```
%pip install numpy
import numpy as np
```

```
Requirement already satisfied: numpy in c:\users\devas\appdata\local\
programs\python\python313\lib\site-packages (2.2.2)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 24.3.1 -> 25.0
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Array creation

```
# 1D array
np.array([1, 2, 3])
```

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

```
# 2D array
np.array([[1, 2], [3, 4]])
```

```
array([[1, 2],
       [3, 4]])
```

```
# Array of zeros
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
# Array of ones
np.ones((3, 3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
# Array of evenly spaced values
np.arange(0, 10, 2)
```

```
array([0, 2, 4, 6, 8])
```

```
# Array of 5 values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Element-Wise Operations

```
arr = np.array([1, 2, 3])
print(arr + 2)  # [3, 4, 5]
print(arr * 2)  # [2, 4, 6]
```

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

Matrix Multiplication

```
mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[5, 6], [7, 8]])
print(np.dot(mat1, mat2))  # Matrix product
```

```
[[19 22]
 [43 50]]
```

Indexing

```
arr = np.array([10, 20, 30, 40])
print(arr[1])  # 20
```

```
20
```

Slicing

```
print(arr[1:3])  # [20, 30]
```

```
[20 30]
```

Shape and Size

```
arr = np.array([[1, 2], [3, 4]])
print(arr.shape)  # (2, 2)
print(arr.size)   # 4
```

```
(2, 2)
4
```

Reshape

```
print(arr.reshape(4, 1))  # Reshape to a 4x1 array
# print(arr.reshape(4, 1, 1))  # Reshape to a 4x1x1 array
print(arr.reshape(1, 4))  # Reshape to a 1x4 array
```

```
[[1]
 [2]
 [3]
 [4]]
[[1 2 3 4]]
```

Aggregation

```
print(arr.sum())  # Sum of all elements
print(arr.mean())  # Mean value
print(arr.max())  # Maximum value
```

```
10
2.5
4
```

ndarray.shape

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

```
(2, 1, 3, 1)
```

```
needle = [[[[1], [1], [1]]], [[[1], [1], [1]]]]

elem = 1
l0 = [elem]
l1 = [l0, l0, l0]
l2 = [l1]
l3 = [l2, l2]

l3 == needle
```

```
True
```

ndarray.ndim

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.ndim)  # Output: 2 (2D array)
```

```
2
```

ndarray.size

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.size)  # Output: 6
```

```
6
```

ndarray.dtype

```
arr = np.array([1, 2, 3])
print(arr.dtype)  # Output: int64 (or int32, depending on the
platform)
```

```
int64
```

ndarray.itemsize

```
arr = np.array([1, 2, 3])
print(arr.itemsize)  # Output: 8 bytes (if dtype is int64, 4 if int32)
```

```
8
```

ndarray.nbytes

```
arr = np.array([1, 2, 3])
print(arr.nbytes)  # Output: 24 bytes (3 elements × 8 bytes each)
```

```
24
```

ndarray.T (Transpose)

```
arr = np.array([[1, 2], [3, 4]])
# [[1 2]
#  [3 4]]
print(arr.T)
# Output: row 1 becomes column 1, row 2 becomes column 2, etc.
# [[1 3]
#  [2 4]]
```

```
[[1 3]
 [2 4]]
```

ndarray.flags

```
# Provides information about the memory layout of the array.
arr = np.array([1, 2, 3])
print(arr.flags)
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

ndarray.real and ndarray.imag

```
arr = np.array([1 + 2j, 3 + 4j])
print(arr.real)  # Output: [1. 3.]
print(arr.imag)  # Output: [2. 4.]

arr.real.dtype, arr.imag.dtype  # Output: float64, float64
```

```
[1. 3.]
[2. 4.]

