_ind`	T-test for means of two independent samples
`scipy.stats.describe`	Descriptive statistics like mean, variance, skew, etc.
`scipy.stats.kstest`	Kolmogorov-Smirnov test for goodness-of-fit

---

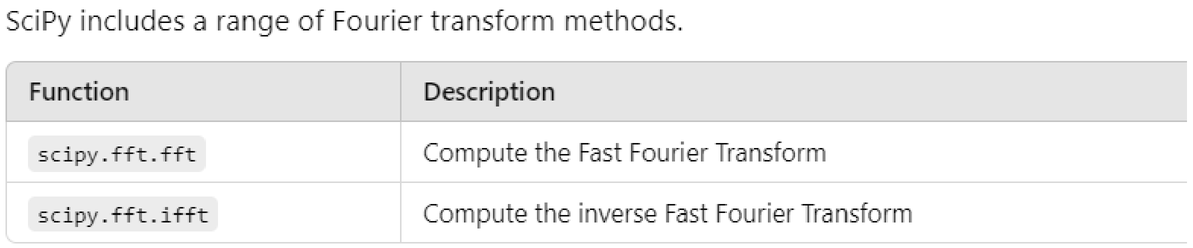
3. **Mathematics and numerical integration**



SciPy provides numerical integration methods, such as single and double integrals.

Function	Description
`scipy.integrate.quad`	Single integration of a function
`scipy.integrate.dblquad`	Double integration over a rectangular region
`scipy.integrate.simps`	Integration using Simpson's rule

4. **Fourier transforms**

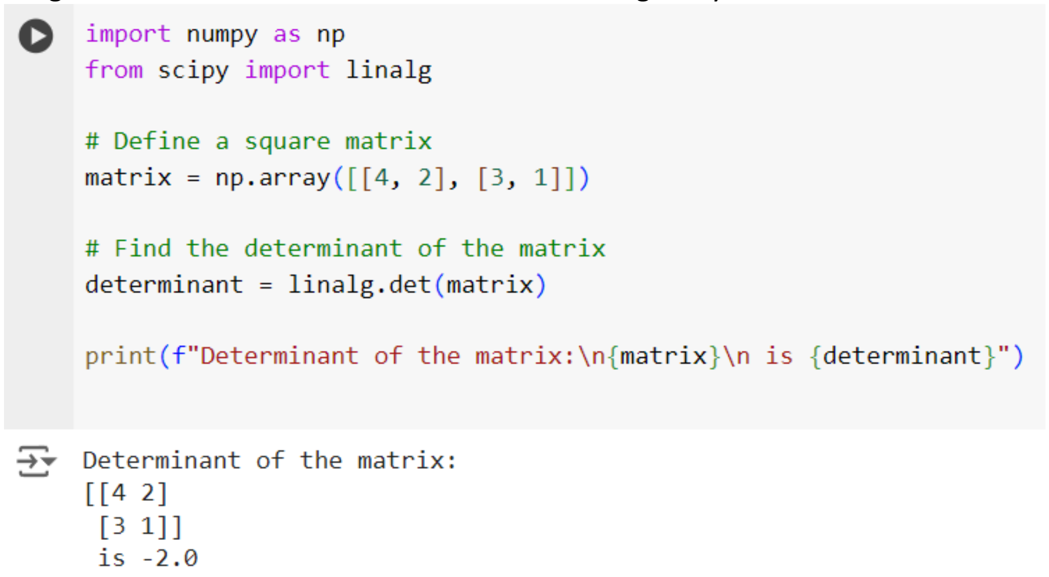


SciPy includes a range of Fourier transform methods.

Function	Description
`scipy.fft.fft`	Compute the Fast Fourier Transform
`scipy.fft.ifft`	Compute the inverse Fast Fourier Transform

Programs

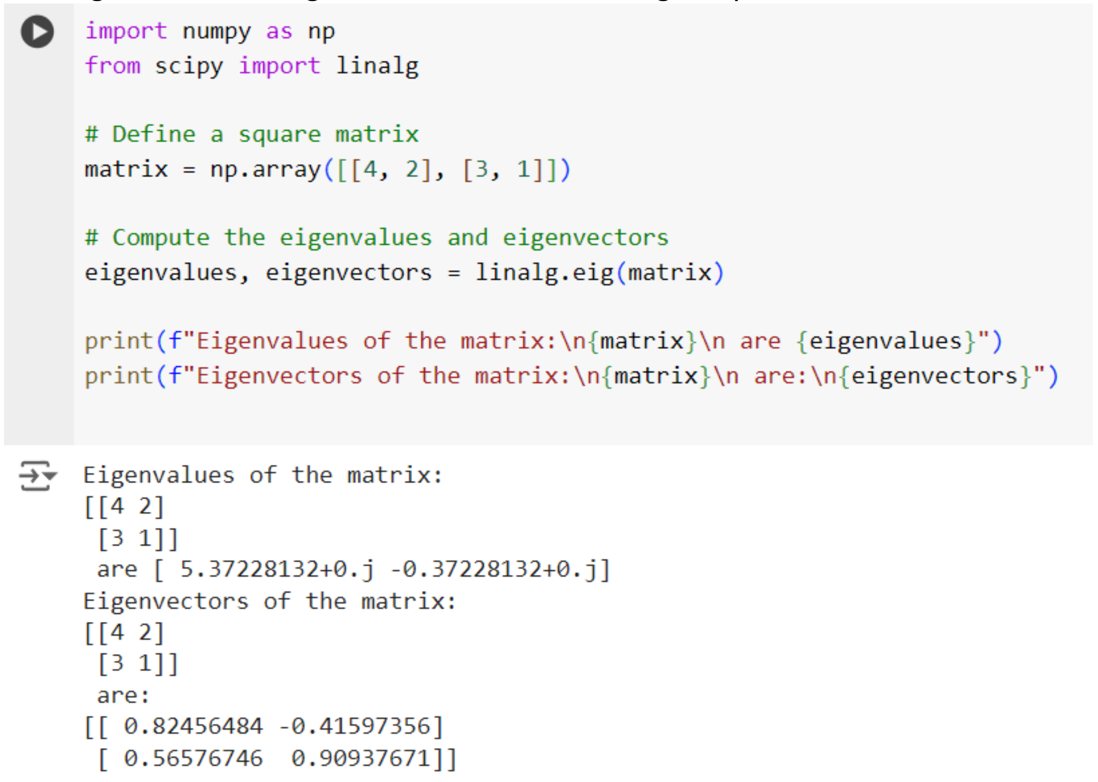
1. Program to find the determinant of a matrix using SciPy



```python
import numpy as np
from scipy import linalg

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

# Find the determinant of the matrix
determinant = linalg.det(matrix)

print(f"Determinant of the matrix:\n{matrix}\n is {determinant}")
```

```
Determinant of the matrix:
[[4 2]
 [3 1]]
 is -2.0
```

