NUMPY

-MUKESH KUMAR

Numpy Intro

- NumPy, short for Numerical Python, is a fundamental library for scientific computing in Python,
- It provides support for arrays, mathematical functions, linear algebra, random number generation, and more.
- Understanding NumPy is essential for data scientists working with numerical data.

Historical Development

- Origins
  - Numeric: The precursor to NumPy was Numeric, created by Jim Hugunin in 1995. This library aimed to provide array computing capabilities in Python and was influenced by languages such as APL, MATLAB, and FORTRAN.
  - Numarray: In the early 2000s, Numarray was developed as a more flexible alternative to Numeric, offering better performance for large arrays but slower performance for smaller ones. Both libraries coexisted for a time, catering to different needs within the Python community.

Creation of NumPy

- In 2005, Travis Oliphant, a key figure in the development of NumPy, sought to unify the community around a single array package.

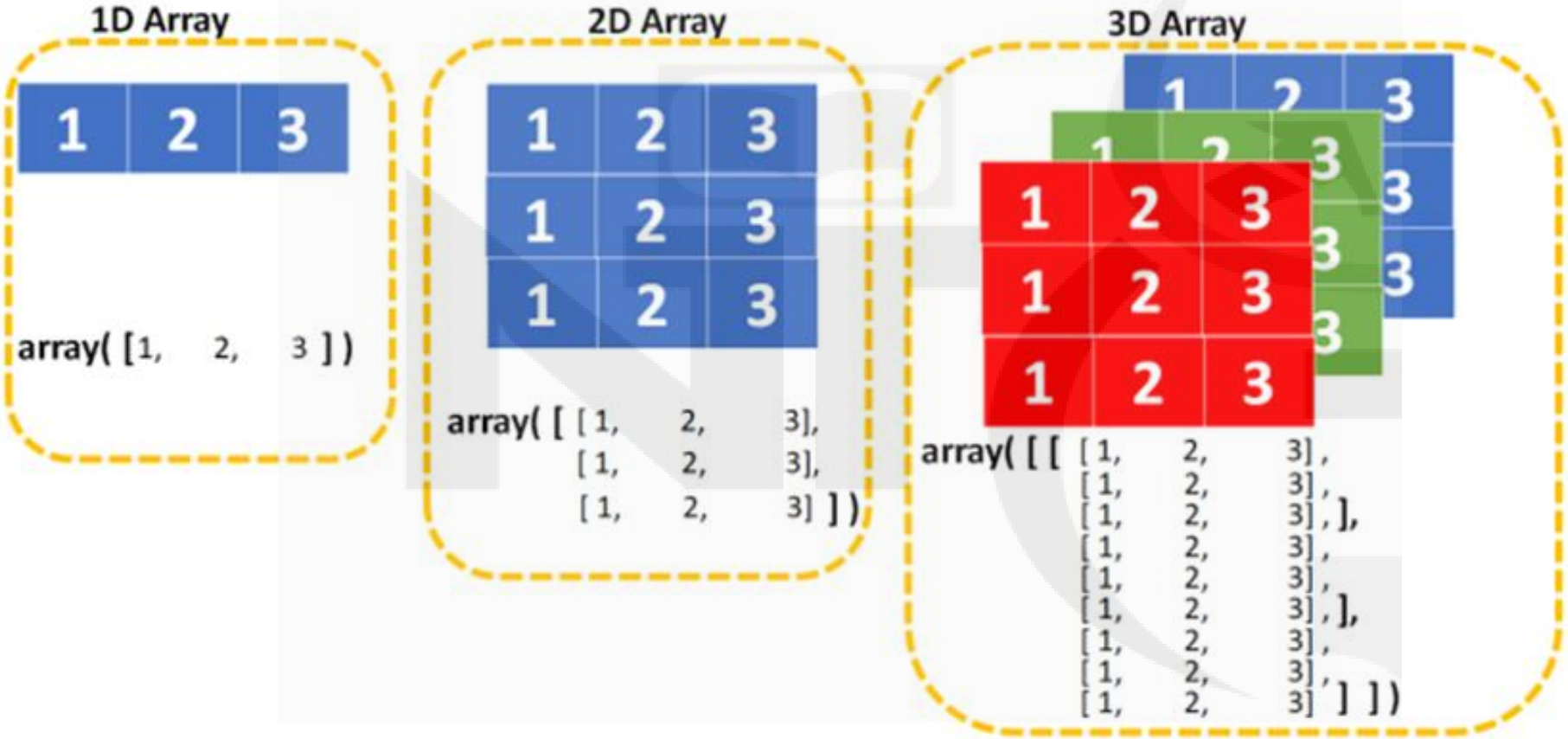
- He combined features from both Numeric and Numarray, leading to the creation of NumPy.

- The first official release, NumPy 1.0, occurred in 2006. This initiative was part of a broader effort to enhance Python's capabilities for scientific computing, and NumPy quickly became integral to the SciPy ecosystem

Numpy In Data Science

- NumPy is a foundational library in the Python ecosystem, and many other libraries are built on top of it, particularly in the data science, machine learning, and scientific computing domains. Here are some key libraries built on top of NumPy:
  - Pandas
  - Scipy
  - Matplotlib
  - Scikit Learn
  - Tensor Flow
  - Pytorch
  - Seaborn
  - OpenCV and so on..

NdArray


1D Array
1 2 3
array([1, 2, 3])
2D Array
1 2 3
1 2 3
1 2 3
array([[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])
3D Array
1 2 3
1 2 3
1 2 3
array([[[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3] ,],
[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3] ,],
[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3]]])

Numpy Creation

- 1. From a Python list or tuple
- 2. Using `arange`
- 3. Using `linspace`
- 4. Using `ones`, `zeros`, and `empty`
- 5. Using `full`
- 6. Using `eye` and `identity`
- 7. Using `random`
- 8. Using `astype` to create arrays of specific data types
- 9. Using `reshape` to create a new array
- 10. Using `tile` to repeat arrays
- 11. Using `fromfunction`
- 12. Using `fromiter`
- 13. Using `diag`

From a Python list

- import numpy as np
- my_list = [1,2,3,4]
- nparray = np.array(my_list)

- Or

- nparray = np.array([1,2,3,4])

From a Python list

- # Creating a 2D array
  - arr = np.array([[1, 2, 3], [4, 5, 6]])
  - This creates an array of 2 rows and 3 cols

From a Python list

- #create a 3D array
  - arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
  - This will create a 2,2,3

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

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

From a Python list using dtype

- #create a 3D array
  - arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]], dtype=np.float64)
  - This will create a 2,2,3

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

       [[ 7.,  8.,  9.],
        [10., 11., 12.]]])
```

From a Python Tuple

- # Creating a 1D array from a tuple
  - import numpy as np
  - my_tp = (1,2,3,4)
  - nparray = np.array(my_tp)

  - Or

  - nparray = np.array((1,2,3,4))

From a Python Tuple

- # Creating a 2D array from a tuple of tuples

- arr_2d = np.array(((1, 2, 3), (4, 5, 6)))
- This creates an array of 2 rows and 3 cols

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

From a Python Tuple

- # Creating a 3D array from a tuple of tuples of tuples
  - arr_3d = np.array((((1, 2, 3), (4, 5, 6)), ((7, 8, 9), (10, 11, 12))))
  - (2,2,3)

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

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

From a Python Tuple using dtype

- # Creating a 3D array from a tuple of tuples of tuples
  - arr_3d = np.array((((1, 2, 3), (4, 5, 6)), ((7, 8, 9), (10, 11, 12))),dtype=np.float64)
  - (2,2,3)

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

       [[11., 22., 33.],
        [44., 55., 66.]]])
```

NUMPY Properties

- **Shape**
- **Size**
- **ndim**
- **dtype**
- **itemsize**
- **Nbytes**
- **T**
- **Flat**

(3,)

NUMPY Properties

- **Shape**: The dimensions of the array, represented as a tuple. For example, (3, 4) indicates an array with 3 rows and 4 columns.

```
arr1 = np.array([1,2,3])
```

```
arr1.shape
```

```
(3,)
```

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

```
arr2.shape
```

```
(2, 3)
```

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.shape
```

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

- **Size**: The total number of elements in the array. It is the product of the array's shape dimensions.

```
arr1 = np.array([1,2,3])
```

```
arr1.size
```

```
3
```

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

```
arr2.size
```

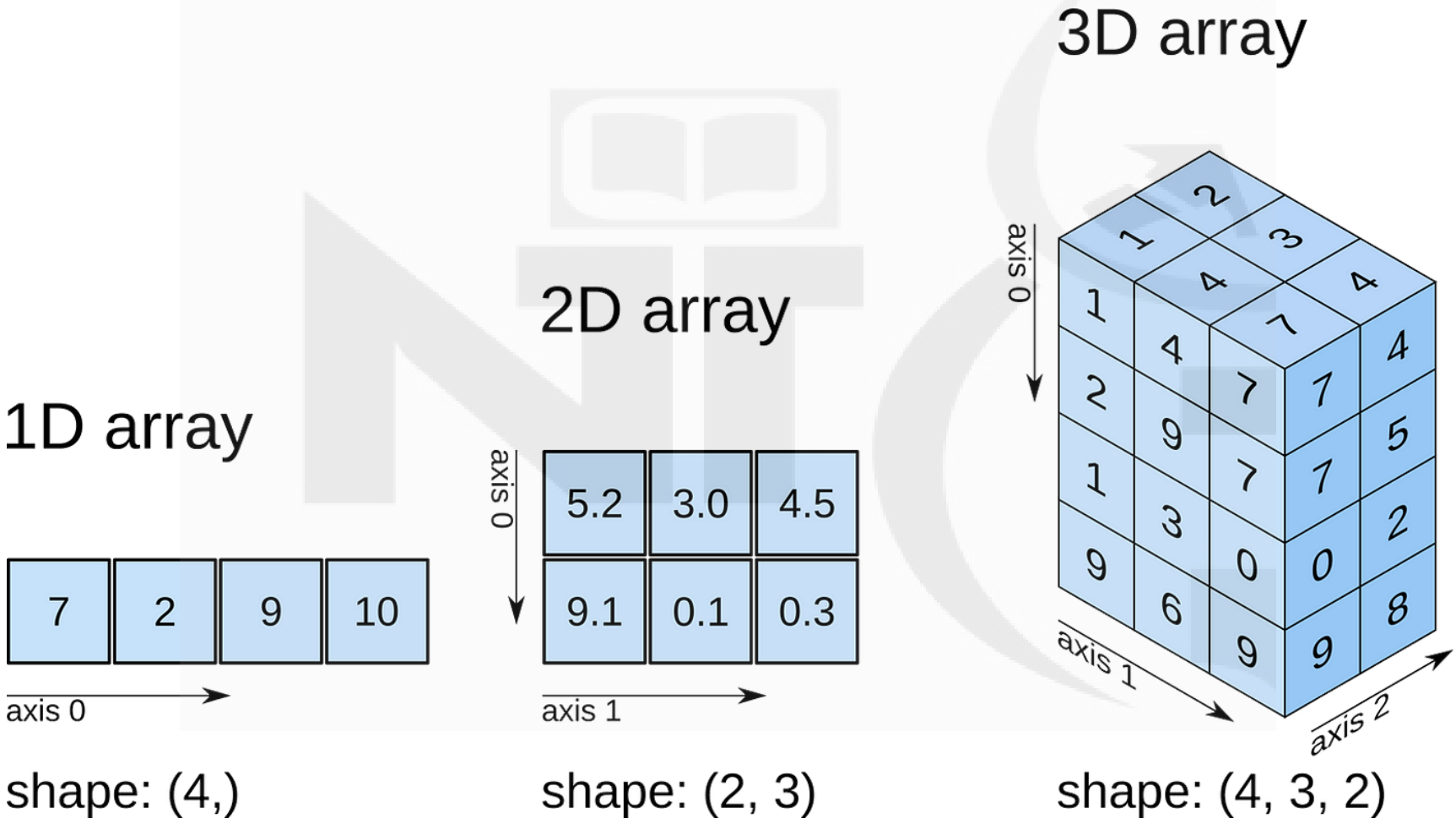
```
6
```

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.size
```

```
12
```

Axes in Numpy



- **ndim**: The number of dimensions (axes) of the array. For example, a 2D array has ndim equal to 2.

```
arr1 = np.array([1,2,3])
```

```
arr1.ndim
```

```
1
```

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

```
arr2.ndim
```

```
2
```

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.ndim
```

```
3
```

- **dtype**: The data type of the elements in the array, such as int, float, or str.

```
arr1 = np.array(['1','2','3'])
```

```
arr1.dtype
```

```
dtype('<U1')
```

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.dtype
```

```
dtype('int64')
```

- **itemsize**: The size (in bytes) of each element in the array.

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.itemsize
```

```
8
```

- **nbytes**: The total number of bytes used by the data portion of the array. It is calculated as itemsize * size.

```
arr3 = np.array([[[1,2,3],[4,5,6]],[[11,22,33],[44,55,66]]])
```

```
arr3.nbytes
```

```
96
```

From Arange

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

- **start***integer or real, optional*Start of interval. The interval includes this value. The default start value is 0.

- **stop***integer or real*End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

- **step***integer or real, optional*Spacing between values. For any output *out*, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1.

- **dtype***dtype, optional*The type of the output array. If dtype is not given, infer the data type from the other input arguments.

From Arange

- # Create an array with values from 0 to 9

  arange_array = np.arange(10)

- # Create an array with values from 0 to 9 with a step of 2
  - arange_array_step = np.arange(0, 10, 2)

From Arange

- # Create an array with values from 0 to 9
  - arange_array = np.arange(10,dtype=np.float64)
- # Create an array with values from 0 to 9 with a step of 2
  - arange_array_step = np.arange(0, 10, 2, dtype=np.float64)

Using Linspace

numpy.linspace

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False,
dtype=None, axis=0, *, device=None) #                                [source]
```

Return evenly spaced numbers over a specified interval.

- By default endpoint=True i.e stop is included

Using Linspace

- By default 50 numbers are generated

```
array_linspace = np.linspace(0, 10)
print("Linspace array:", array_linspace)
print(len(array_linspace))
```

```
Linspace array: [ 0.         0.20408163  0.40816327  0.6122449   0.81632653  1.02040816
  1.2244898   1.42857143  1.63265306  1.83673469  2.04081633  2.24489796
  2.44897959  2.65306122  2.85714286  3.06122449  3.26530612  3.46938776
  3.67346939  3.87755102  4.08163265  4.28571429  4.48979592  4.69387755
  4.89795918  5.10204082  5.30612245  5.51020408  5.71428571  5.91836735
  6.12244898  6.32653061  6.53061224  6.73469388  6.93877551  7.14285714
  7.34693878  7.55102041  7.75510204  7.95918367  8.16326531  8.36734694
  8.57142857  8.7755102   8.97959184  9.18367347  9.3877551   9.59183673
  9.79591837 10.        ]
50
```

Using Linspace

- Num =10

```
array_linspace = np.linspace(0, 10,10)
print("Linspace array:", array_linspace)
print(len(array_linspace))
```

```
Linspace array: [ 0.         1.11111111  2.22222222  3.33333333  4.44444444  5.55555556
  6.66666667  7.77777778  8.88888889 10.        ]
10
```

Using Linspace

- Excluding stop and returning step size:

```
array_linspace = np.linspace(0, 10,11,endpoint=False,retstep=True)
print("Linspace array:", array_linspace)
print(len(array_linspace))
```

```
Linspace array: (array([0.        , 0.90909091, 1.81818182, 2.72727273, 3.63636364,
       4.54545455, 5.45454545, 6.36363636, 7.27272727, 8.18181818,
       9.09090909]), 0.9090909090909091)
2
```

Using ones

- 1D:

```
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```

```
>>> np.ones((5,), dtype=int)
array([1, 1, 1, 1, 1])
```

Using Ones

- 2D:

```
>>> np.ones((2, 1))
array([[1.],
       [1.]])
```

```
>>> s = (2,2)
>>> np.ones(s)
array([[1.,  1.],
       [1.,  1.]])
```

Using Zeroes

- 1D

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])
```

Using Zeroes

- 2D:

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

Using Empty

- 1D:

```
[32] x = np.empty([2, ])
     print (x)

     x.shape
```

```
[1. 1.]
(2,)
```

- 2D:

```
np.empty([2, 2])
```

```
array([[ 2.5,  5. ],
       [ 7.5, 10. ]])
```

Using full

- **Description:** Creates an array filled with a specified value.

```
>>> np.full((2, 2), np.inf)
array([[inf, inf],
       [inf, inf]])
>>> np.full((2, 2), 10)
array([[10, 10],
       [10, 10]])
```

```
>>> np.full((2, 2), [1, 2])
array([[1, 2],
       [1, 2]])
```

Using eye

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

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

- N= rows, M = cols
- By default M = N, if M is not provided

Using eye

- When M is not provided:

```
np.eye(3, dtype=int)

array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

- When M is provided:

```
np.eye(3,2 ,dtype=int)

array([[1, 0],
       [0, 1],
       [0, 0]])
```

Using eye

- K= positive moves the diagonal up and negative moves the diagonal down, 0 is the main diagonal

```
np.eye(5 ,k=-1, dtype=int)

array([[0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0]])
```

```
np.eye(5 ,k=1, dtype=int)

array([[0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0]])
```

Using Identity

- The identity array is a square array with ones on the main diagonal.

```
array_identity = np.identity(4)
print("Identity array:\n", array_identity)
```

```
Identity array:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Using random

- Return random floats in the half-open interval [0.0, 1.0).

```
array_random = np.random.random((2, 3))
print("Random array:\n", array_random)

Random array:
 [[0.65788935 0.1441667  0.75790809]
 [0.83083293 0.19494352 0.86885061]]
```

Random Array of Floats (Uniform Distribution)

- Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

numpy.random.rand

- 1D

```
random_array_1d = np.random.rand(5)
random_array_1d
```

```
array([0.71608838, 0.38094562, 0.40788327, 0.48256015, 0.65663368])
```

- 2D

```
random_array_2d = np.random.rand(3, 4)
random_array_2d
```

```
array([[0.83193139, 0.01067393, 0.2597149 , 0.56144636],
       [0.61320039, 0.03637135, 0.38359894, 0.26431046],
       [0.82655725, 0.67507209, 0.42417355, 0.04741543]])
```

Random Array of Floats (Normal Distribution)

- np.random.randn()
- Generates an array of random numbers from a standard normal distribution (mean=0, std=1).

- 1D

```
# Generate a 1D array of 5 random floats from a normal distribution
random_normal_array_1d = np.random.randn(3)
random_normal_array_1d
```

```
array([ 0.43432346, -0.99477822, -0.01952865])
```

- 2D

```
# Generate a 2D array of shape (3, 4) from a normal distribution
random_normal_array_2d = np.random.randn(3, 4)
random_normal_array_2d
```

```
array([[-0.84997195, -0.59681364, -0.27956121,  1.85392287],
       [ 1.59966557, -0.529944  , -0.76757237, -1.27842979],
       [ 0.49814877, -0.34420772,  0.73140108,  0.13093803]])
```

Random Integers

- np.random.randint()
- Generates an array of random integers between a specified range

- 1D

```
# Generate a 1D array of 5 random integers between 10 and 50
random_integers_1d = np.random.randint(10, 50, size=5)
random_integers_1d
```

```
array([14, 32, 31, 49, 42])
```

- 2D

```
# Generate a 2D array of shape (3, 4) with integers between 0 and 100
random_integers_2d = np.random.randint(0, 100, size=(3, 4))
random_integers_2d
```

```
array([[44, 84,  5, 57],
       [58, 96, 57, 42],
       [69, 28, 21, 48]])
```

Using astype to create arrays of specific data types

- Changes the data type of an array.

```
# Creating an array and converting it to an integer type
array_float = np.array([1.5, 2.8, 3.3])
array_int = array_float.astype(int)
print("Original array:", array_float)
print("Integer array:", array_int)
```

```
Original array: [1.5 2.8 3.3]
Integer array: [1 2 3]
```

Using reshape to create a new array

- Reshapes an existing array to a different shape

```
# Creating a 1D array and reshaping it to a 2D array
array_1d = np.arange(6)
array_reshaped = array_1d.reshape(2, 3)
print("Original array:", array_1d)
print("Reshaped array:\n", array_reshaped)
```

```
Original array: [0 1 2 3 4 5]
Reshaped array:
 [[0 1 2]
 [3 4 5]]
```

Using tile to repeat arrays

- Repeats an array a specified number of times.

```
# Creating a small array and repeating it
array_tile = np.tile(np.array([1, 2, 3]), (2, 3))
print("Tiled array:\n", array_tile)
```

```
Tiled array:
 [[1 2 3 1 2 3 1 2 3]
 [1 2 3 1 2 3 1 2 3]]
```

Using fromfunction

- Construct an array by executing a function over each coordinate.

```
# Creating a 3x3 array using fromfunction
def func(i, j):
    return i + j

array_fromfunc = np.fromfunction(func, (3, 3))
print("Fromfunction array:\n", array_fromfunc)
```

```
Fromfunction array:
 [[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]]
```

Using fromiter

- Creates an array from an iterable.

```
# Creating an array from an iterable (range in this case)
iterable = range(5)
array_fromiter = np.fromiter(iter(iterable), dtype=int)
print("Fromiter array:", array_fromiter)
```

```
Fromiter array: [0 1 2 3 4]
```

Using diag

numpy.diag(*v*, *k=0*)

Extract a diagonal or construct a diagonal array.

```
# Creating a diagonal array
array_diag = np.diag([1, 2, 3, 4])
print("Diagonal array:\n", array_diag)
```

```
Diagonal array:
 [[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

```
# Extracting the diagonal from an existing array
array_extract_diag = np.diag(array_diag)
print("Extracted diagonal:", array_extract_diag)
```

```
Extracted diagonal: [1 2 3 4]
```