What is NumPy, and why is it important in the Python ecosystem?

=> Numpy (Numerical Python) is a powerful library for scientific computing in Python, providing efficient storage and manipulation of large datasets.

Important of Numpy are as below:

1. Efficient numerical operations:NumPy provides high-performance multidimensional array objects and functions for efficient numerical computations.
2. Data manipulation: It offers powerful tools for manipulating arrays, such as slicing, indexing, reshaping, and broadcasting, which are essential for data preprocessing and analysis.
3. Integration with other libraries: NumPy seamlessly integrates with many other libraries and frameworks in the scientific computing ecosystem, such as SciPy, pandas, and scikit-learn.
4. Performance optimization: NumPy's array-oriented computing paradigm and implementation in optimized C code lead to faster execution compared to traditional Python lists.
5. Foundation for scientific computing: It serves as a foundational library for scientific computing in Python, providing the building blocks for implementing algorithms in areas like mathematics, statistics, physics, and engineering.

Discuss the history and development of NumPy. How has it evolved over time?

=> The origins of NumPy trace back to the Numeric library, primarily developed by Jim Hugunin. In 2005, Travis Oliphant combined features from the alternative Numarray into Numeric, creating what we now know as NumPy. Since then, NumPy has remained an open-source project governed by the NumFOCUS organization and licensed under the modified BSD license.

Initially launched on November 11, 2005, NumPy gradually replaced Numeric, becoming the cornerstone of scientific Python ecosystems. An effort to adapt NumPy for Python 3 began in 2011, culminating in the release of NumPy version 1.5.0. Although there have been attempts to implement NumPy on platforms like PyPy, complete compatibility remains elusive as of 2023.

Describe the core features of NumPy. How do these features benefit scientific and mathematical computing efforts?

=> NumPy is a powerful Python library designed for numerical and scientific computing. Its core features include:

1. **Ndarrays:** Multi-dimensional, homogeneous, and typified data arrays resembling those in MATLAB. These arrays enable efficient computations since they store elements contiguously in memory, allowing vectorized operations and reducing overhead associated with traditional Python lists.
2. **Integration with Python:** NumPy combines both C and Python code, making it highly efficient while preserving the simplicity and flexibility of Python. This hybrid approach ensures seamless interaction with native Python components and enables easy integration with other libraries.
3. **Mathematical Functions:** A wide range of mathematical functions tailored explicitly for ndarrays, facilitating intricate calculations required in data science, machine learning, physics, and mathematics.
4. **Compatibility and Extendability:** NumPy serves as the bedrock for numerous Python packages related to data science, such as SciPy, pandas, and scikit-learn. Moreover, developers can create custom functionality through extensions written in lower-level languages like Cython, Fortran, and C.

These features collectively empower scientific and mathematical computing efforts by:

- Accelerating computations, especially array-based ones, leading to significant improvements in performance compared to standard Python data types.
- Simplifying coding processes thanks to the consistency provided by uniform data containers (i.e., ndarrays) and extensive built-in functionalities.
- Encouraging modularity, enabling users to combine multiple libraries within a coherent framework, thereby streamlining workflows and enhancing productivity.
- Fostering collaboration among researchers due to its widespread adoption across academia and industries, ensuring knowledge transfer and shared best practices.

Explain the concept of ndarrays. How do ndarrays differ from standard Python lists?

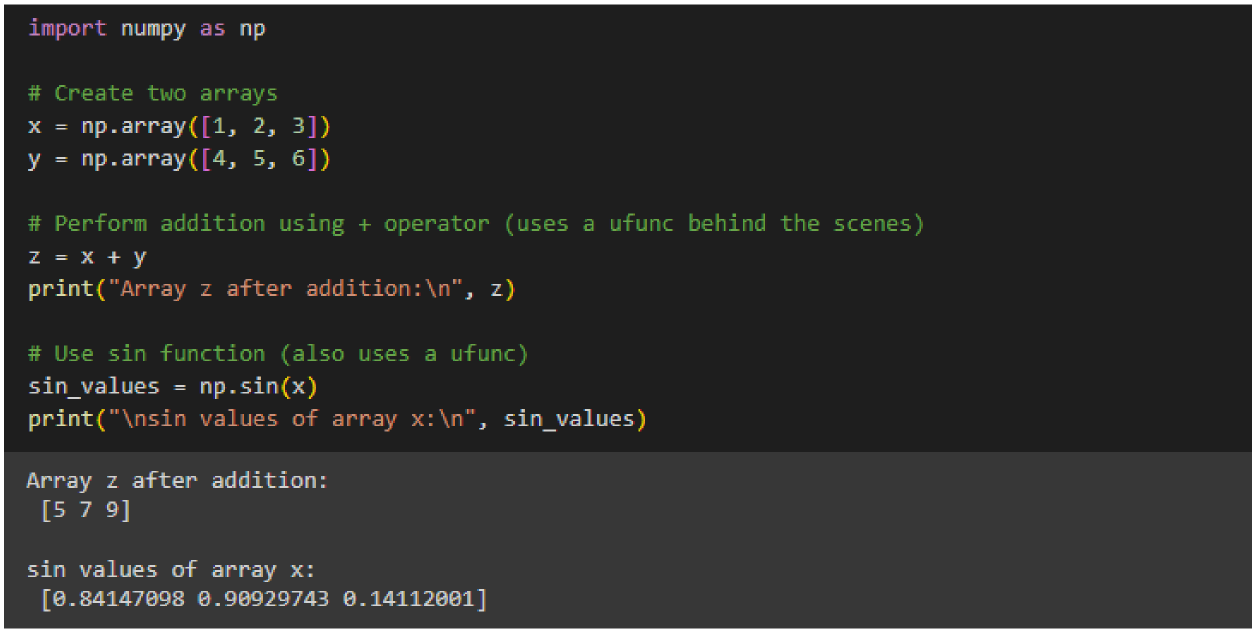
=> An ndarray (short for "N-dimensional array") in NumPy is a specialized, multi-dimensional, homogeneous data structure optimized for numerical computations. Each item in the array shares the same data type—integer, floating-point numbers, etc. They provide several benefits over standard Python lists:

1. **Efficient storage:** Contiguous blocks of memory store ndarray elements, resulting in faster access times than standard Python lists. This design allows for more efficient use of cache during computation, ultimately improving overall performance.
2. **Vectorized operations:** With ndarray, you can perform vectorized arithmetic operations directly between arrays instead of iterating over individual list elements. Vectorization simplifies your code and improves execution speed because it leverages compiled NumPy functions rather than slower interpreted Python loops.
3. **Fixed size:** Once created, the shape (dimensions) and data type of an ndarray cannot change, unlike Python lists. While this might appear restrictive at first glance, having fixed sizes makes certain operations much faster and less error-prone. Additionally, it helps avoid unexpected memory usage changes throughout your application.
4. **Broadcasting:** When performing arithmetic operations involving two arrays of different shapes, NumPy automatically broadcasts them according to specific rules, eliminating the need for manual reshaping or looping through smaller arrays. Broadcasting provides great convenience while writing concise, readable code.
5. **Built-in functions:** Compared to standard Python lists, ndarray offers many additional built-in functions specifically designed for advanced numerical processing, such as linear algebra operations, statistical analysis, discrete Fourier Transforms, random number generation, and others.

What are universal functions (ufuncs) in NumPy? Give examples and explain their significance.

=> Universal functions (ufuncs) in NumPy are functions operating on ndarrays in an element-wise manner, supporting array broadcasting, typecasting, and several other standard features. Ufuncs serve as 'vectorized' wrappers around functions taking a fixed number of scalar inputs and producing a fixed number of scalar outputs. Examples of ufuncs include math operations (addition, subtraction), trigonometric functions (sin, cos), bit-twiddling functions (bitwise AND, OR), comparison functions (equal, greater), and floating functions (log, exp).

Here's a simple example illustrating the power of ufuncs in NumPy:



```
import numpy as np

# Create two arrays
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

# Perform addition using + operator (uses a ufunc behind the scenes)
z = x + y
print("Array z after addition:\n", z)

# Use sin function (also uses a ufunc)
sin_values = np.sin(x)
print("\nsin values of array x:\n", sin_values)
```

```
Array z after addition:
 [5 7 9]

sin values of array x:
 [0.84147098 0.90929743 0.14112001]
```

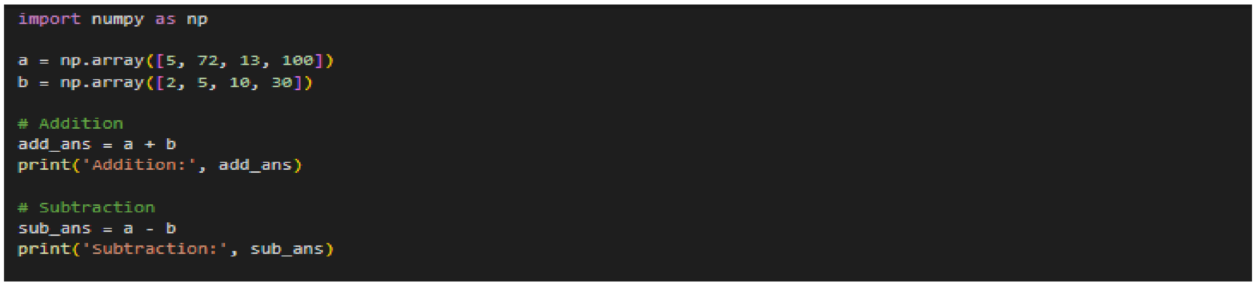
Significance of ufuncs lies in their capability to process arrays element-wise efficiently, promoting better code maintainability, enhanced performance, and easier integration with external libraries. The vast collection of predefined ufuncs spanning diverse domains further strengthens their utility in scientific and mathematical computing endeavors.

Describe various mathematical operations that can be performed using NumPy. Provide examples for each.

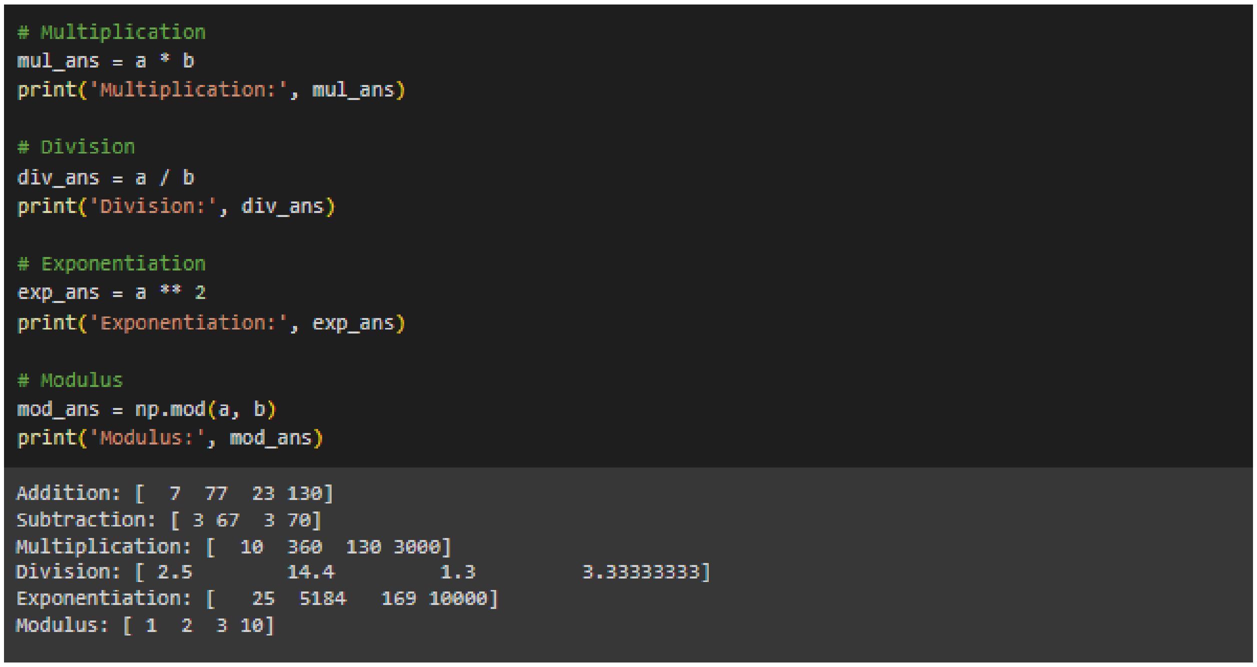
=> NumPy provides support for various mathematical operations, including basic arithmetic operations, trigonometry, logarithmic, exponential functions, and more. Below are some common mathematical operations you can perform using NumPy.