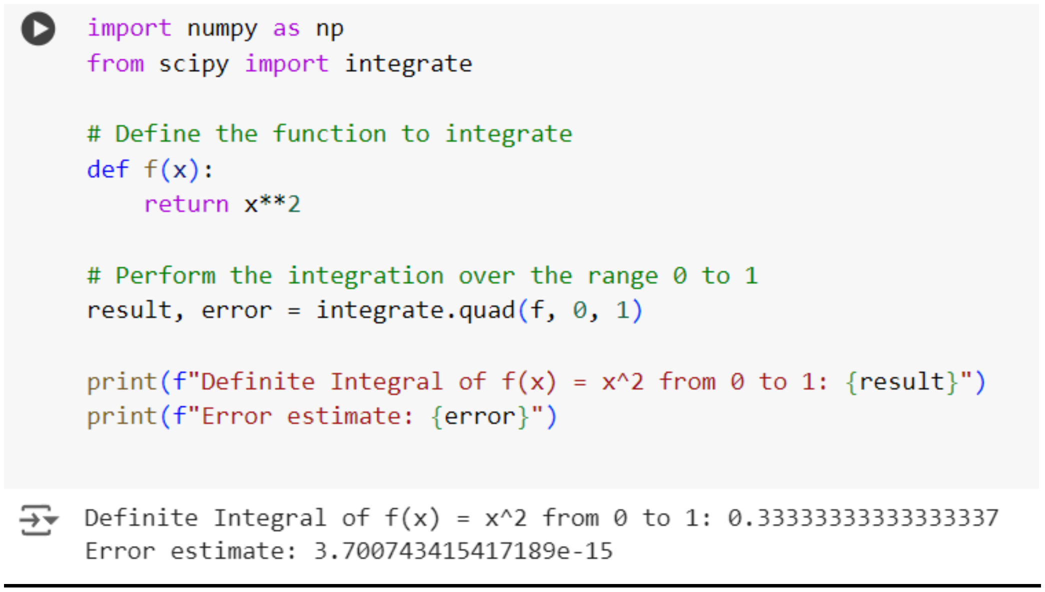
2. Find Eigenvalues and Eigenvectors of a Matrix using SciPy



```python
import numpy as np
from scipy import linalg

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

# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = linalg.eig(matrix)

print(f"Eigenvalues of the matrix:\n{matrix}\n are {eigenvalues}")
print(f"Eigenvectors of the matrix:\n{matrix}\n are:\n{eigenvectors}")
```

```
Eigenvalues of the matrix:
[[4 2]
 [3 1]]
 are [ 5.37228132+0.j -0.37228132+0.j]
Eigenvectors of the matrix:
[[4 2]
 [3 1]]
 are:
[[ 0.82456484 -0.41597356]
 [ 0.56576746  0.90937671]]
```

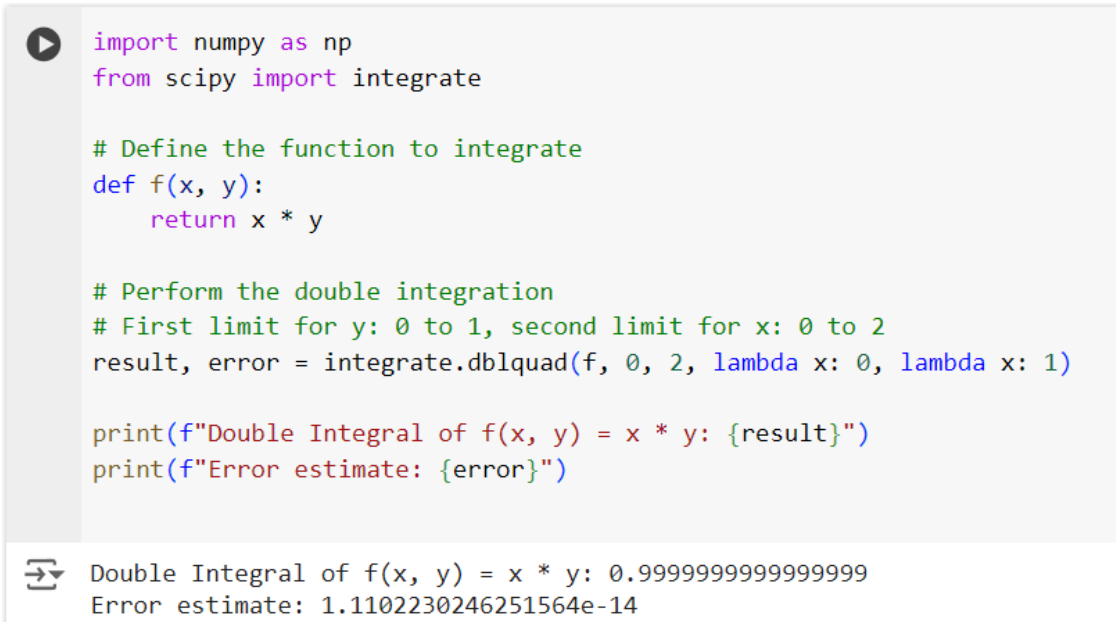
3. Write a python program to calculate the definite integral of the function f(x) = x^2 over the interval [0, 1].



```python
import numpy as np
from scipy import integrate

# Define the function to integrate
def f(x):
    return x**2

# Perform the integration over the range 0 to 1
result, error = integrate.quad(f, 0, 1)

print(f"Definite Integral of f(x) = x^2 from 0 to 1: {result}")
print(f"Error estimate: {error}")
```

```
Definite Integral of f(x) = x^2 from 0 to 1: 0.33333333333333337
Error estimate: 3.700743415417189e-15
```

