Creating Arrays

Importing numpy

In [1]:
```
import numpy as np
```

Create a 1D array (like a list)

In [2]:
```
arr1=np.array([1,2,3,4])
print(arr1)
```

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

Create a 2D array (matrix)

In [7]:
```
arr2=np.array([[1,2,3,4],[5,6,7,8]])
print(arr2)
```

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

Create a 3D array

In [12]:
```
arr3=np.array([[[1,2], [3,4]], [[5,6],[7,9]]])
print(arr3)
```

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

 [[5 6]
  [7 9]]]
```

Initial placeholder

creating an array of zeros

In [16]:
```
#Creates a 2D array (3 rows and 4 columns) filled with zeros.

np.zeros((3,4))
```

Out[16]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Create an array of ones

In [17]:
```
# Creates a 3D array (2 blocks, each with 3 rows and 4 columns) filled with ones. The data type is int16 (16-bit integers).

np.ones((2,3,4), dtype=np.int16)
```

Out[17]:
```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

Create an array of evenly spaced values (step value)

In [20]:
```
# Creates a 1D array starting at 10, ending before 25, with a step size of 5.

np.arange(10,25,5)
```

Out[20]:
```
array([10, 15, 20])
```

Create an array of evenly spaced values (number of samples)

In [22]:
```
np.linspace(0,2,9)
```

Out[22]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

Create a constant array

In [24]:
```
# Creates a 2x2 array where every element is the constant 7.

np.full((2,2),7)
```

Out[24]:
```
array([[7, 7],
       [7, 7]])
```

Create a 2X2 identity matrix

In [26]:
```
np.eye(2)
```

Out[26]:
```
array([[1., 0.],
       [0., 1.]])
```

Create an array with random values

In [27]:
```
np.random.random((2,2))
```

Out[27]:
```
array([[0.43053429, 0.56402564],
       [0.82959718, 0.90675133]])
```

Create an empty array

In [28]:
```
np.empty((3,2))
```

Out[28]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

Input/ Output

Saving & Loading On Disk

Saving a single array using `np.save`

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

np.save('my_array', a)
```

Saving multiple arrays using `np.savez`

In [3]:
```
b=np.array([5,6,7,8])

np.savez('array.npz',a, b)
```

Loading an array using `np.load`

In [6]:
```
Loaded_a=np.load('my_array.npy')
Loaded_a
```

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

Saving & Loading Text Files

Loading a text file using `np.loadtxt`

In [7]:
```
np.loadtxt("myfile.txt")
```

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

Loading CSV file using `np.genfromtxt`

In [9]:
```
np.genfromtxt("myfile.csv", delimiter=',')
```

Out[9]:
```
array([[1., 2., 3., 4., 6., 8., 9.],
       [2., 3., 9., 3., 5., 8., 2.]])
```

Saving an array to a text file using `np.savetxt`

In [13]:
```
a = np.array([[1, 2, 3], [4, 5, 6]])
np.savetxt('a_arr.txt',a, delimiter=' ')
```

Inspecting Your array

In [14]:
```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([1.5, 2.5, 3.5])
e = np.array([1, 2, 3, 4, 5])
```

Array dimensions

In [15]:
```
a.shape
```

Out[15]:
```
(2, 3)
```

Length of array

In [17]:
```
print(len(a))
print(len(b))
```

```
2
3
```

Number of array dimensions

In [19]:
```
a.ndim
```

Out[19]:
```
2
```

Number of array elements

In [20]:
```
e.size
```

Out[20]: 5

Name of data type

In [21]:
```
b.dtype.name
```

Out[21]: 'float64'

Convert an array to a different type

In [22]:
```
b.astype(int)
```

Out[22]: array([1, 2, 3])

Asking For Help

In [28]:
```
np.info(np.ndarray.dtype)
```

```
Data-type of the array's elements.

.. warning::

    Setting ``arr.dtype`` is discouraged and may be deprecated in the
    future.  Setting will replace the ``dtype`` without modifying the
    memory (see also `ndarray.view` and `ndarray.astype`).