(dtype('float64'), dtype('float64'))
```

ndarray.strides

Returns the number of bytes to step in each dimension when traversing an array.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.strides)  # Output: (24, 8) (depending on platform)

(24, 8)

arr[int((arr.nbytes / arr.strides[0]) - 1)]

array([4, 5, 6])
```

ndarray.base

If the array is a view of another array, this attribute points to the original array.

```
arr = np.array([1, 2, 3])
view = arr[1:]
print(view.base is arr)  # Output: True

True

arr = np.array([1, 2, 3])
view = arr[1:] * 2
print(view.base is arr)

False
```

ndarray.flat

Returns a 1D iterator over the array.

```
arr = np.array([[1, 2], [3, 4]])
for element in arr.flat:
    print(element)  # Outputs: 1 2 3 4

1
2
3
4

arr = np.array([[1, 2], [3, 4]])
for element in arr:
```

```
    for subelement in element:
        print(subelement)
```

```
1
2
3
4
```

```
arr = np.array([[1, 2], [3, 4]])
for element in arr.flatten():
    print(element)
```

```
1
2
3
4
```

Random Array Creation

```
# Creates an array of random values between 0 and 1.
np.random.rand(2, 3)
```

```
array([[0.96060088, 0.98697505, 0.53191142],
       [0.18505828, 0.27100305, 0.29615902]])
```

```
# Creates an array of random integers.
np.random.randint(1, 10, (3, 3))
```

```
array([[4, 6, 9],
       [8, 3, 8],
       [6, 3, 3]], dtype=int32)
```

Joining Arrays

```
# Concatenate arrays

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Join along the default axis (1D arrays)
joined = np.concatenate((arr1, arr2))
print(joined)  # Output: [1 2 3 4 5 6]
```

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

```
# Vertical Stack

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

vstacked = np.vstack((arr1, arr2))
```

```
print(vstacked)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

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

```
# Depth Stack

arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([4, 5, 6, 7])

dstacked = np.dstack((arr1, arr2))
print(dstacked)
# Output:
# [[[1 4]
#   [2 5]
#   [3 6]
#   [4 7]]]
```

```
[[[1 4]
  [2 5]
  [3 6]
  [4 7]]]
```

Splitting Arrays

Splits an array into equal parts.

```
arr = np.array([1, 2, 3, 4, 5, 6])

# Split into 3 parts
split = np.split(arr, 3)
print(split)  # Output: [array([1, 2]), array([3, 4]), array([5, 6])]
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Splits an array into unequal parts.

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

# Split into 3 parts (last part may be uneven)
split = np.array_split(arr, 3)
print(split)  # Output: [array([1, 2, 3]), array([4, 5]), array([6,
7])]
```

```
[array([1, 2, 3]), array([4, 5]), array([6, 7])]
```

Splitting Multi-Dimensional Arrays

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
# [[1 2]
#  [3 4]
#  [5 6]]

# Horizontal split
h_split = np.hsplit(arr, 2)
print(h_split)
# Output: [array([[1], [3], [5]]), array([[2], [4], [6]])]

# Vertical split
v_split = np.vsplit(arr, 3)
print(v_split)
# Output: [array([[1, 2]]), array([[3, 4]]), array([[5, 6]])]
```

```
[array([[1],
       [3],
       [5]]), array([[2],
       [4],
       [6]])]
[array([[1, 2]]), array([[3, 4]]), array([[5, 6]])]
```

Searching in Arrays

Using np.where()

Finds the indices of elements that satisfy a condition.

```
arr = np.array([10, 20, 30, 40, 50])
indices = np.where(arr > 25)
print(indices)  # Output: (array([2, 3, 4]),)
```

```
(array([2, 3, 4]),)
```

Using np.searchsorted()

Searches for a value and returns the index where it should be inserted to maintain order.

```
arr = np.array([1, 3, 5, 7])

# Find the index to insert 4
index = np.searchsorted(arr, 4)
print(index)  # Output: 2
```

```
2
```

Using np.isin()

Checks if elements exist in another array.

```
arr = np.array([1, 2, 3, 4, 5])
result = np.isin(arr, [2, 4, 6])
print(result)  # Output: [False  True False  True False]
```

```
[False  True False  True False]
```

Sorting Arrays

Using np.sort()

Sorts an array along a specified axis.

```
arr = np.array([5, 2, 9, 1, 7])
sorted_arr = np.sort(arr)
print(sorted_arr)  # Output: [1 2 5 7 9]
```

```
[1 2 5 7 9]
```