4. Write a python program to perform a double integration of the function f(x,y)=x·y where: x ranges from 0 to 2 and y ranges from 0 to 1



```python
import numpy as np
from scipy import integrate

# Define the function to integrate
def f(x, y):
    return x * y

# Perform the double integration
# First limit for y: 0 to 1, second limit for x: 0 to 2
result, error = integrate.dblquad(f, 0, 2, lambda x: 0, lambda x: 1)

print(f"Double Integral of f(x, y) = x * y: {result}")
print(f"Error estimate: {error}")
```

```
Double Integral of f(x, y) = x * y: 0.9999999999999999
Error estimate: 1.1102230246251564e-14
```

Week3:

Study of various Python Libraries for ML application such as Pandas and Matplotlib

Aim:

To write a python program to Study various Python Libraries for ML application such as Pandas and Matplotlib

Description:

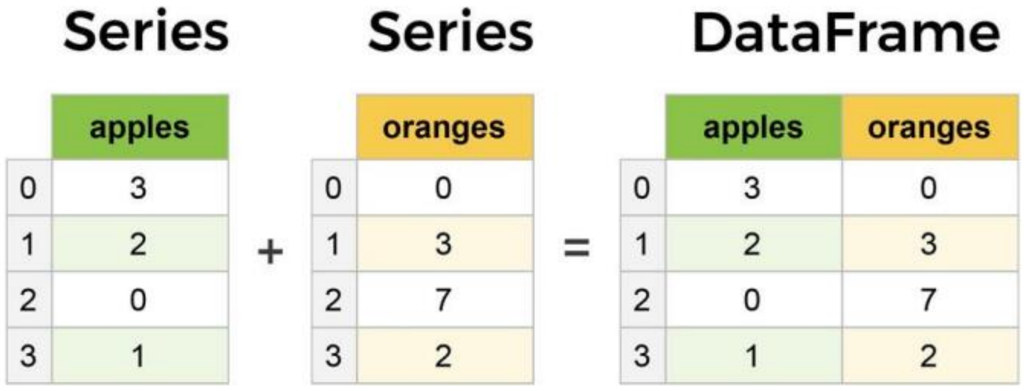
Pandas is a powerful open-source library used for data manipulation and analysis in Python. It is particularly useful for working with structured data (tabular data) in the form of data frames (2D tables). It provides a wide range of functionalities such as data cleaning, transformation, aggregation, and statistical analysis.

Matplotlib is a very popular Python library for data visualization. It is a 2D plotting library used for creating 2D graphs and plots. It provides various kinds of graphs and plots for data visualization, viz., line graphs, bar charts, histograms, scatter plots, histogram, etc,

Pandas library:

The primary components of **Pandas** include two main data structures: **Series** and **DataFrame**. These components form the backbone of data manipulation and analysis in Pandas.

1. **Series:** A Pandas **Series** is a one-dimensional labeled array capable of holding any data type (integers, strings, floats, etc.). It is similar to a column in a spreadsheet or a database table.
2. **DataFrame:** Data Frame is a multi-dimensional table made up of a collection of Series.



Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

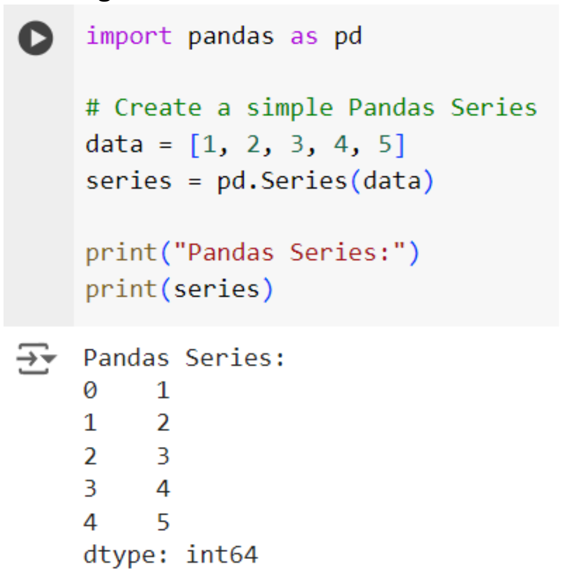
=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Programs:

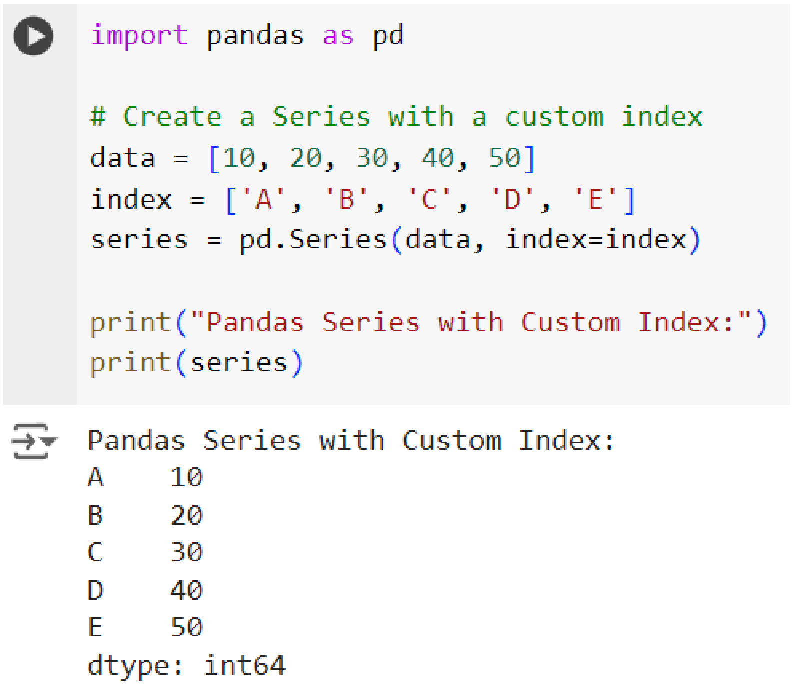
1. Creating a series



```python
import pandas as pd

# Create a simple Pandas Series
data = [1, 2, 3, 4, 5]
series = pd.Series(data)

print("Pandas Series:")
print(series)
```

```
Pandas Series:
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

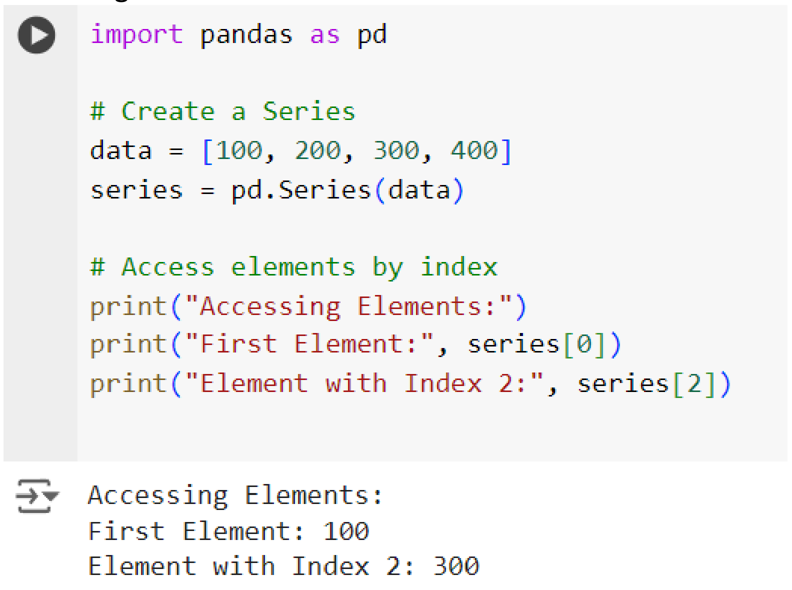
2. Creating series with custom index



```python
import pandas as pd

# Create a Series with a custom index
data = [10, 20, 30, 40, 50]
index = ['A', 'B', 'C', 'D', 'E']
series = pd.Series(data, index=index)

print("Pandas Series with Custom Index:")
print(series)
```

```
Pandas Series with Custom Index:
A    10
B    20
C    30
D    40
E    50
dtype: int64
```

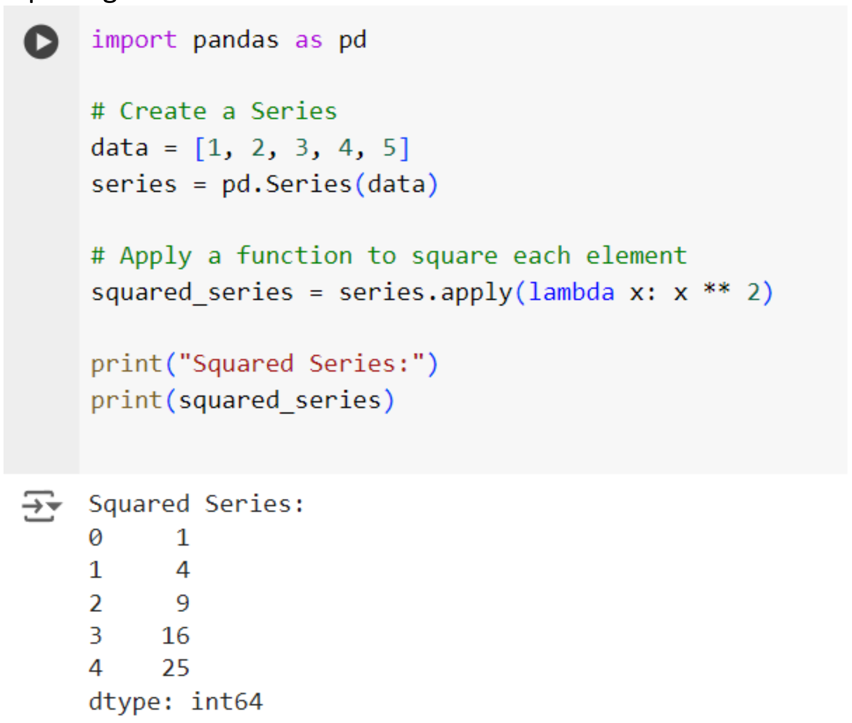
3. Accessing elements in a series



```python
import pandas as pd

# Create a Series
data = [100, 200, 300, 400]
series = pd.Series(data)

# Access elements by index
print("Accessing Elements:")
print("First Element:", series[0])
print("Element with Index 2:", series[2])
```