Parameters
----------
None

Returns
-------
d : numpy dtype object

See Also
--------
ndarray.astype : Cast the values contained in the array to a new data-type.
ndarray.view : Create a view of the same data but a different data-type.
numpy.dtype

Examples
--------
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

Array Mathematics

Arithmatic operation

In [29]:
```
import numpy as np
```

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

b = np.array([[1.5, 2.0, 3.0],
              [1.0, 2.0, 3.0]])

e = np.full((2, 2), 7)  # e.g., 2x2 array filled with 7
f = np.eye(2)           # 2x2 identity matrix
```

In [34]:
```
# Subtraction
g=a-b
print(g)
```

```
[[-0.5  0.   0. ]
 [ 3.   3.   3. ]]
```

In [44]:
```
# Subtraction
np.subtract(a,b)
```

Out[44]:
```
array([[-0.5,  0. ,  0. ],
       [ 3. ,  3. ,  3. ]])
```

In [38]:
```
# Addition
a+b
```

Out[38]:
```
array([[2.5, 4. , 6. ],
       [5. , 7. , 9. ]])
```

In [37]:
```
# Addition
np.add(a,b)
```

Out[37]:
```
array([[2.5, 4. , 6. ],
       [5. , 7. , 9. ]])
```

In [39]:
```
# Division
a/b
```

Out[39]:
```
array([[0.66666667, 1.        , 1.        ],
       [4.        , 2.5       , 2.        ]])
```

In [40]:
```
# Division
np.divide(a,b)
```

Out[40]:
```
array([[0.66666667, 1.        , 1.        ],
       [4.        , 2.5       , 2.        ]])
```

In [41]:
```
# Multiplication
a*b
```

Out[41]:
```
array([[ 1.5,  4. ,  9. ],
       [ 4. , 10. , 18. ]])
```

In [42]:
```
# Multiplication
np.multiply(a,b)
```

Out[42]:
```
array([[ 1.5,  4. ,  9. ],
       [ 4. , 10. , 18. ]])
```

In [47]:
```
# exponential: Calculate e^x for each element in b:
np.exp(b)
```

Out[47]:
```
array([[ 4.48168907,  7.3890561 , 20.08553692],
       [ 2.71828183,  7.3890561 , 20.08553692]])
```

In [48]:
```
# Square root
np.sqrt(b)
```

Out[48]:
```
array([[1.22474487, 1.41421356, 1.73205081],
       [1.        , 1.41421356, 1.73205081]])
```

In [49]:
```
# Sines of an array
np.sin(a)
```

Out[49]:
```
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ]])
```

In [50]:
```
# Cosines of an array
np.cos(b)
```

Out[50]:
```
array([[ 0.0707372 , -0.41614684, -0.9899925 ],
       [ 0.54030231, -0.41614684, -0.9899925 ]])
```

In [51]:
```
#Natural logarithm
np.log(a)
```

Out[51]:
```
array([[0.        , 0.69314718, 1.09861229],
       [1.38629436, 1.60943791, 1.79175947]])
```

In [52]:
```
# dot product
e.dot(f)
```

Out[52]:
```
array([[7., 7.],
       [7., 7.]])
```

Comparison

In [53]:
```
# Element-wise comparison
a==b
```

Out[53]:
```
array([[False,  True,  True],
       [False, False, False]])
```

In [58]:
```
# Element-wise comparison
a<3
```

Out[58]:
```
array([[ True,  True, False],
       [False, False, False]])
```

In [59]:
```
# Array-wise comparison
np.array_equal(a,b)
```

Out[59]:
```
False
```

Aggregate Functions

In [5]:
```
import numpy as np
```

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

b = np.array([[1.5, 2.0, 3.0],
              [1.0, 2.0, 4.0]])
```

Array-wise sum

In [2]:
```
a.sum()
```

Out[2]:
```
np.int64(21)
```

Array-wise minimum value

In [3]:
```
a.min()
```

Out[3]:
```
np.int64(1)
```

Maximum value of an array row

Finds the maximum value along each column (axis 0) of b.

In [6]:
```
b.max(axis=0)
```

Out[6]:
```
array([1.5, 2. , 4. ])
```

Cumulative sum of the elements

Computes the cumulative sum along each row (axis 1) of b.

In [7]:
```
b.cumsum(axis=1)
```

Out[7]:
```
array([[1.5, 3.5, 6.5],
       [1. , 3. , 7. ]])
```

Mean

Calculates the mean (average) of all elements in a.