Arithmetic Operations:

NumPy supports basic arithmetic operations like addition, subtraction, multiplication, division, exponentiation, and modulus. To demonstrate this, consider two arrays a and b.



```
import numpy as np

a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])

# Addition
add_ans = a + b
print('Addition:', add_ans)

# Subtraction
sub_ans = a - b
print('Subtraction:', sub_ans)
```

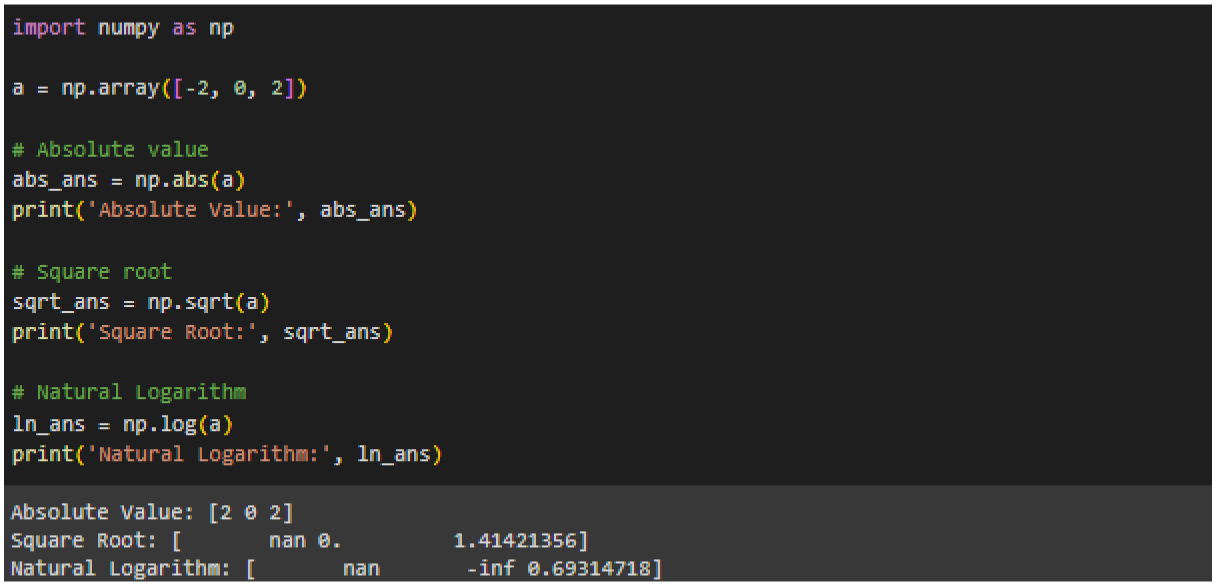


```
# Multiplication
mul_ans = a * b
print('Multiplication:', mul_ans)

# Division
div_ans = a / b
print('Division:', div_ans)

# Exponentiation
exp_ans = a ** 2
print('Exponentiation:', exp_ans)

# Modulus
mod_ans = np.mod(a, b)
print('Modulus:', mod_ans)
```

```
Addition: [  7  77  23 130]
Subtraction: [ 3 67  3 70]
Multiplication: [  10  360  130 3000]
Division: [ 2.5        14.4         1.3         3.33333333]
Exponentiation: [   25  5184   169 10000]
Modulus: [ 1  2  3 10]
```

Universal Functions (UFuncs):

NumPy includes a variety of UFuncs for handling mathematical operations. Some examples include np.abs(), np.sqrt(), np.square(), np.sin(), np.cos(), np.tan(), np.arcsin(), np.arccos(), np.arctan(), np.log(), np.exp(), and so on.

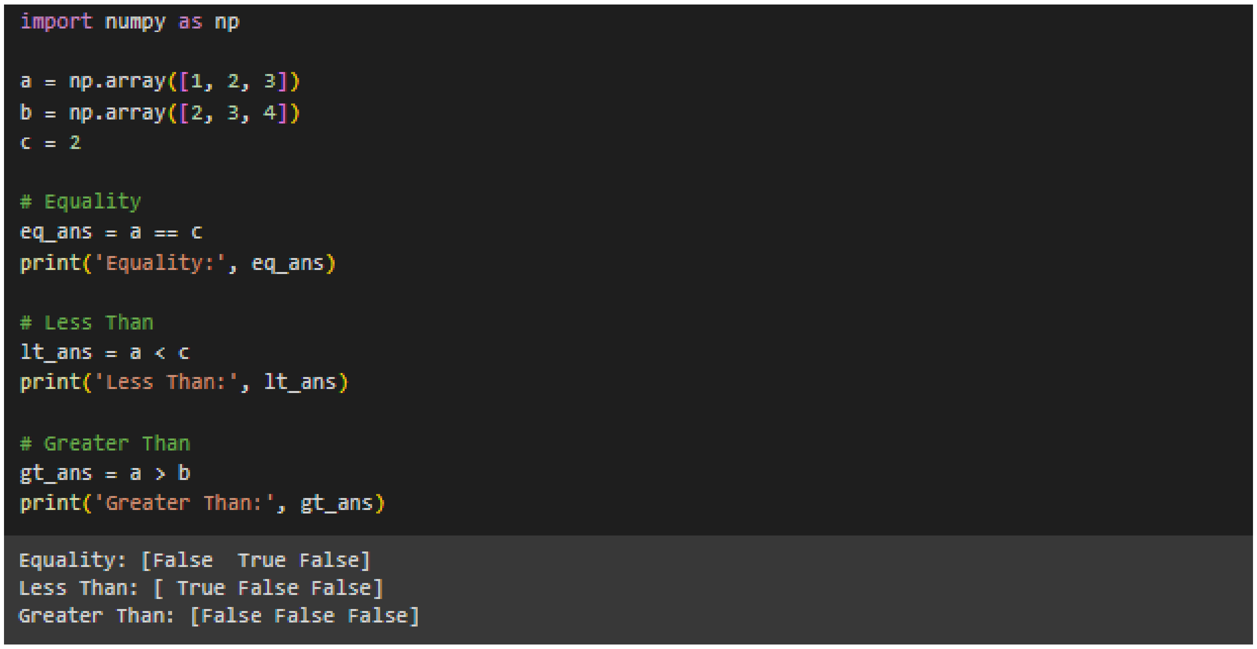


```
import numpy as np

a = np.array([-2, 0, 2])

# Absolute value
abs_ans = np.abs(a)
print('Absolute Value:', abs_ans)

# Square root
sqrt_ans = np.sqrt(a)
print('Square Root:', sqrt_ans)

# Natural Logarithm
ln_ans = np.log(a)
print('Natural Logarithm:', ln_ans)
```

```
Absolute Value: [2 0 2]
Square Root: [       nan 0.         1.41421356]
Natural Logarithm: [       nan       -inf 0.69314718]
```

Comparison Operations:

NumPy supports comparing arrays against scalars or another array. Comparison operations return Boolean arrays indicating whether the relationship holds true element-wise.

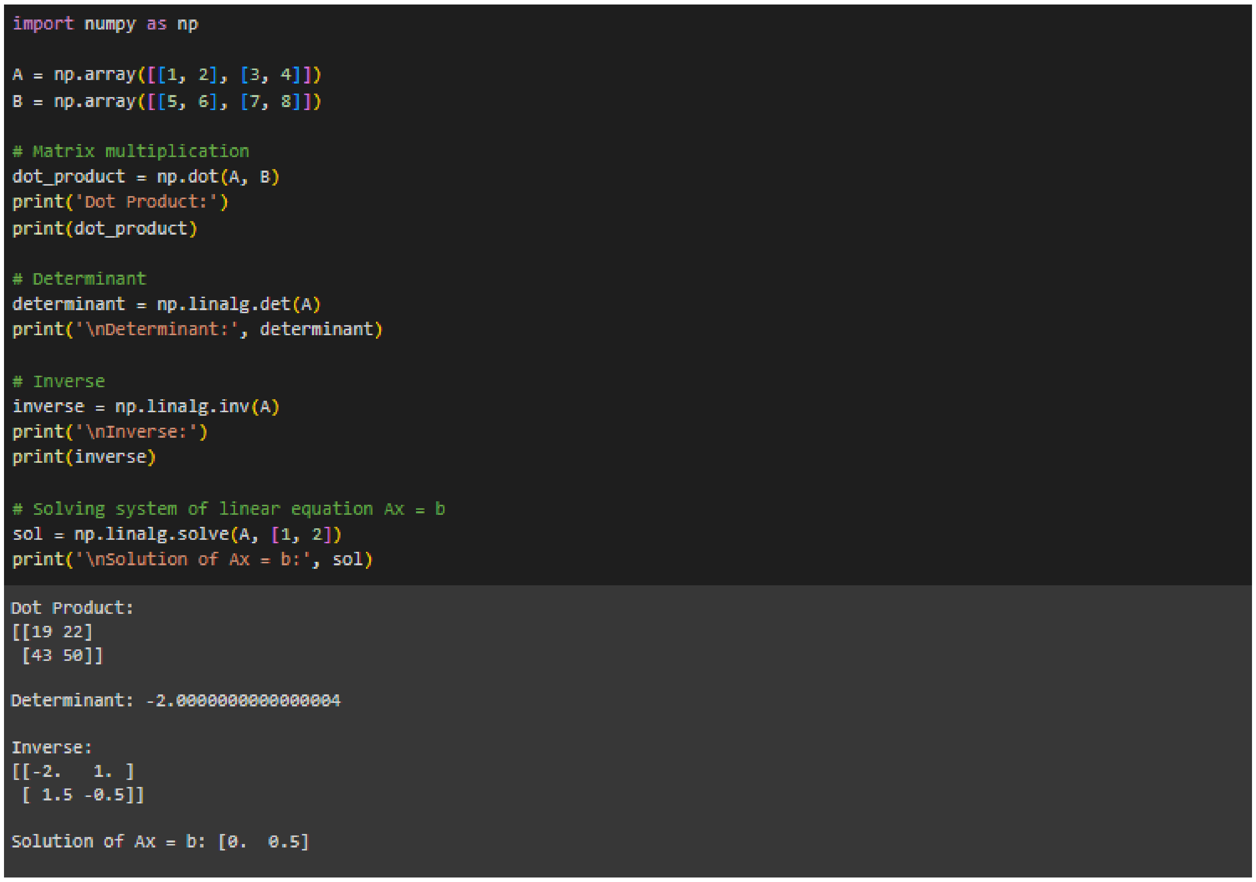


```python
import numpy as np

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

# Equality
eq_ans = a == c
print('Equality:', eq_ans)

# Less Than
lt_ans = a < c
print('Less Than:', lt_ans)

# Greater Than
gt_ans = a > b
print('Greater Than:', gt_ans)
```

```
Equality: [False  True False]
Less Than: [ True False False]
Greater Than: [False False False]
```

Matrix Operations:

You can find transposes, determinants, inverses, dot products, and solve systems of linear equations using NumPy's Linear Algebra module.