```
Accessing Elements:
First Element: 100
Element with Index 2: 300
```

4. Squaring each element in a series

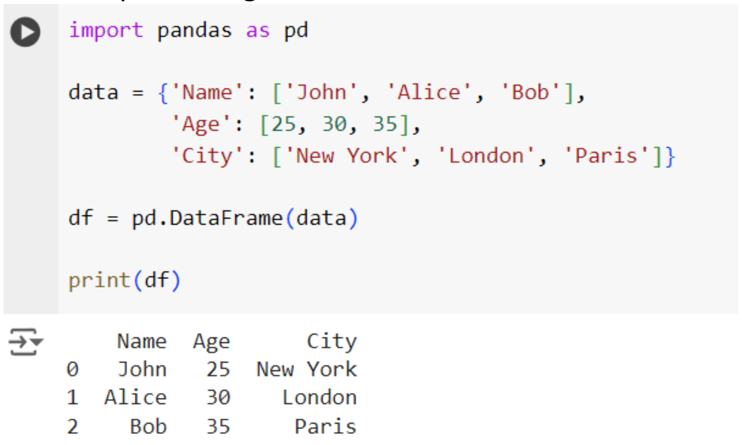


```python
import pandas as pd

# Create a Series
data = [1, 2, 3, 4, 5]
series = pd.Series(data)

# Apply a function to square each element
squared_series = series.apply(lambda x: x ** 2)

print("Squared Series:")
print(squared_series)
```

```
Squared Series:
0     1
1     4
2     9
3    16
4    25
dtype: int64
```

5. Creating a dataframe from a dictionary

   Description: The code creates a dictionary data with keys representing column names and values representing column data.

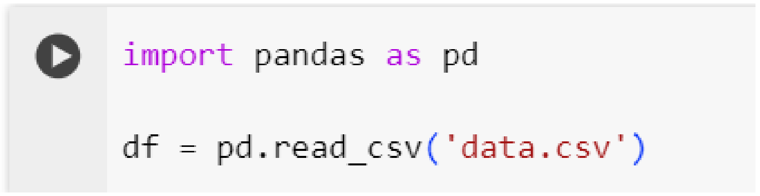


```
import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [25, 30, 35],
        'City': ['New York', 'London', 'Paris']}

df = pd.DataFrame(data)

print(df)
```

```
    Name  Age      City
0   John   25  New York
1  Alice   30    London
2    Bob   35     Paris
```

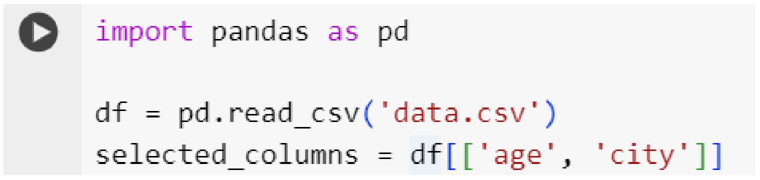
6. Read csv file into a data frame

   Description: The code uses the read_csv() function from pandas to read a CSV file named 'data.csv' and store the data in a DataFrame called df.



```
import pandas as pd

df = pd.read_csv('data.csv')
```

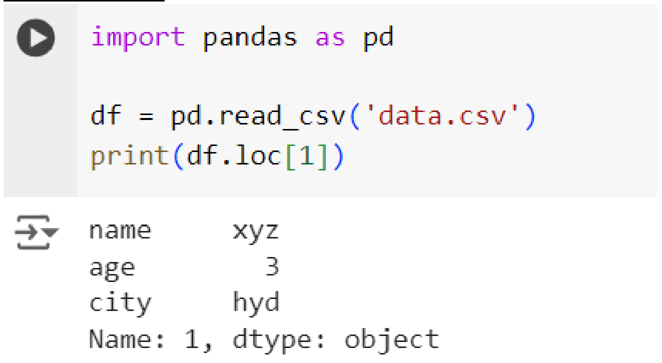
7. Selecting specific colums from a data frame



```
import pandas as pd

df = pd.read_csv('data.csv')
selected_columns = df[['age', 'city']]
```

8. Return a specific row from a data frame

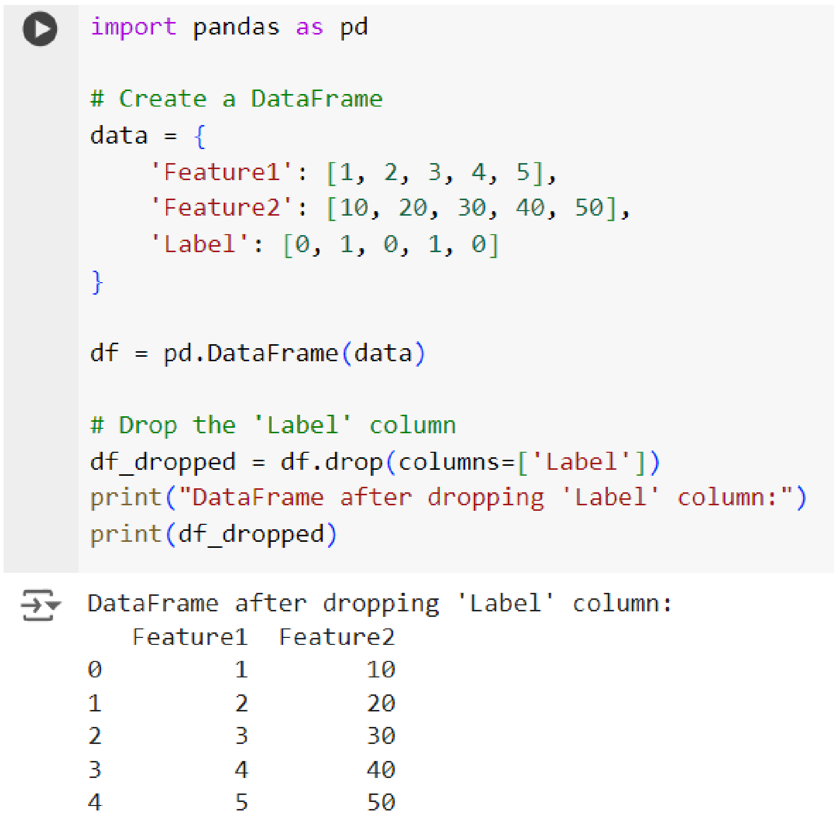
Description: Pandas use the loc attribute to return one or more specified row(s)