In [8]:
```
a.mean()
```

Out[8]:
```
np.float64(3.5)
```

Median

Calculates the median value of all elements in b.

In [11]:
```
np.median(b)
```

Out[11]:
```
np.float64(2.0)
```

Correlation coefficient

Calculates the correlation coefficient matrix of array a.

In [18]:
```
np.corrcoef(b)
```

Out[18]:
```
array([[1., 1.],
       [1., 1.]])
```

Standard deviation

Calculates the standard deviation of all elements in b.

In [19]:
```
np.std(b)
```

Out[19]:
```
np.float64(0.9895285072531598)
```

Copying Arrays

In [1]:
```
import numpy as np

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

1. `h = a.view()`

- Creates a view of the array a.
- View means h and a share the same data in memory.
- Changes to h will affect a, and vice versa.

- However, h is a new array object.

In [2]:
```
h=a.view()
print(h)        #[1  2   3   4]

h[0]=100
print(a)         # [100   2   3   4]  <-- a changed too!
```

```
[1 2 3 4]
[100   2   3   4]
```

2. `np.copy(a)`

- Creates a deep copy of array a.
- This copy is a new array with its own data.
- Modifications to the copied array do not affect the original.

In [3]:
```
copy_a=np.copy(a)
print(a)            #[100   2   3   4]

copy_a[0]=200
print(a)            # [100   2   3   4]  <-- original unchanged
print(copy_a)       # [200   2   3   4]
```

```
[100   2   3   4]
[100   2   3   4]
[200   2   3   4]
```

3. `h = a.copy()`

- This is basically the same as np.copy(a).
- Creates a deep copy of a.
- Changes to h don't affect a.

In [6]:
```
h = a.copy()
h[1] = 300
```

```
print(a)  # [100   2   3   4]  (unchanged)
print(h)  # [100 300   3   4]
```

```
[100   2   3   4]
[100 300   3   4]
```

Sorting Arrays

In [7]:
```
import numpy as np

a = np.array([3, 1, 4, 2])
c = np.array([[9, 4, 7],
              [3, 8, 1]])
```

1. `a.sort()`

   Sorts the array in-place (modifies the original array).

   For a 1D array, it sorts all elements in ascending order.

In [8]:
```
print("Before sorting: ",a)
a.sort()
print("After sorting: ", a)
```

```
Before sorting:  [3 1 4 2]
After sorting:  [1 2 3 4]
```

2. `c.sort(axis=0)`

- Sorts the array along a specific axis.
- axis=0 means sorting each column individually.
- Sort is done in-place.

In [9]:
```
print("Before sorting: \n", c)
c.sort(axis=0)
print("After Sorting: \n",c)
```

```
Before sorting:
 [[9 4 7]
 [3 8 1]]
After Sorting:
 [[3 4 1]
 [9 8 7]]
```

Subsetting, Slicing, Indexing

In [37]:
```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])
c = np.array([[[3, 2, 1],
               [4, 5, 6]]])
```

1. Subsetting

Select element at index 2 in 1D array `a`:

In [13]:
```
print(a[2])
```

```
3
```

Select element at row 1, column 2 in 2D array `b`:

In [14]:
```
print(b[1,2])
```

```
6.0
```

2. Slicing

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: `[start:end]`.
- We can also define the step, like this: `[start:end:step]`.

- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

Slice elements from index 1 to index 5 from the following array:

In [23]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

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

Slice elements from index 4 to the end of the array:

In [24]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

```
[5 6 7]
```

Slice elements from the beginning to index 4 (not included):

In [25]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

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

Negative Slicing

Use the minus operator to refer to an index from the end:

Slice from the index 3 from the end to index 1 from the end:

In [27]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

```
[5 6]
```

STEP

Use the `step` value to determine the step of the slicing:

Return every other element from index 1 to index 5:

In [28]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

```
[2 4]
```

Return every other element from the entire array:

In [29]:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[::2])
```

```
[1 3 5 7]
```

Slicing 2-D Arrays

From the second element, slice elements from index 1 to index 4 (not included):

In [30]:
```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

```
[7 8 9]
```

From both elements, return index 2:

In [33]:
```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

```
[3 8]
```