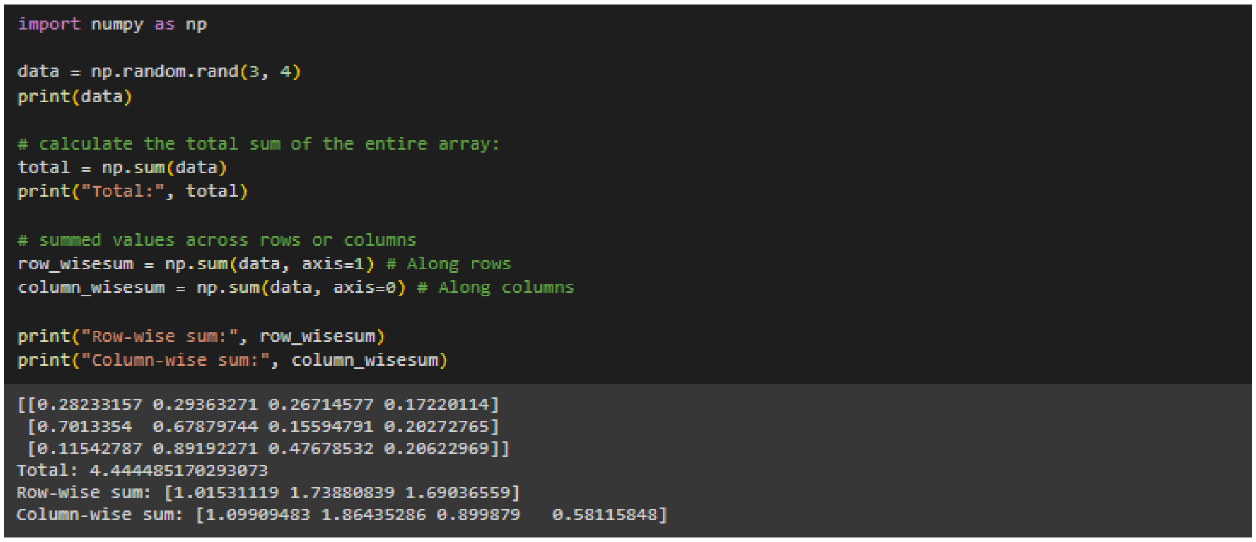


```python
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication
dot_product = np.dot(A, B)
print('Dot Product:')
print(dot_product)

# Determinant
determinant = np.linalg.det(A)
print('\nDeterminant:', determinant)

# Inverse
inverse = np.linalg.inv(A)
print('\nInverse:')
print(inverse)

# Solving system of linear equation Ax = b
sol = np.linalg.solve(A, [1, 2])
print('\nSolution of Ax = b:', sol)
```

```
Dot Product:
[[19 22]
 [43 50]]

Determinant: -2.0000000000000004

Inverse:
[[-2.   1. ]
 [ 1.5 -0.5]]

Solution of Ax = b: [0.  0.5]
```

Explain the concept of aggregation in NumPy. How does it differ from simple summation or multiplication?

=>Aggregation functions in NumPy typically operate on either the whole array or along specified axes, where axes represent dimensions of the array. Manipulation along axes reduces dimensionality and effectively shrinks the original array, thus generating valuable information regarding trends and patterns hidden beneath raw data points.



```
import numpy as np

data = np.random.rand(3, 4)
print(data)

# calculate the total sum of the entire array:
total = np.sum(data)
print("Total:", total)

# summed values across rows or columns
row_wisesum = np.sum(data, axis=1) # Along rows
column_wisesum = np.sum(data, axis=0) # Along columns

print("Row-wise sum:", row_wisesum)
print("Column-wise sum:", column_wisesum)
```

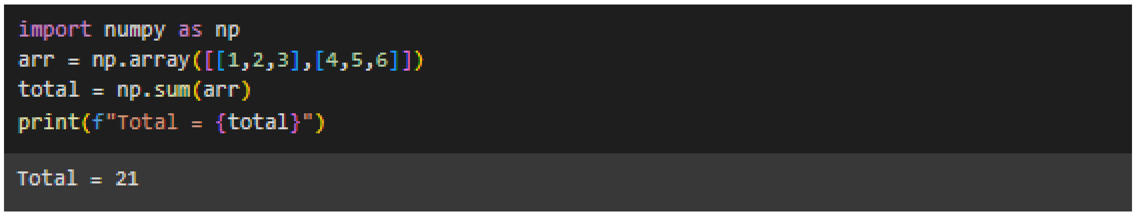
```
[[0.28233157 0.29363271 0.26714577 0.17220114]
 [0.7013354  0.67879744 0.15594791 0.20272765]
 [0.11542787 0.89192271 0.47678532 0.20622969]]
Total: 4.444485170293073
Row-wise sum: [1.01531119 1.73880839 1.69036559]
Column-wise sum: [1.09909483 1.86435286 0.899879   0.58115848]
```

Provide examples of different aggregation functions available in NumPy and their practical applications.

=>There are several aggregation functions available in NumPy, each with its unique purpose and application. Let me share some examples of these functions and describe their potential use cases:

1. **numpy.sum(array, axis=None):** Compute the sum of array elements. You may specify an optional axis argument to compute the sum along particular dimensions. Practical applications include computing total counts or quantities, evaluating marginal totals in contingency tables, and verifying checksums.

   Example:

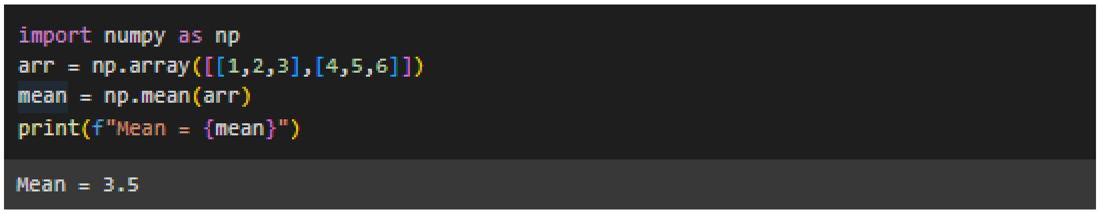


```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
total = np.sum(arr)
print(f"Total = {total}")
```

```
Total = 21
```

2. **numpy.mean(array, axis=None):** Calculate the arithmetic mean of array elements. Similar to numpy.sum, specifying the axis parameter computes the average along given dimensions. Mean calculation plays a vital role in descriptive statistics, helping identify central tendency indicators and normalizing distributions.

   Example:

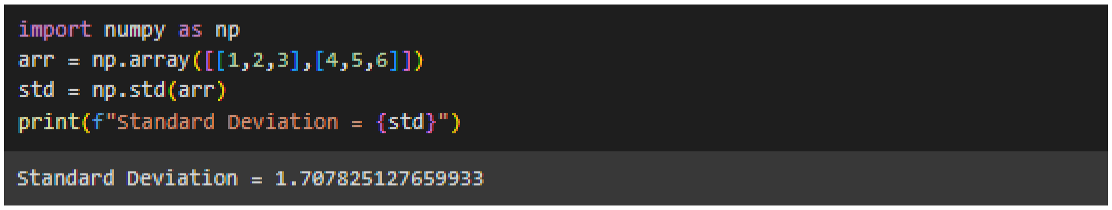


```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
mean = np.mean(arr)
print(f"Mean = {mean}")
```

```
Mean = 3.5
```

3. **numpy.std(array, axis=None):** Compute the population standard deviation. Standard deviation measures dispersion or spread in a dataset; hence, it finds relevance in hypothesis testing and probability density estimation.

   Example:

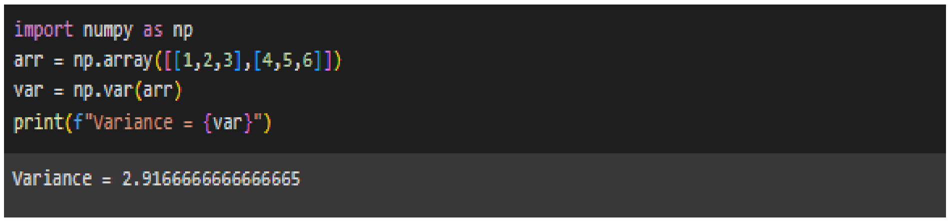


```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
std = np.std(arr)
print(f"Standard Deviation = {std}")
```

```
Standard Deviation = 1.707825127659933
```

4. **numpy.var(array, axis=None):** Calculate the population variance. Variance quantifies the degree of variation in a dataset, assisting analysts in measuring volatility and risk assessment.

   Example:

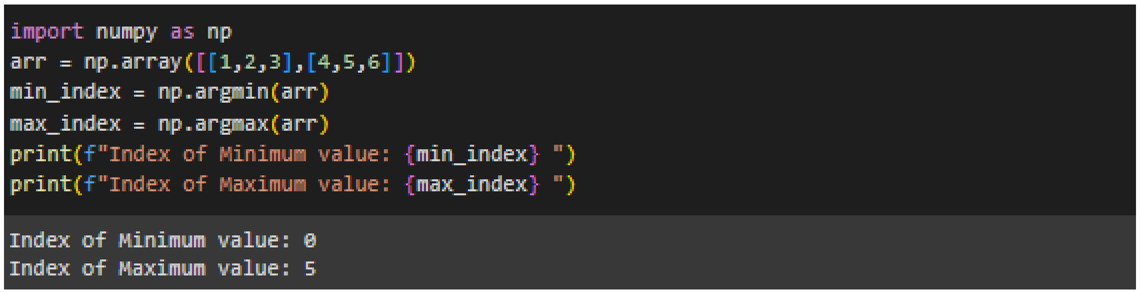


```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
var = np.var(arr)
print(f"Variance = {var}")
```

```
Variance = 2.9166666666666665
```

5. **numpy.argmin(array, axis=None) & numpy.argmax(array, axis=None):** Retrieve indices corresponding to the minimum and maximum values, respectively. Location identification helps pinpoint critical regions, select optimal parameters, and determine influential factors.

Example:



```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
min_index = np.argmin(arr)
max_index = np.argmax(arr)
print(f"Index of Minimum value: {min_index} ")
print(f"Index of Maximum value: {max_index} ")
```

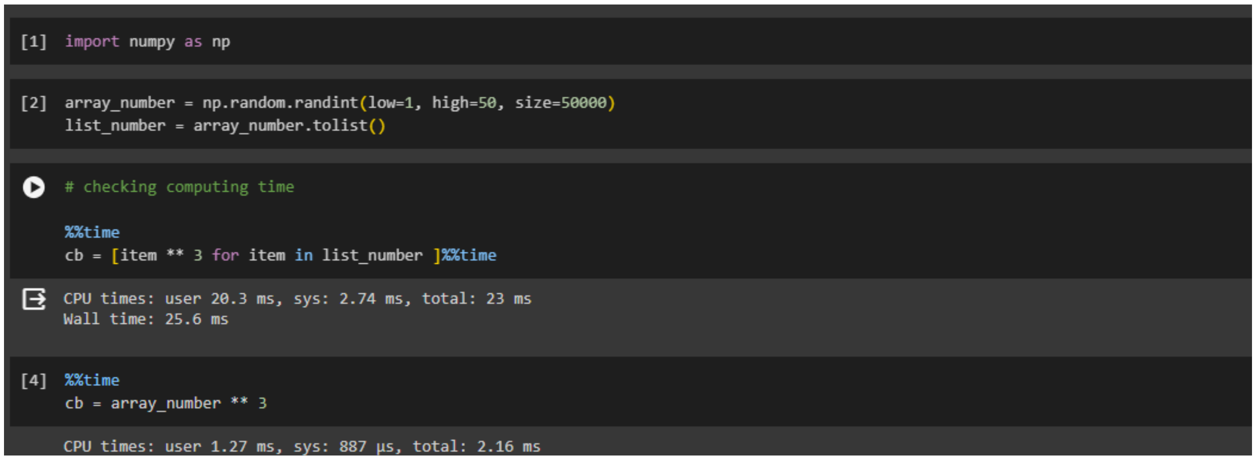
```
Index of Minimum value: 0
Index of Maximum value: 5
```

Difference between numpy array and python list

=> The following table shows the difference between numpy array and python list:

Aspect	NumPy Arrays	Python Lists
Data Structure	Homogeneous (same data type for all elements)	Heterogeneous (can contain elements of different types)
Memory Usage	More memory-efficient due to contiguous storage	Less memory-efficient due to overhead and fragmentation
Speed/Performance	Faster execution for numerical operations	Slower for numerical computations, especially with large datasets
Functionality	Optimized for numerical computations with extensive mathematical functions	More general-purpose with built-in methods for data manipulation
Ease of use	May have a steeper learning curve but powerful for numerical tasks	Simple syntax and flexibility for beginners and general tasks

Time complexity comparison of NumPy arrays and Python list:


```
[1] import numpy as np

[2] array_number = np.random.randint(low=1, high=50, size=50000)
    list_number = array_number.tolist()

    # checking computing time

    %%time
    cb = [item ** 3 for item in list_number ]%%time

    CPU times: user 20.3 ms, sys: 2.74 ms, total: 23 ms
    Wall time: 25.6 ms

[4] %%time
    cb = array_number ** 3

    CPU times: user 1.27 ms, sys: 887 µs, total: 2.16 ms
```

Here, array_number take 1.59ms to calculate cube where list_number takes 25.6 ms to calculate cube. Hence, numpy array operation is faster than python list

Explain the following with appropriate examples:

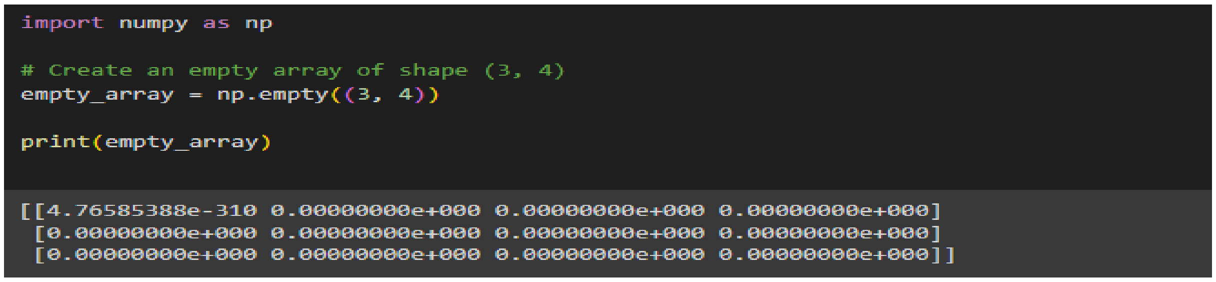
1. **np.empty:**

   "np.empty" is a function provided by the NumPy library in Python. It creates an uninitialized array of specified shape and data type, but its elements are not initialized.