```python
import pandas as pd

df = pd.read_csv('data.csv')
print(df.loc[1])
```

```
name      xyz
age         3
city      hyd
Name: 1, dtype: object
```

9. Dropping a specific label from a data frame

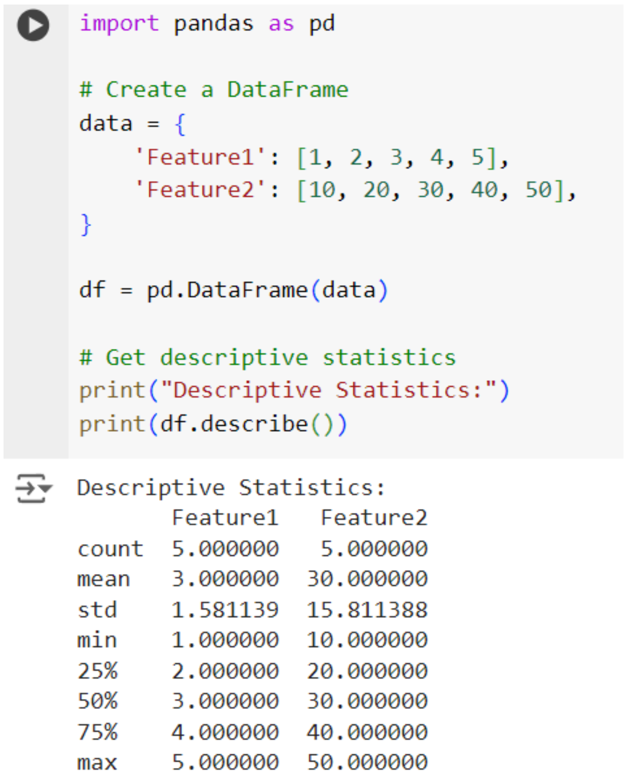


```python
import pandas as pd

# Create a DataFrame
data = {
    'Feature1': [1, 2, 3, 4, 5],
    'Feature2': [10, 20, 30, 40, 50],
    'Label': [0, 1, 0, 1, 0]
}

df = pd.DataFrame(data)

# Drop the 'Label' column
df_dropped = df.drop(columns=['Label'])
print("DataFrame after dropping 'Label' column:")
print(df_dropped)
```

```
DataFrame after dropping 'Label' column:
   Feature1  Feature2
0         1        10
1         2        20
2         3        30
3         4        40
4         5        50
```

10. Display descriptive statistics

Description: The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.



```python
import pandas as pd

# Create a DataFrame
data = {
    'Feature1': [1, 2, 3, 4, 5],
    'Feature2': [10, 20, 30, 40, 50],
}

df = pd.DataFrame(data)

# Get descriptive statistics
print("Descriptive Statistics:")
print(df.describe())
```

```
Descriptive Statistics:
       Feature1   Feature2
count  5.000000   5.000000
mean   3.000000  30.000000
std    1.581139  15.811388
min    1.000000  10.000000
25%    2.000000  20.000000
50%    3.000000  30.000000
75%    4.000000  40.000000
max    5.000000  50.000000
```

Matplotlib: pyplot is a module within the Matplotlib library in Python, widely used for creating static, animated, and interactive visualizations in data analysis. It provides a collection of functions that make it easy to generate a variety of plots and charts, enabling users to visualize data effectively.

It supports numerous types of plots, including:

- Line plots
- Scatter plots
- Bar charts
- Histograms
- Pie charts
- Box plots
- Heatmaps
- 3D plots

1. Line Plot:

It is commonly used to represent quantitative data over a continuous interval or time, making it particularly useful for visualizing trends, patterns, and changes in data over time.

2. Scatter Plots

Scatter plots display individual data points as dots on a two-dimensional graph, showing the relationship between two numerical variables. They are useful for identifying correlations, distributions, and outliers in datasets.

3. Bar Charts

Bar charts represent categorical data with rectangular bars, where the length of each bar is proportional to the value it represents. They allow for easy comparison between different categories or groups.

4. Histograms

Histograms visualize the distribution of numerical data by grouping data points into bins or intervals. The height of each bar represents the frequency of data points within that range, making it useful for understanding data distribution.

5. Pie Charts

Pie charts represent the composition of a whole by dividing a circle into slices, each representing a category's proportion. They are best used for displaying relative sizes of parts to a whole but can be less effective for precise comparisons.

6. Box Plots

Box plots summarize the distribution of a dataset through its quartiles, displaying the median, upper, and lower quartiles, as well as potential outliers. They provide a clear view of the data's central tendency and variability.

7. Heatmaps

Heatmaps use color to represent values in a matrix, showing the intensity of data points in a two-dimensional space. They are effective for visualizing correlations, frequencies, or patterns in complex datasets.

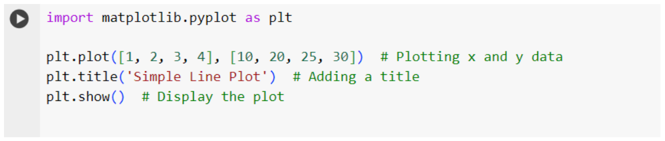
8. 3D Plots

3D plots visualize data in three-dimensional space, allowing for the representation of relationships among three variables. They are useful for exploring complex datasets but can become cluttered and harder to interpret than 2D plots.

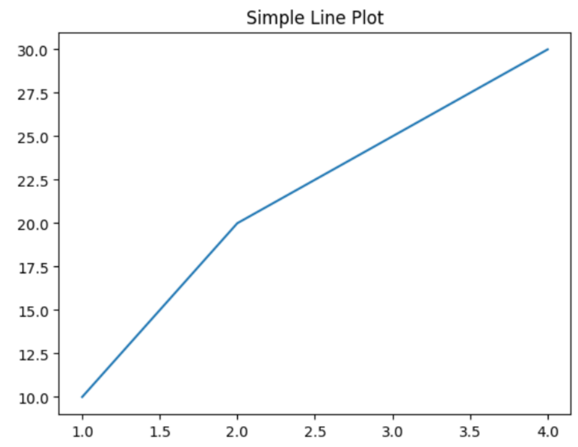
Programs:

1. Program to illustrate simple line plot

   We pass two arrays as input arguments to Pyplot's plot() method and use show()method to invoke the required plot. Here note that the first array appears on the x-axis and second array appears on the y-axis of the plot.

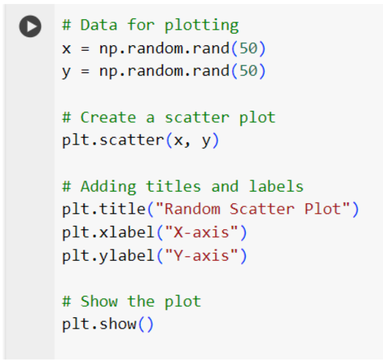


```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])  # Plotting x and y data
plt.title('Simple Line Plot')  # Adding a title
plt.show()  # Display the plot
```

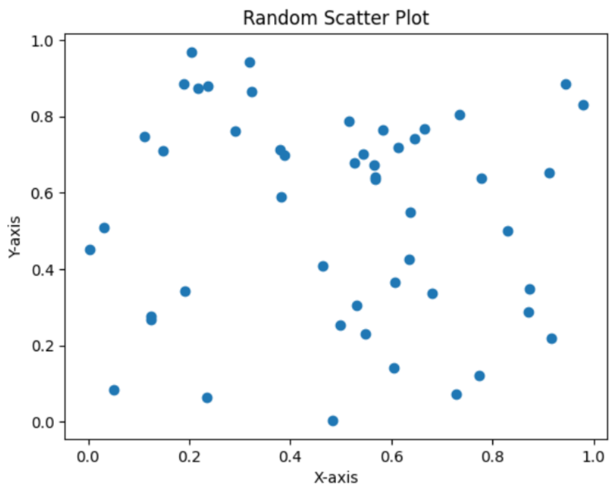


2. **Program to illustrate scatter plot**

The code generates 50 random data points, creates a scatter plot to visualize these points, adds a title and axis labels, and finally displays the plot.

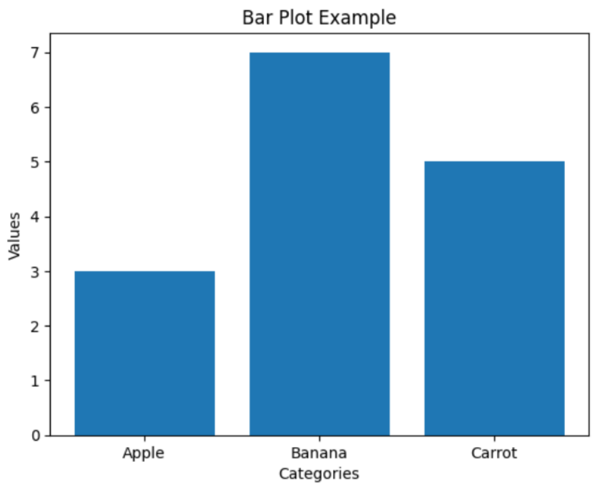


```
# Data for plotting
x = np.random.rand(50)
y = np.random.rand(50)

# Create a scatter plot
plt.scatter(x, y)

# Adding titles and labels
plt.title("Random Scatter Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")

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



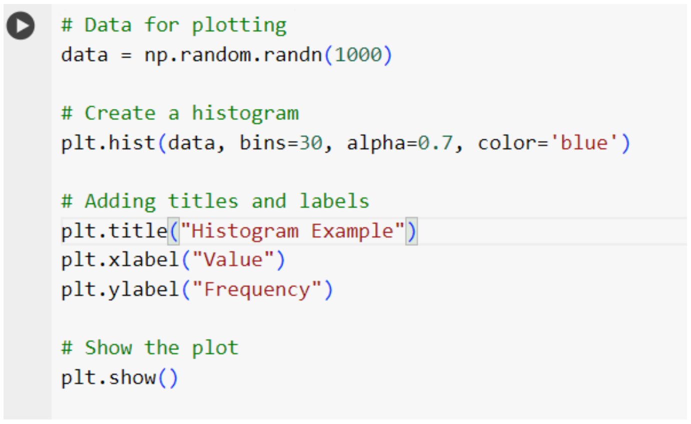
3. Program to illustrate bar chart

```
# Data for plotting
categories = ['Apple', 'Banana', 'Carrot']
quantity = [3, 7, 5]

# Create a bar plot
plt.bar(categories, quantity)

# Adding titles and labels
plt.title("Bar Plot Example")
plt.xlabel("Categories")
plt.ylabel("Values")

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

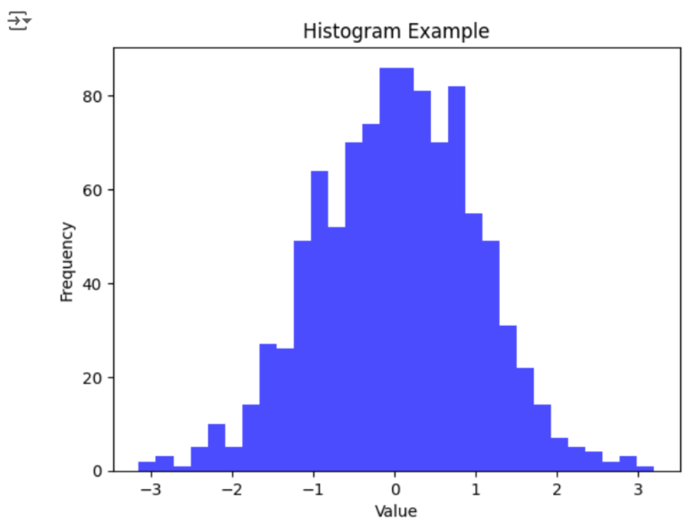


4. Program to illustrate histogram

The code generates 1000 random data points from a normal distribution, creates a histogram to visualize the distribution of these points across 30 bins, adds appropriate titles and labels, and finally displays the histogram. Histograms are useful for understanding the distribution, central tendency, and spread of numerical data.



```
# Data for plotting
data = np.random.randn(1000)

# Create a histogram
plt.hist(data, bins=30, alpha=0.7, color='blue')

# Adding titles and labels
plt.title("Histogram Example")
plt.xlabel("Value")
plt.ylabel("Frequency")

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



week4:

Aim:

To write a Python program to implement Simple Linear Regression

Description:

Simple Linear Regression is a statistical method used to model the relationship between two variables: one independent variable (predictor variable) and one dependent variable (response variable). It assumes that there is a linear relationship between the independent variable X and the dependent variable Y.

The mathematical representation of simple linear regression can be expressed as:

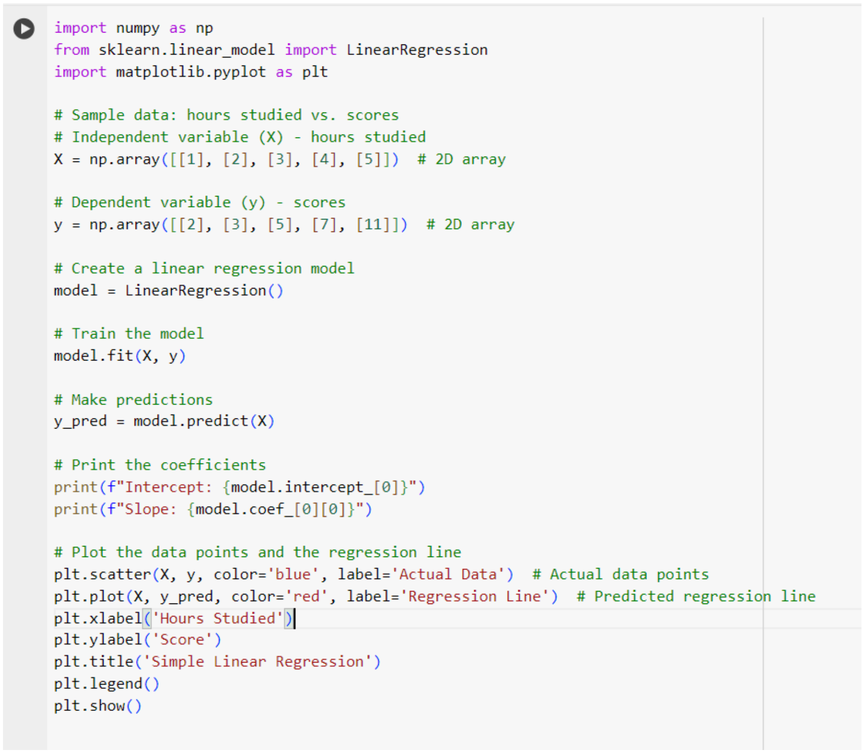
$$Y = \beta_0 + \beta_1 X + \epsilon$$

where:

- Y is the dependent variable (response variable),
- X is the independent variable (predictor variable),
- β_0 is the intercept (the value of Y when $X = 0$),
- β_1 is the slope (the change in Y for a unit change in X),
- ϵ is the error term (the difference between the observed and predicted values).

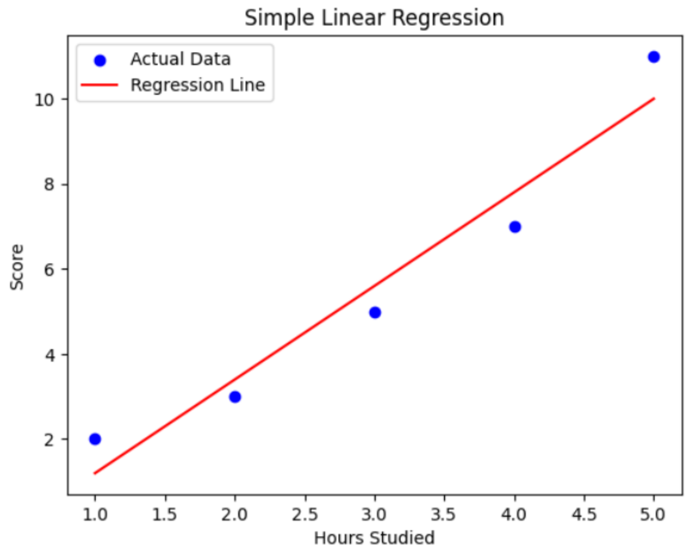
The goal of simple linear regression is to estimate the coefficients β_0 and β_1 that minimize the sum of squared errors between the observed and predicted values. This is often done using the method of least squares.

Program:



```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Sample data: hours studied vs. scores
# Independent variable (X) - hours studied
X = np.array([[1], [2], [3], [4], [5]])  # 2D array

# Dependent variable (y) - scores
y = np.array([[2], [3], [5], [7], [11]])  # 2D array

# Create a linear regression model
model = LinearRegression()

# Train the model
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)

# Print the coefficients
print(f"Intercept: {model.intercept_[0]}")
print(f"Slope: {model.coef_[0][0]}")

# Plot the data points and the regression line
plt.scatter(X, y, color='blue', label='Actual Data')  # Actual data points
plt.plot(X, y_pred, color='red', label='Regression Line')  # Predicted regression line
plt.xlabel('Hours Studied')
plt.ylabel('Score')
plt.title('Simple Linear Regression')
plt.legend()
plt.show()
```

```
Intercept: -1.0000000000000018
Slope: 2.2000000000000006
```


Simple Linear Regression
● Actual Data
Regression Line
Score
Hours Studied