Sorting Multi-Dimensional Arrays

Sorts along each row or column.

```
arr = np.array([[3, 2, 7], [6, 5, 4]])

# Sort along rows
row_sorted = np.sort(arr, axis=1)
print(row_sorted)
# Output:
# [[2 3 7]
#  [4 5 6]]

# Sort along columns
col_sorted = np.sort(arr, axis=0)
print(col_sorted)
# Output:
# [[3 2 4]
#  [6 5 7]]
```

```
[[2 3 7]
 [4 5 6]]
[[3 2 4]
 [6 5 7]]
```

Using np.argsort()

Returns the indices that would sort the array.

```
arr = np.array([5, 2, 9, 1, 7])

indices = np.argsort(arr)
print(indices)  # Output: [3 1 0 4 2]
```

```
[3 1 0 4 2]
```

Iterating

a. Iterating Through 1D Arrays

You can iterate over the elements of a 1D array using a simple for loop

```
arr = np.array([1, 2, 3, 4])
for element in arr:
    print(element)
# Output: 1, 2, 3, 4
```

```
1
2
3
4
```

Iterating Through 2D Arrays

Iteration in 2D arrays occurs row by row

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
for row in arr:
    print(row)
# Output:
# [1 2]
# [3 4]
# [5 6]
```

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

Iterating Through Each Element in a Multi-Dimensional Array

To iterate over each element in a multidimensional array, use the nditer() function

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
# for element in np.nditer(arr):
#     print(element)
for element in arr.flat:
```

```
    print(element)
# Output: 1, 2, 3, 4, 5, 6
```

```
1
2
3
4
5
6
```

Iterating With Indices

Use the ndenumerate() function to get both the index and the value during iteration.

```
arr = np.array([[1, 2], [3, 4]])
for index, value in np.ndenumerate(arr):
    print(index, value)
# Output:
# (0, 0) 1
# (0, 1) 2
# (1, 0) 3
# (1, 1) 4
```

```
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Copying arrays

Shallow Copy (View)

A shallow copy creates a new array object that refers to the same data as the original array. Changes in one array will affect the other.

```
arr = np.array([1, 2, 3])
shallow_copy = arr.view()

shallow_copy[0] = 99
print(arr)           # Output: [99  2  3]
print(shallow_copy)  # Output: [99  2  3]
```

```
[99  2  3]
[99  2  3]
```

Deep Copy

A deep copy creates a new array object with a completely independent copy of the data. Changes in one array do not affect the other.

```
arr = np.array([1, 2, 3])
deep_copy = arr.copy()

deep_copy[0] = 99
print(arr)           # Output: [1  2  3]
print(deep_copy)     # Output: [99  2  3]
```

```
[1 2 3]
[99  2  3]
```

Check for Shared Memory

You can check whether two arrays share memory using the np.may_share_memory() function

```
arr = np.array([1, 2, 3])
shallow_copy = arr.view()
deep_copy = arr.copy()

print(np.may_share_memory(arr, shallow_copy))  # Output: True
print(np.may_share_memory(arr, deep_copy))     # Output: False
```

```
True
False
```

Identity array

Using np.identity()

Creates a square identity matrix of a specified size.

```
identity_matrix = np.identity(4)
print(identity_matrix)
# Output:
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]
```

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

Using np.eye()

Allows you to specify the number of rows and columns, and the position of the diagonal.

numpy.eye(N, M=None, k=0, dtype=float, order='C')

- N: Number of rows.
- M: (Optional) Number of columns. Defaults to N (square matrix).
- k: Index of the diagonal:
  - k=0 (default): Main diagonal.
  - k>0: Above the main diagonal.
  - k<0: Below the main diagonal.
- dtype: Data type of the returned array. Default is float.
- order: Memory layout ('C' for row-major, 'F' for column-major).

```
eye_matrix = np.eye(3, 4, k=1)  # 3 rows, 4 columns, diagonal shifted
by 1
print(eye_matrix)
# Output:
# [[0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]

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

matrix = np.eye(4)
print(matrix)

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

matrix = np.eye(3, 5)
print(matrix)

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

matrix = np.eye(4, k=1)  # Diagonal above the main diagonal
print(matrix)

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