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

In [32]:
```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

3. Boolean Indexing

Select elements from a that are less than 4:

In [34]:
```
import numpy as np
a = np.array([1, 2, 3, 4, 6, 7, 8])

print(a[a<4])
```

```
[1 2 3]
```

4. Fancy Indexing

Select elements from `b` at positions `(1,0)`, `(0,1)`, `(1,2)`, and `(0,0)`

In [38]:
```
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])

rows=[1, 0, 1, 0]
columns=[0, 1, 2, 0]
b[rows, columns]
```

Out[38]:
```
array([4. , 2. , 6. , 1.5])
```

Select a subset of `b`'s rows and columns:

In [39]:
```
print(b[[1, 0, 1]][:, [0, 1, 2, 0]])
# Explanation:
# Select rows 1, 0, 1 (in that order), then select columns 0, 1, 2, 0 from these rows
```

```
[[4.  5.  6.  4. ]
 [1.5 2.  3.  1.5]
 [4.  5.  6.  4. ]]
```

Array Manipulation

Transposing Array

In [49]:
```
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])

i=np.transpose(b)
print(i)
```

```
[[1.5 4. ]
 [2.  5. ]
 [3.  6. ]]
```

Transpose back to original 'b'

In [47]:
```
print(i.T)
```

```
[[1.5 2.  3. ]
 [4.  5.  6. ]]
```

Changing Array Shape

Flatten array `b`:

In [51]:
```
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])
```

```
print(b.ravel())
```

```
[1.5 2.  3.  4.  5.  6. ]
```

Reshape array `g` to 3 rows and columns 2: don't change data

In [56]:
```
g = np.arange(6)  # array([0, 1, 2, 3, 4, 5])

print(g.reshape(3,2))
```

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

Adding/Removing Elements

Resize array `h` to shape (2, 6):

In [64]:
```
h = np.full((2, 3), 7)  # 2x3 array filled with 7
print(h)
print()

h.resize(2,8)
print(h)
```

```
[[7 7 7]
 [7 7 7]]

[[7 7 7 7 7 7 0 0]
 [0 0 0 0 0 0 0 0]]
```

Append arrays `h` and `g`:

In [67]:
```
g = np.arange(6)            # array([0, 1, 2, 3, 4, 5])
h = np.full((2, 3), 7)      # 2x3 array filled with 7

print(g)
print()
```

```
print(h)
print()

print(np.append(g,h))
```

```
[0 1 2 3 4 5]

[[7 7 7]
 [7 7 7]]

[0 1 2 3 4 5 7 7 7 7 7 7]
```

Insert element 5 into `a` at index 1:

In [68]:
```
a = np.array([1, 2, 3])

print(np.insert(a,1,5))
```

```
[1 5 2 3]
```

Delete element at index 1 from array `a`:

In [70]:
```
a = np.array([1, 2, 3])

deleted=np.delete(a,[1])
print(deleted)
```

```
[1 3]
```

Combining Arrays

Concatenate arrays `a` and `d` along axis 0:

In [109…
```
a = np.array([1, 2, 3])
d = np.array([10, 15, 20])

print(np.concatenate((a,d)))
```

```
[ 1  2  3 10 15 20]
```

Stack arrays `a` and `b` vertically (row-wise):

In [121…
```
a = np.array([1, 2, 3])
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])

vstacked = np.vstack((a, b))
print(vstacked)
```

```
[[1.  2.  3. ]
 [1.5 2.  3. ]
 [4.  5.  6. ]]
```

Stack arrays `b` and `h` vertically (row-wise):

In [140…
```
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])

h = np.full((2, 3), 7)

np.r_[b, h]
```

Out[140…
```
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ],
       [7. , 7. , 7. ],
       [7. , 7. , 7. ]])
```

Stack arrays e and f horizontally (column-wise):

In [131…
```
np.hstack((b,h))
```

Out[131…
```
array([[1.5, 2. , 3. , 7. , 7. , 7. ],
       [4. , 5. , 6. , 7. , 7. , 7. ]])
```

Create stacked column-wise arrays:

In [132…
```
a = np.array([1, 2, 3])
d = np.array([10, 15, 20])
```

```
col_stack = np.column_stack((a, d))
print(col_stack)
# Output:
# [[ 1 10]
#  [ 2 15]
#  [ 3 20]]
```