   Syntax:

   numpy.empty(shape, dtype=float, order='C', *, like=None)

   Return a new array of given shape and type, without initializing entries.


```
import numpy as np

# Create an empty array of shape (3, 4)
empty_array = np.empty((3, 4))

print(empty_array)

[[4.76585388e-310 0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000 0.00000000e+000]]
```

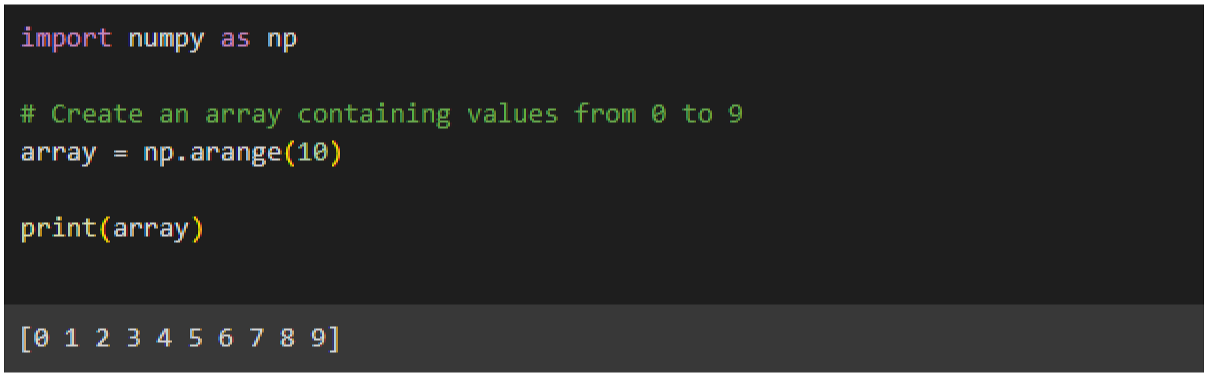
2. **np.arange:**

   "np.arange" is a function provided by the NumPy library in Python. It creates an array containing evenly spaced values within a specified range.

   Syntax:

   numpy.arange([start,]stop, [step,]dtype=None, *, like=None)

   Return evenly spaced values within a given interval.



```
import numpy as np

# Create an array containing values from 0 to 9
array = np.arange(10)

print(array)
```

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

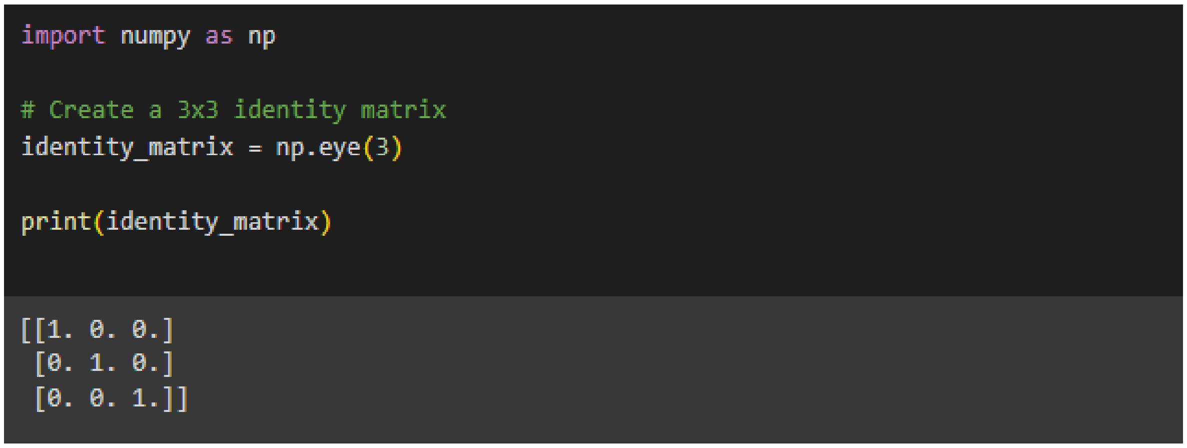
3. **np.eye:**

   “np.eye” is a function provided by the NumPy library in Python. It is used to create a 2-dimensional array (or matrix) with diagonal elements set to 1 and the remaining elements set to 0.

   Syntax:

   numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, device=None, like=None)

   Return a 2-D array with ones on the diagonal and zeros elsewhere.



```
import numpy as np

# Create a 3x3 identity matrix
identity_matrix = np.eye(3)

print(identity_matrix)
```

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

4. **np.linspace:**

   “np.linspace” is a function provided by the NumPy library in Python. It is used to create an array of evenly spaced numbers over a specified interval.

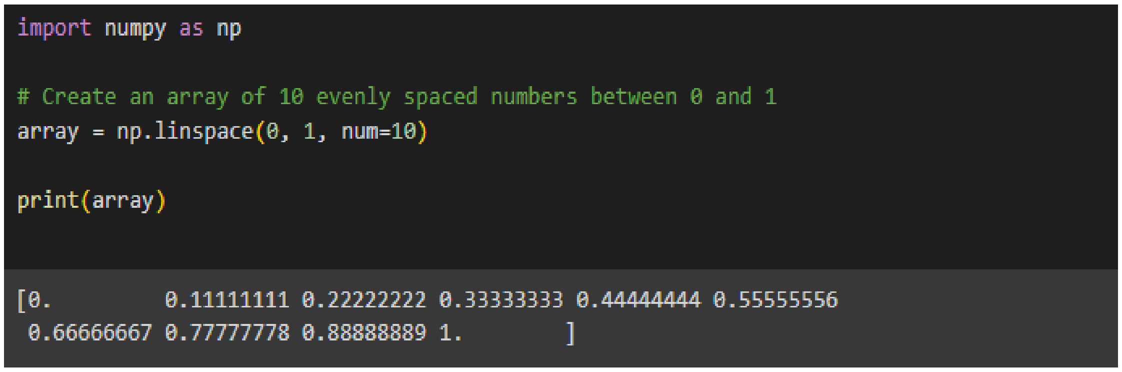
   Syntax:

   numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.



```python
import numpy as np

# Create an array of 10 evenly spaced numbers between 0 and 1
array = np.linspace(0, 1, num=10)

print(array)
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

5. **Shape vs reshape:**

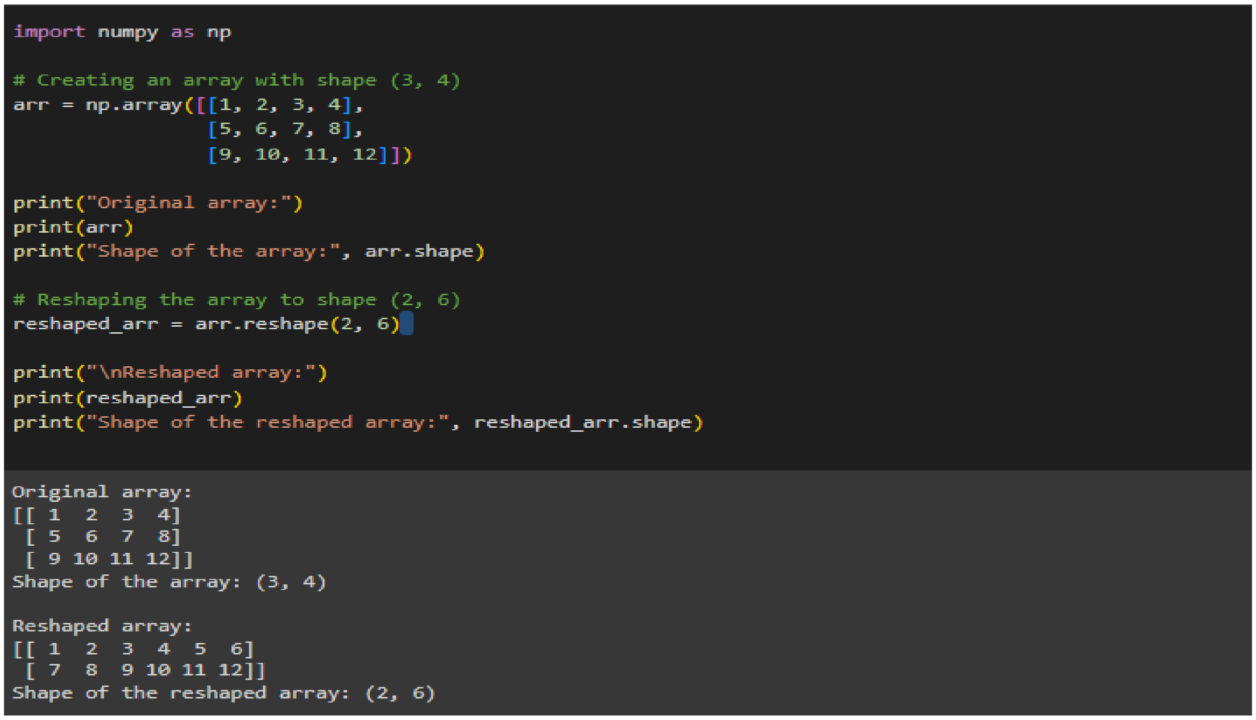
Shape:

- shape refers to the dimensions (or size) of an array. It is a tuple indicating the number of elements along each dimension of the array.
- For example, for a 2D array with 3 rows and 4 columns, the shape would be (3, 4).

Reshape:

- reshape is a method in NumPy used to change the shape of an array without changing its data. It returns a new array with a modified shape.
- The total number of elements in the original array must be the same as the total number of elements in the reshaped array, otherwise, a ValueError will be raised.
- Reshaping does not change the original array; it returns a new array with the specified shape.

Here's an example demonstrating the difference between shape and reshape:

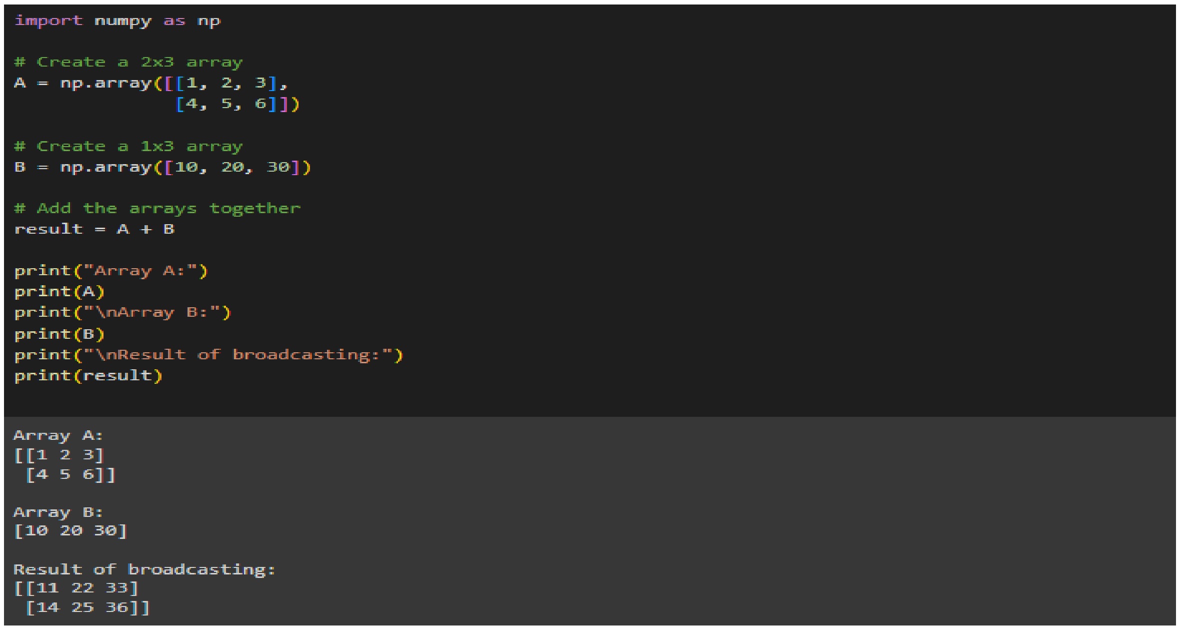


```python
import numpy as np

# Creating an array with shape (3, 4)
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])

print("Original array:")
print(arr)
print("Shape of the array:", arr.shape)

# Reshaping the array to shape (2, 6)
reshaped_arr = arr.reshape(2, 6)

print("\nReshaped array:")
print(reshaped_arr)
print("Shape of the reshaped array:", reshaped_arr.shape)
```

```
Original array:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Shape of the array: (3, 4)

Reshaped array:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
Shape of the reshaped array: (2, 6)
```

6. **Broadcasting:**

Broadcasting is a concept in NumPy that allows arrays of different shapes to be combined in arithmetic operations. It enables efficient computation and reduces the need for explicit looping over array elements.



```python
import numpy as np

# Create a 2x3 array
A = np.array([[1, 2, 3],
              [4, 5, 6]])

# Create a 1x3 array
B = np.array([10, 20, 30])

# Add the arrays together
result = A + B

print("Array A:")
print(A)
print("\nArray B:")
print(B)
print("\nResult of broadcasting:")
print(result)
```

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

Array B:
[10 20 30]

Result of broadcasting:
[[11 22 33]
 [14 25 36]]
```

7. **Numpy stacking:**

NumPy stacking refers to the process of combining multiple arrays along a new axis to create a new array. It allows you to stack arrays horizontally or vertically to form a larger array.

NumPy provides several functions for stacking arrays, including np.vstack() for vertical stacking and np.hstack() for horizontal stacking.

Here's a brief overview of each stacking function:

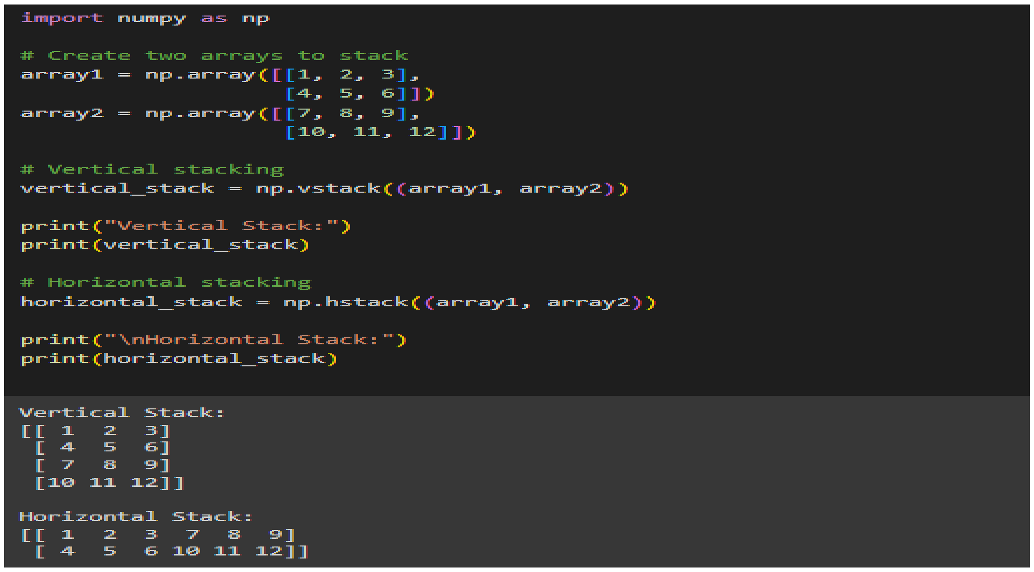
Vertical stacking (np.vstack()):

- np.vstack() stacks arrays vertically, meaning it combines arrays along the vertical axis (rows).
- The arrays being stacked must have the same number of columns (i.e., the same shape along the second axis).
- It creates a new array with the stacked arrays placed one below the other.

Horizontal stacking (np.hstack()):

- np.hstack() stacks arrays horizontally, meaning it combines arrays along the horizontal axis (columns).
- The arrays being stacked must have the same number of rows (i.e., the same shape along the first axis).
- It creates a new array with the stacked arrays placed side by side.

Here's an example demonstrating both vertical and horizontal stacking:



```
import numpy as np

# Create two arrays to stack
array1 = np.array([[1, 2, 3],
                   [4, 5, 6]])
array2 = np.array([[7, 8, 9],
                   [10, 11, 12]])

# Vertical stacking
vertical_stack = np.vstack((array1, array2))

print("Vertical Stack:")
print(vertical_stack)

# Horizontal stacking
horizontal_stack = np.hstack((array1, array2))

print("\nHorizontal Stack:")
print(horizontal_stack)
```

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

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

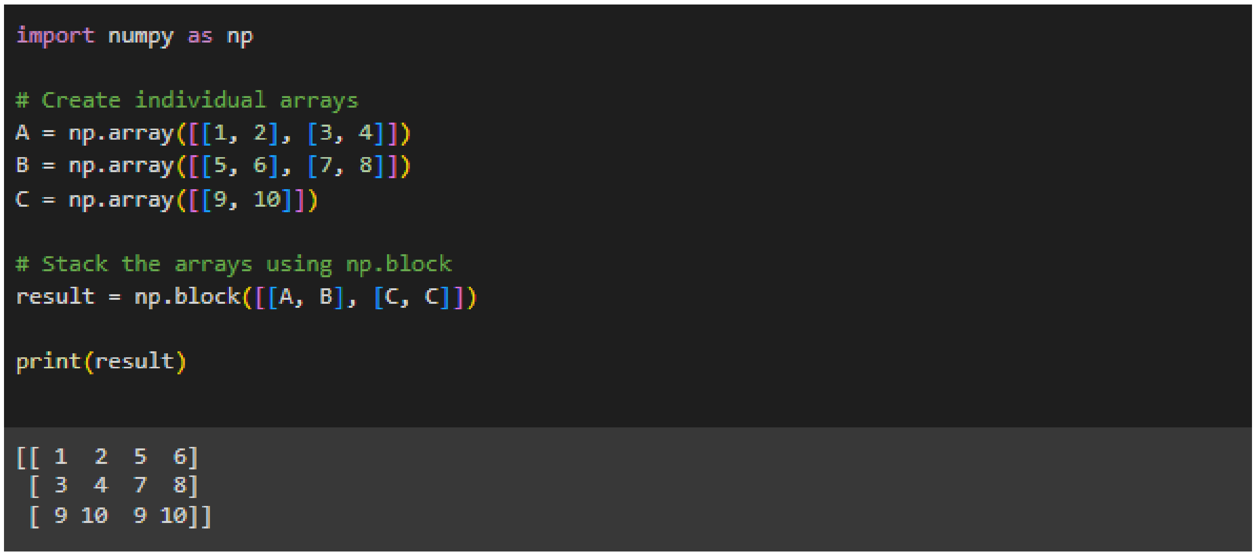
8. **np.block:**

"np.block" is a function provided by the NumPy library in Python that allows you to construct arrays by stacking blocks of arrays along different dimensions. It provides more flexibility than np.vstack() and np.hstack() as it allows for the creation of multi-dimensional arrays by combining arrays of different shapes and sizes.

Syntax:

numpy.block(arrays)

Assemble an nd-array from nested lists of blocks.



```python
import numpy as np

# Create individual arrays
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.array([[9, 10]])

# Stack the arrays using np.block
result = np.block([[A, B], [C, C]])

print(result)
```

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

9. **np.hsplit VS np.vsplit VS np.dsplit:**
   a. np.hsplit (Horizontal Split):
      i. np.hsplit is a function in NumPy used to split an array horizontally along its horizontal axis (axis 1), resulting in multiple sub-arrays.
      ii. The syntax is **np.hsplit(array, indices_or_sections).**
      iii. array: The array to be split.
      iv. indices_or_sections: Either an integer specifying the number of equally-sized sub-arrays to create, or a list of indices at which to split the array.
      v. It returns a list of sub-arrays.

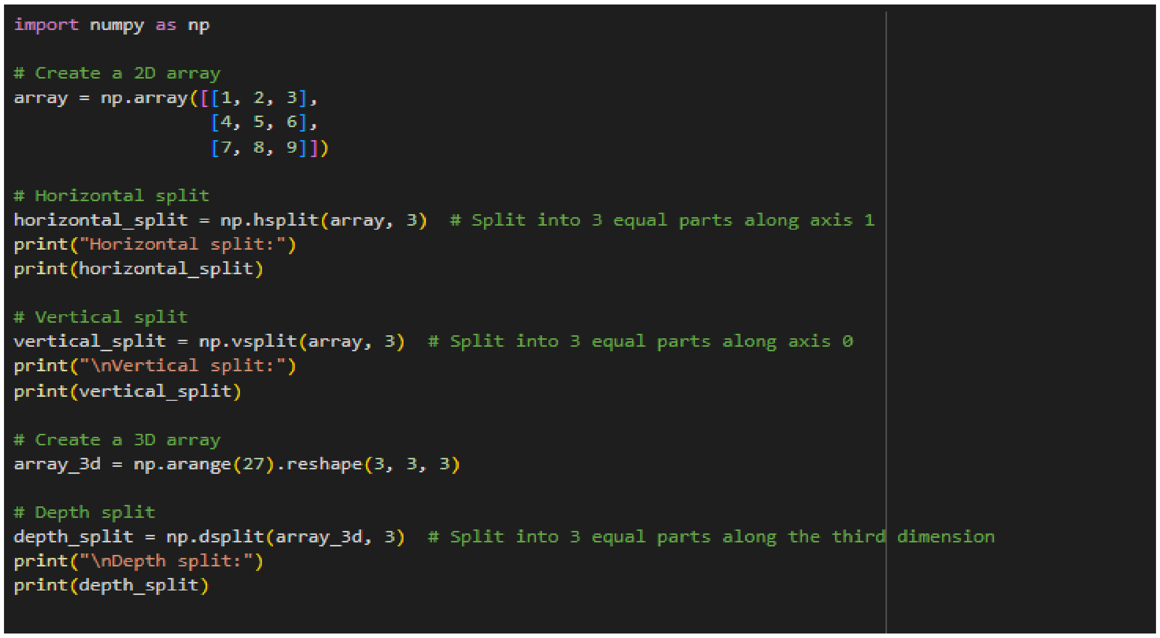
b. np.vsplit (Vertical Split):

   i. np.vsplit is a function in NumPy used to split an array vertically along its vertical axis (axis 0), resulting in multiple sub-arrays.
   ii. The syntax is **np.vsplit(array, indices_or_sections).**
   iii. array: The array to be split.
   iv. indices_or_sections: Either an integer specifying the number of equally-sized sub-arrays to create, or a list of indices at which to split the array.
   v. It returns a list of sub-arrays.

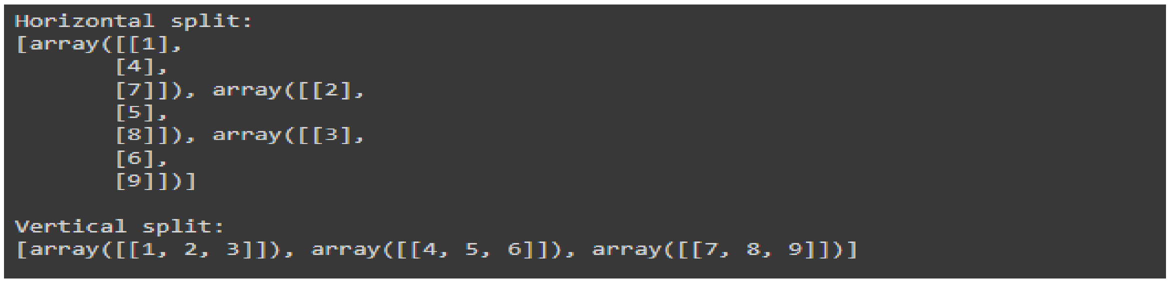
c. np.dsplit (Depth Split):

   i. np.dsplit is a function in NumPy used to split an array along its third dimension (depth-wise), resulting in multiple sub-arrays.
   ii. The syntax is **np.dsplit(array, indices_or_sections).**
   iii. array: The 3D array to be split.
   iv. indices_or_sections: Either an integer specifying the number of equally-sized sub-arrays to create, or a list of indices at which to split the array.
   v. It returns a list of sub-arrays.

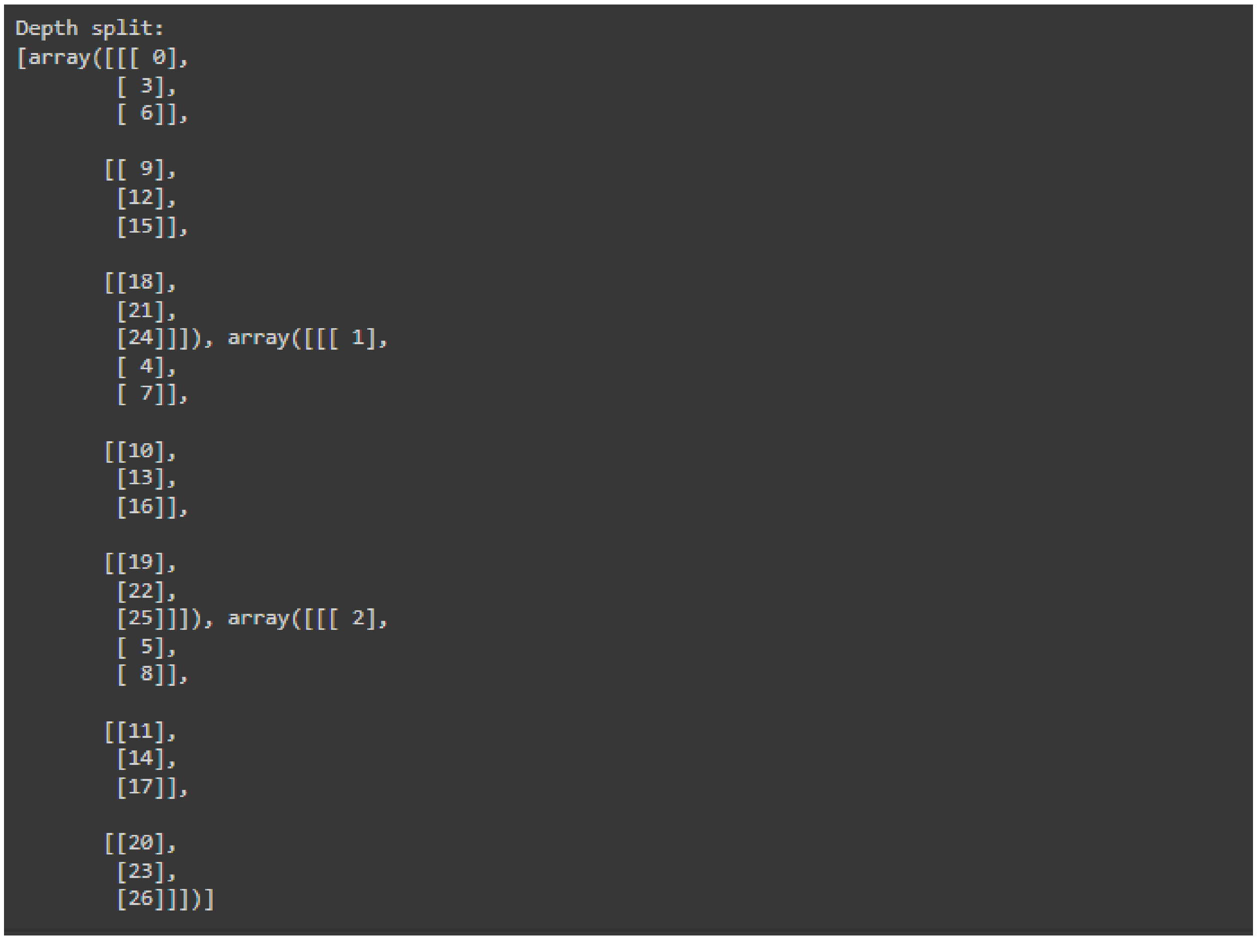
Here's an example demonstrating the usage of these functions:



```python
import numpy as np

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

# Horizontal split
horizontal_split = np.hsplit(array, 3)  # Split into 3 equal parts along axis 1
print("Horizontal split:")
print(horizontal_split)

# Vertical split
vertical_split = np.vsplit(array, 3)  # Split into 3 equal parts along axis 0
print("\nVertical split:")
print(vertical_split)

# Create a 3D array
array_3d = np.arange(27).reshape(3, 3, 3)

# Depth split
depth_split = np.dsplit(array_3d, 3)  # Split into 3 equal parts along the third dimension
print("\nDepth split:")
print(depth_split)
```



```
Horizontal split:
[array([[1],
       [4],
       [7]]), array([[2],
       [5],
       [8]]), array([[3],
       [6],
       [9]])]

Vertical split:
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
```



```
Depth split:
[array([[[ 0],
        [ 3],
        [ 6]],

       [[ 9],
        [12],
        [15]],

       [[18],
        [21],
        [24]]]), array([[[ 1],
        [ 4],
        [ 7]],

       [[10],
        [13],
        [16]],

       [[19],
        [22],
        [25]]]), array([[[ 2],
        [ 5],
        [ 8]],

       [[11],
        [14],
        [17]],

       [[20],
        [23],
        [26]]])]
```

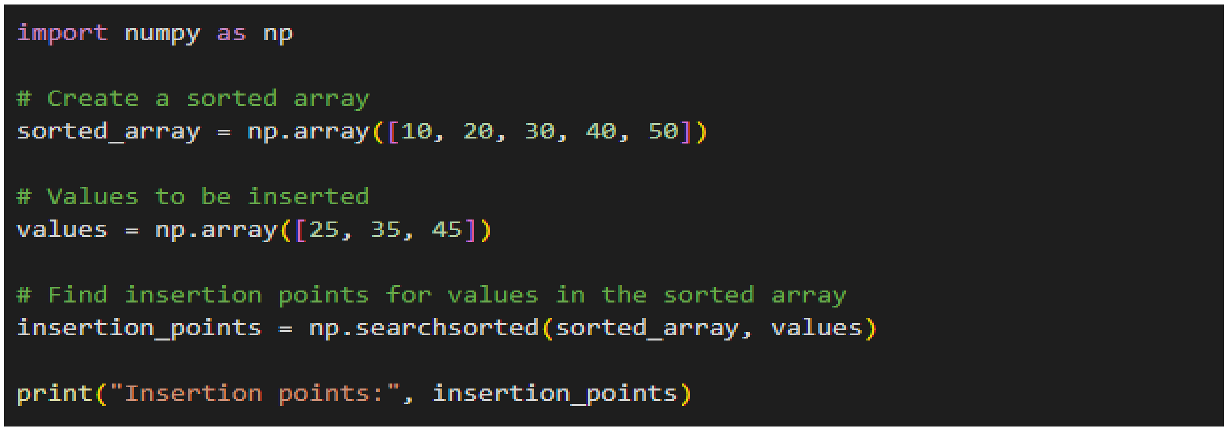
10. np.searchsorted:

"np.searchsorted" is a function provided by the NumPy library in Python. It is used to find the indices where elements should be inserted into a sorted array to maintain the order of the array.

Syntax:

numpy.searchsorted(a, v, side='left', sorter=None)

Find indices where elements should be inserted to maintain order.



```python
import numpy as np

# Create a sorted array
sorted_array = np.array([10, 20, 30, 40, 50])

# Values to be inserted
values = np.array([25, 35, 45])

# Find insertion points for values in the sorted array
insertion_points = np.searchsorted(sorted_array, values)

print("Insertion points:", insertion_points)
```

11. np.sort and argsort:

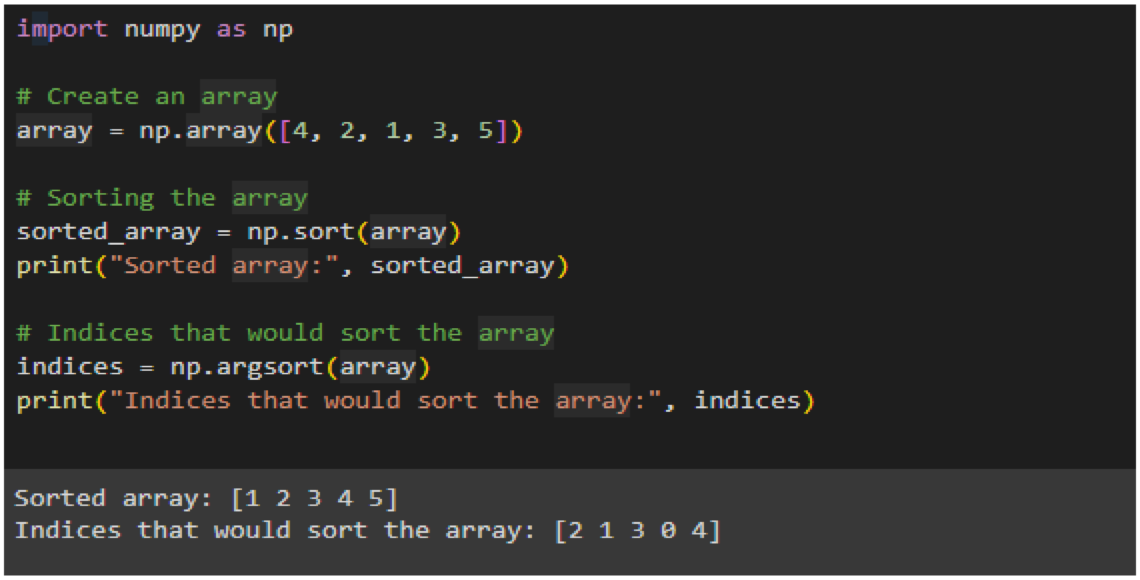
a. np.sort:

   i. np.sort is a function used to sort elements of an array along a specified axis.
   ii. The syntax is **np.sort(array, axis=-1, kind='quicksort', order=None).**
   iii. array: The input array to be sorted.
   iv. axis: Optional. The axis along which to sort the array. Default is -1, indicating the last axis.
   v. kind: Optional. The sorting algorithm to be used. Default is 'quicksort'.
   vi. order: Optional. If the array is structured (i.e., it has fields defined), this parameter specifies the field(s) to use when sorting.
   vii. It returns a new array containing the sorted elements.

b. np.argsort:

   i. np.argsort is a function used to return the indices that would sort an array.
   ii. The syntax is **np.argsort(array, axis=-1, kind='quicksort', order=None)**.
   iii. array: The input array to be sorted.
   iv. axis: Optional. The axis along which to sort the array. Default is -1, indicating the last axis.
   v. kind: Optional. The sorting algorithm to be used. Default is 'quicksort'.
   vi. order: Optional. If the array is structured (i.e., it has fields defined), this parameter specifies the field(s) to use when sorting.
   vii. It returns an array of indices that would sort the input array.

Here's an example illustrating the usage of np.sort and np.argsort:



```
import numpy as np

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

# Sorting the array
sorted_array = np.sort(array)
print("Sorted array:", sorted_array)

# Indices that would sort the array
indices = np.argsort(array)
print("Indices that would sort the array:", indices)
```

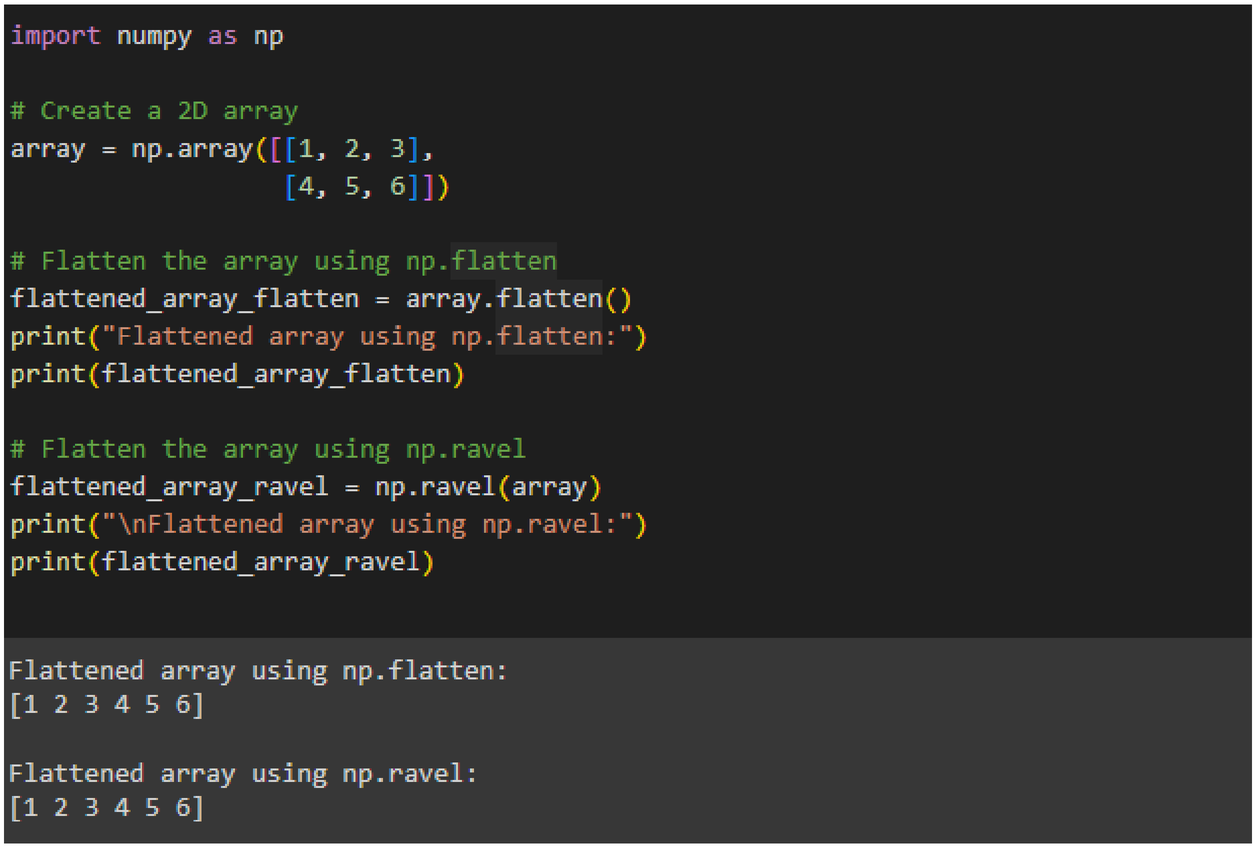
```
Sorted array: [1 2 3 4 5]
Indices that would sort the array: [2 1 3 0 4]
```

12. np.flatten vs np.ravel:

a. np.flatten:
   i. np.flatten is a method of NumPy arrays that returns a copy of the array collapsed into one dimension.
   ii. The syntax is **array.flatten(order='C').**
   iii. array: The input array to be flattened.
   iv. order: Optional. Specifies the order of flattening. It can be 'C' (row-major, default) or 'F' (column-major).
   v. It returns a one-dimensional copy of the input array, and modifying the flattened array does not affect the original array.

b. np.ravel:
   i. np.ravel is a function in NumPy that returns a flattened array without making a copy if possible. If a copy is required, it returns one.
   ii. The syntax is **np.ravel(array, order='C').**
   iii. array: The input array to be flattened.
   iv. order: Optional. Specifies the order of flattening. It can be 'C' (row-major, default) or 'F' (column-major).
   v. It returns a one-dimensional view of the input array, and modifying the flattened array may affect the original array.

Here's an example illustrating the usage of np.flatten and np.ravel:



```
import numpy as np

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

# Flatten the array using np.flatten
flattened_array_flatten = array.flatten()
print("Flattened array using np.flatten:")
print(flattened_array_flatten)

# Flatten the array using np.ravel
flattened_array_ravel = np.ravel(array)
print("\nFlattened array using np.ravel:")
print(flattened_array_ravel)
```

```
Flattened array using np.flatten:
[1 2 3 4 5 6]

Flattened array using np.ravel:
[1 2 3 4 5 6]
```

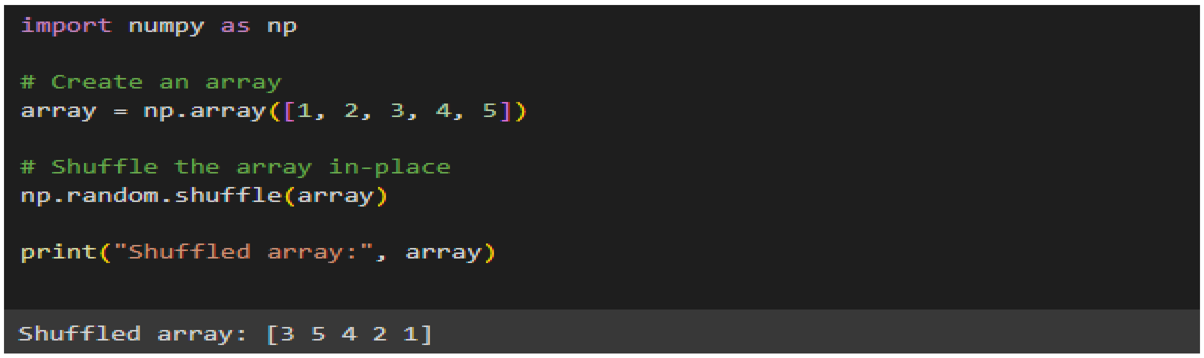
13. np.shuffle:

"np.shuffle" is a function provided by NumPy that shuffles the elements of an array in-place, meaning it modifies the original array. This function randomly reorders the elements along the first axis of the array.

Syntax:

random.shuffle(x)

Modify a sequence in-place by shuffling its contents.



```
import numpy as np

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

# Shuffle the array in-place
np.random.shuffle(array)

print("Shuffled array:", array)
```

```
Shuffled array: [3 5 4 2 1]
```

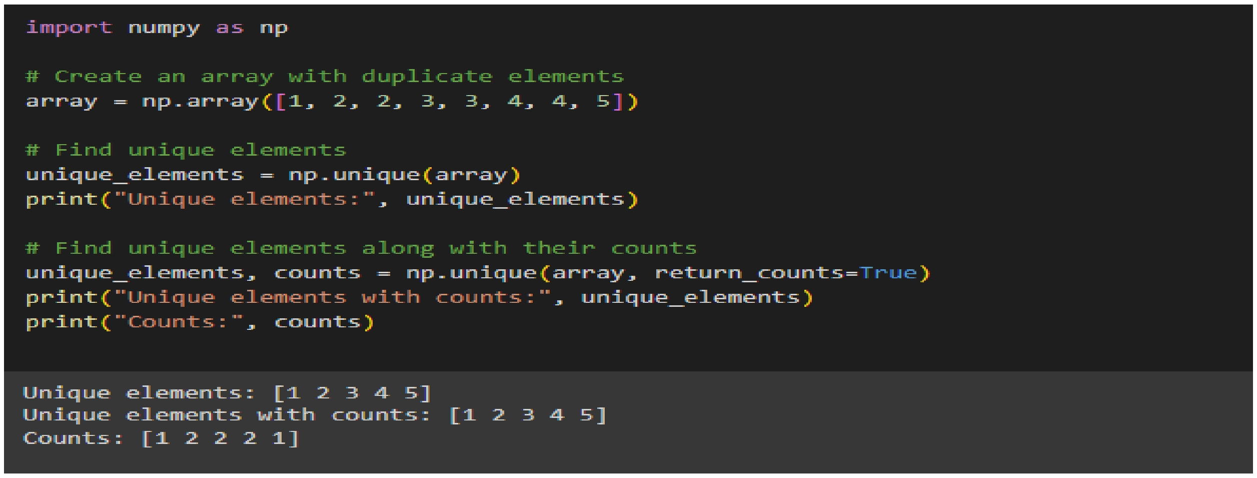
14. np.unique:

"np.unique" is a function provided by NumPy that returns the unique elements of an array. It returns the sorted unique elements of an array along with the indices that can be used to reconstruct the original array.

Syntax:

numpy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *, equal_nan=True)

Find the unique elements of an array.



```python
import numpy as np

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

# Find unique elements
unique_elements = np.unique(array)
print("Unique elements:", unique_elements)

# Find unique elements along with their counts
unique_elements, counts = np.unique(array, return_counts=True)
print("Unique elements with counts:", unique_elements)
print("Counts:", counts)
```

```
Unique elements: [1 2 3 4 5]
Unique elements with counts: [1 2 3 4 5]
Counts: [1 2 2 2 1]
```

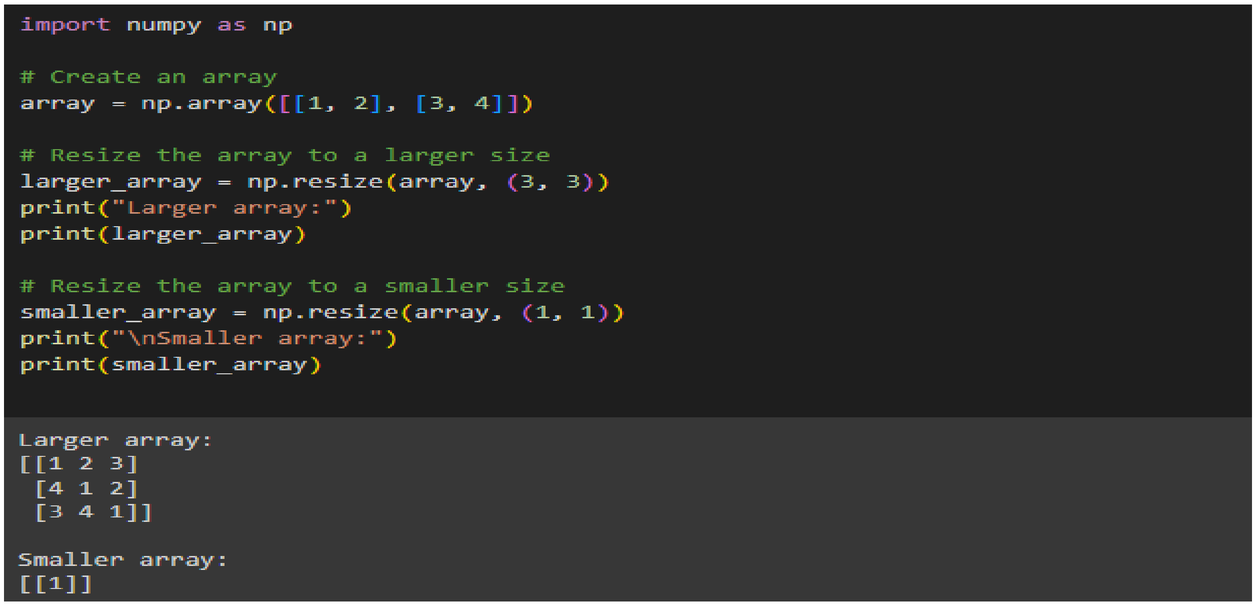
15. np.resize:

"np.resize" is a function provided by NumPy that changes the shape of an array without changing its data. It can increase or decrease the size of the array as needed, either by adding elements or by removing them.

Syntax:

numpy.resize(a, new_shape)

Return a new array with the specified shape.



```python
import numpy as np

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

# Resize the array to a larger size
larger_array = np.resize(array, (3, 3))
print("Larger array:")
print(larger_array)

# Resize the array to a smaller size
smaller_array = np.resize(array, (1, 1))
print("\nSmaller array:")
print(smaller_array)
```

```
Larger array:
[[1 2 3]
 [4 1 2]
 [3 4 1]]

Smaller array:
[[1]]
```

16. Transpose VS Swapaxes:

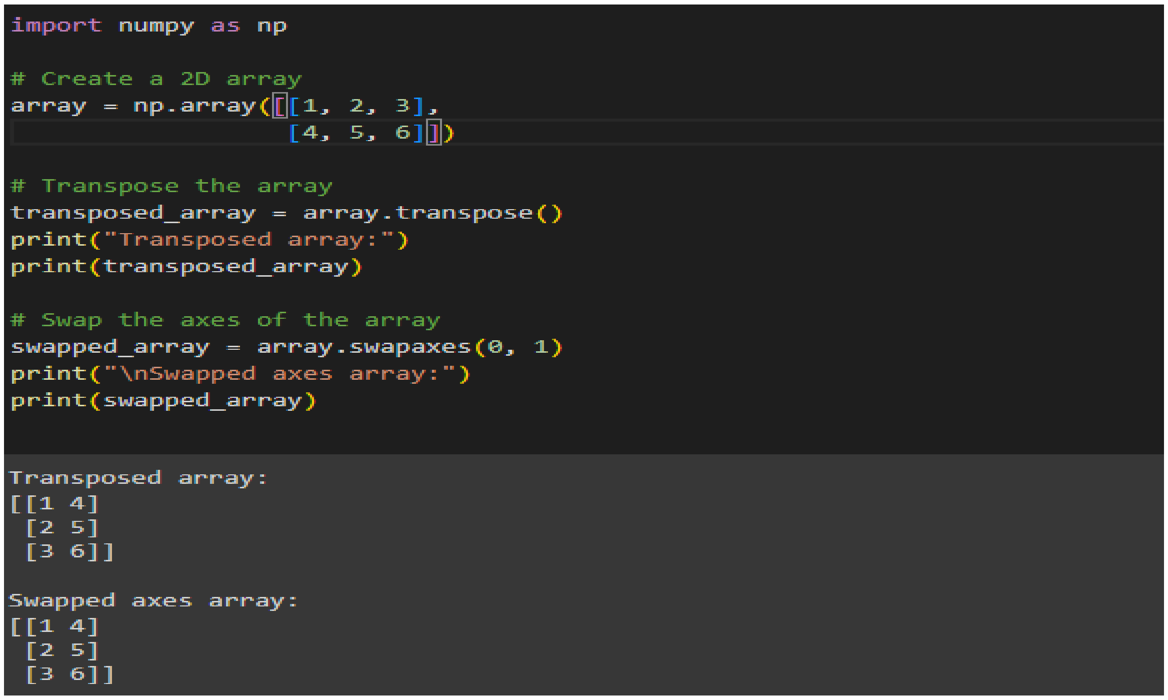
Transpose (transpose):

- transpose is a method of NumPy arrays used to permute the dimensions of an array.
- The syntax is **array.transpose(*axes).**
- array: The input array to be transposed.
- *axes: Optional. The new order of dimensions. By default, the axes are reversed.
- It returns a view of the input array with its axes permuted according to the specified order.

Swapaxes (swapaxes):

- swapaxes is a method of NumPy arrays used to swap the two specified axes of an array.
- The syntax is **array.swapaxes(axis1, axis2).**
- array: The input array to be operated on.
- axis1 and axis2: The two axes to be swapped.
- It returns a view of the input array with the specified axes swapped.

Here's an example illustrating the usage of both transpose and swapaxes:

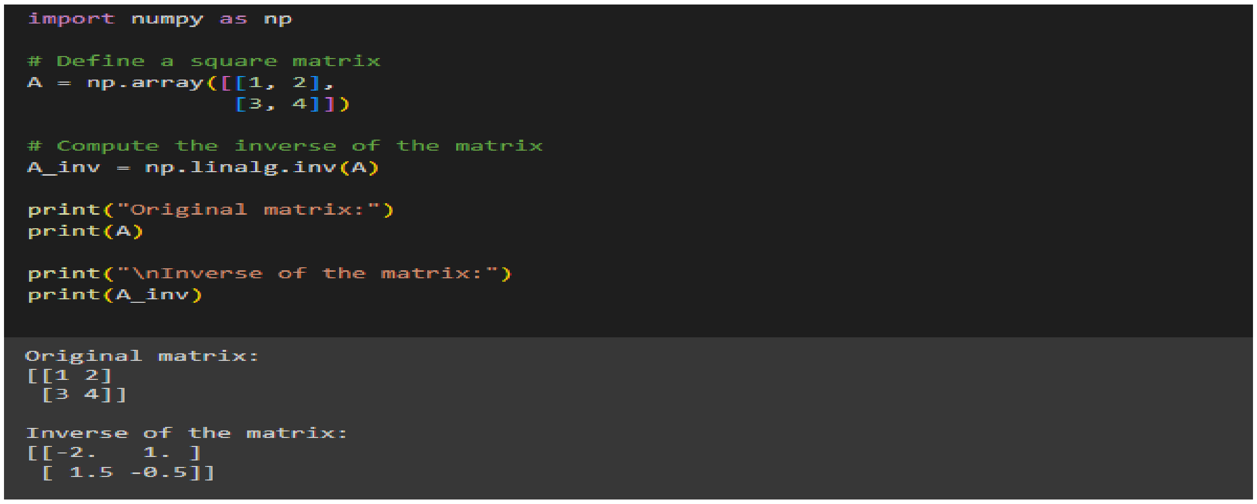


```
import numpy as np

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

# Transpose the array
transposed_array = array.transpose()
print("Transposed array:")
print(transposed_array)

# Swap the axes of the array
swapped_array = array.swapaxes(0, 1)
print("\nSwapped axes array:")
print(swapped_array)
```

```
Transposed array:
[[1 4]
 [2 5]
 [3 6]]

Swapped axes array:
[[1 4]
 [2 5]
 [3 6]]
```

17. **Inverse:**

In NumPy, the term "inverse" typically refers to finding the inverse of a square matrix. NumPy provides the function **np.linalg.inv()** to compute the inverse of a matrix.



```
import numpy as np

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

# Compute the inverse of the matrix
A_inv = np.linalg.inv(A)

print("Original matrix:")
print(A)

print("\nInverse of the matrix:")
print(A_inv)
```

```
Original matrix:
[[1 2]
 [3 4]]

Inverse of the matrix:
[[-2.   1. ]
 [ 1.5 -0.5]]
```

18. Power VS determinant:

Power of a Matrix:

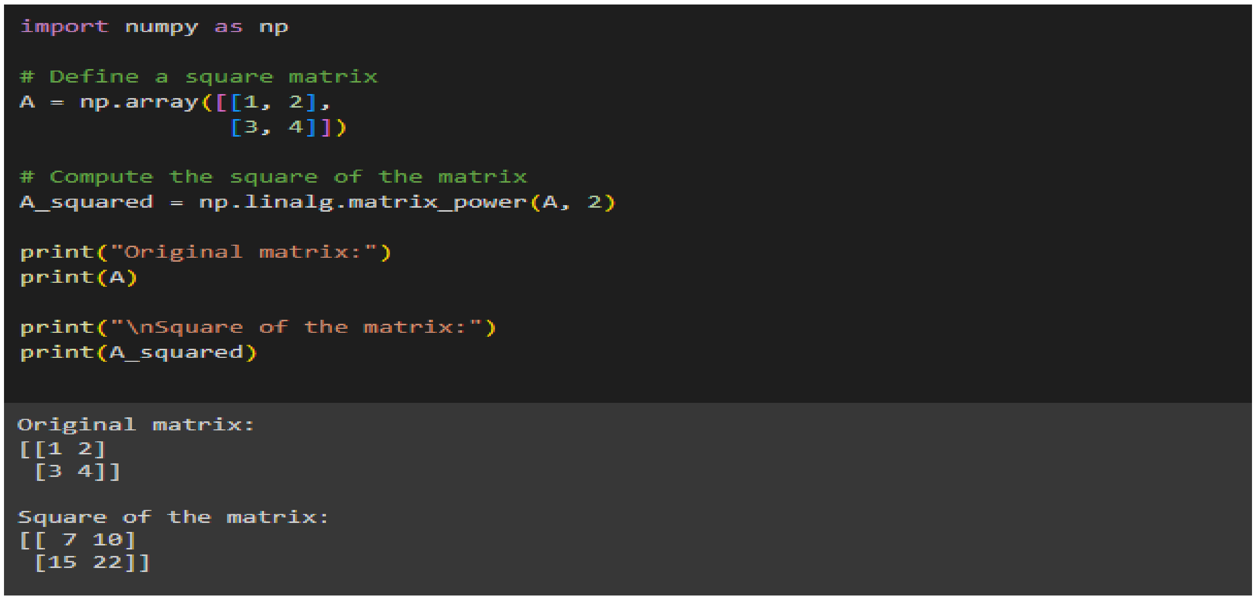
a. NumPy provides the numpy.linalg.matrix_power() function to compute the power of a square matrix.
b. The syntax is numpy.linalg.matrix_power(matrix, n).

c. matrix: The input square matrix.

d. n: The exponent to which the matrix is raised.

e. It returns the result of raising the input matrix to the power of n.

Here's an example of computing the power of a matrix using NumPy:



```
import numpy as np

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

# Compute the square of the matrix
A_squared = np.linalg.matrix_power(A, 2)

print("Original matrix:")
print(A)

print("\nSquare of the matrix:")
print(A_squared)
```

```
Original matrix:
[[1 2]
 [3 4]]

Square of the matrix:
[[ 7 10]
 [15 22]]
```

Determinant of a Matrix:

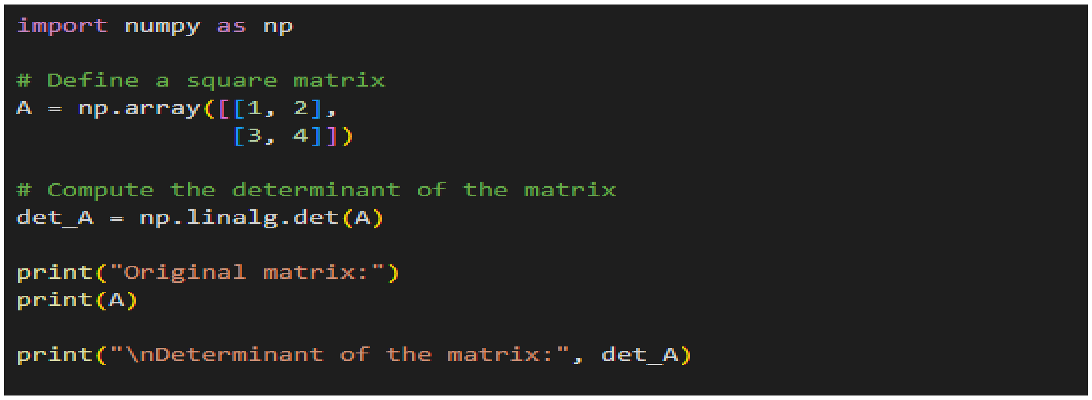
a. NumPy provides the numpy.linalg.det() function to compute the determinant of a square matrix.

b. The syntax is numpy.linalg.det(matrix).

c. matrix: The input square matrix.

d. It returns the determinant of the input matrix as a scalar value.

Here's an example of computing the determinant of a matrix using NumPy:



```
import numpy as np

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

# Compute the determinant of the matrix
det_A = np.linalg.det(A)

print("Original matrix:")
print(A)

print("\nDeterminant of the matrix:", det_A)
```