```
[[ 1 10]
 [ 2 15]
 [ 3 20]]
```

Create stacked column-wise arrays with `np.c_`:

In [133…
```
np.c_[a,d]
```

Out[133…
```
array([[ 1, 10],
       [ 2, 15],
       [ 3, 20]])
```

Spliting Arrays

`np.hsplit(a,3)`

Split array a horizontally at index 3:

In [139…
```
a = np.array([7, 2, 9])

print(np.hsplit(a,3))
```

```
[array([7]), array([2]), array([9])]
```

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

Split array b vertically at index 2:

In [141…
```
b = np.array([[1.5, 2, 3],
              [4, 5, 6]])

vsplit = np.vsplit(b, 2)
print(vsplit)
```

```
[array([[1.5, 2. , 3. ]]), array([[4., 5., 6.]])]
```

Generating Random Numbers Using NumPy

In [143…
```python
# Step 1: Import NumPy
import numpy as np
```

1. Generate Random Floats Between 0 and 1

`np.random.rand()` generates a random float number between 0 and 1 (exclusive)

In [144…
```python
# Generate one random number between 0 and 1
random_number = np.random.rand()
print(random_number)

# 1D array of random numbers between 0 and 1
array_of_randoms_1d = np.random.rand(5)
print(array_of_randoms_1d)

# Generate a 3x2 matrix of random numbers between 0 and 1
array_of_randoms = np.random.rand(3, 2)
print(array_of_randoms)
```

```
0.6915203376648857
[0.76439538 0.07876377 0.1809825  0.08467522 0.27675735]
[[0.24163974 0.2881456 ]
 [0.31455344 0.54847303]
 [0.54185151 0.43182983]]
```

2. Generate Random Integers

`np.random.randint(low, high)` gives a random integer between low (inclusive) and high (exclusive)

In [145…
```python
# This will generate a random integer in the range [1, 10)
random_integer = np.random.randint(1, 10)
```

```
print(random_integer)

# This will generate a 3x2 matrix of random integers in the range [1, 10)
random_integers = np.random.randint(1, 10, size=(3, 2))
print(random_integers)
```

```
6
[[7 5]
 [8 9]
 [9 6]]
```

3. Generate Random Numbers from a Standard Normal Distribution

`np.random.randn()` generates numbers from a standard normal distribution (mean = 0, std = 1)

In [146…

```
normal_numbers = np.random.randn(5)
print(normal_numbers)

# This will generate a 2x3 matrix of random numbers from a normal distribution with mean 0 and standard deviation 1
normal_matrix = np.random.randn(2, 3)
print(normal_matrix)
```

```
[ 0.88762728 -0.73289487 -1.18802434 -0.11661608  0.27963432]
[[-0.94354398 -0.57606686 -0.72224853]
 [-0.24351469  0.92731846  1.43310769]]
```

4. Generate Random Numbers from a Custom Normal Distribution

`loc` is the mean, `scale` is the standard deviation, and `size` is how many numbers to generate.

In [147…

```
# This will generate a 4 random numbers from a normal distribution with mean 50 and standard deviation 5

normal_custom = np.random.normal(loc=50, scale=5, size=4)
print(normal_custom)
```

```
[46.86128987 47.13706271 56.39019345 47.48276563]
```

5. Generate Random Numbers from a Uniform Distribution

This gives 3 float numbers between 5 and 10.

In [148…
```
uniform_numbers = np.random.uniform(low=5, high=10, size=3)
print(uniform_numbers)
```

```
[6.39841573 9.62851625 6.04059883]
```

6. Randomly Pick Elements from a List

In [150…
```
# Randomly selects 3 elements from the list.
choices = np.random.choice([10, 20, 20, 20, 30, 40, 40, 40, 50], size=3)
print(choices)

# You can also set replace=False to avoid duplicates
unique_choices = np.random.choice([10, 20, 30, 40, 50], size=3, replace=False)
print(unique_choices)
```

```
[10 20 30]
[20 10 50]
```

7. Shuffle an Array

This shuffles the array in place (changes the original array).

In [151…
```
arr = np.array([1, 2, 3, 4, 5])
np.random.shuffle(arr)
print(arr)
```

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

Set a Seed for Reproducibility

This fixes the sequence of random numbers so they are repeatable.

In [152…

```
np.random.seed(42)
print(np.random.rand(3))
```

```
[0.37454012 0.95071431 0.73199394]
```

In [ ]:

In [ ]: