UNIT II

Mathematical Computing with Python (NumPy):Working with NumPy Arrays, Data Types, Array Creation, Indexing and Slicing, Numerical Operations on Arrays, Array Functions, Data Processing using Arrays**,** Loading and Saving Data, Saving an Array, Loading an Array, Numpy Random Numbers

Data Manipulation with Pandas: Data Wrangling, Data Exploration, Cleaning Data, Filtering, Merging Data, Reshaping Data, Data Aggregation**,** Reading and Writing Files, Loading and Saving Data with Pandas

NUMPY

NumPy is a Python library.

NumPy is used for working with arrays.

NumPy is short for "Numerical Python"

**NumPy(Numerical Python)** is a fundamental library for Python **numerical computing**. It provides efficient multi-dimensional array objects and various mathematical functions for handling large datasets making it a critical tool for professionals in fields that require heavy computation.

**Creating NumPy Arrays**

* **Using ndarray :**The array object is called [ndarray.](https://www.geeksforgeeks.org/numpy-ndarray/" \t "_blank) NumPy arrays are created using the array() function.

import numpy as np

# Creating a 1D array

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

# Creating a 2D array

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

# Creating a 3D array

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

print(x)

print(y)

print(z)

[1 2 3]

[[1 2]

[3 4]]

[[[1 2]

[3 4]]

[[5 6]

[7 8]]]

**NumPy Array Indexing**

Knowing the basics of [NumPy array indexing](https://www.geeksforgeeks.org/numpy-indexing/) is important for analyzing and manipulating the array object.

* **Basic Indexing:**Basic indexing in NumPy allows you to access elements of an array using indices.

**Example:**

1

import numpy as np

2

​

3

# Create a 1D array

4

arr1d = np.array([10, 20, 30, 40, 50])

5

​

6

# Single element access

7

print("Single element access:", arr1d[2])

8

​

9

# Negative indexing

10

print("Negative indexing:", arr1d[-1])

11

​

12

# Create a 2D array

13

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

14

​

15

# Multidimensional array access

16

print("Multidimensional array access:", arr2d[1, 0])

**Output**

Single element access: 30

Negative indexing: 50

Multidimensional array access: 4

* **Slicing:** Just like lists in Python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.

**Example:**

1

import numpy as np

2

​

3

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

4

#elements from index 1 to 3

5

print("Range of Elements:",arr[1:4])

6

​

7

#all rows, second column

8

print("Multidimensional Slicing:", arr[:, 1])

**Output**

Range of Elements: [[4 5 6]]

Multidimensional Slicing: [2 5]

* **Advanced Indexing:**Advanced Indexing in NumPy provides more powerful and flexible ways to access and manipulate array elements.

**Example:**

1

import numpy as np

2

arr = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

3

​

4

# Integer array indexing

5

indices = np.array([1, 3, 5])

6

print ("Integer array indexing:", arr[indices])

7

​

8

# boolean array indexing

9

cond = arr > 0

10

print ("\nElements greater than 0:\n", arr[cond])

**Output**

Elements at indices (0, 3), (1, 2), (2, 1),(3, 0):

[4. 6. 0. 3.]

Elements greater than 0:

[2. 4. 4. 6. 2.6 7. 8. 3. 4. 2. ]

**NumPy Basic Operations**

Element-wise operations in NumPy allow you to perform mathematical operations on each element of an array individually, without the need for explicit loops.

* **Element-wise Operations:** We can perform arithmetic operations like addition, subtraction, multiplication, and division directly on NumPy arrays.

**Example:**

1

import numpy as np

2

​

3

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

4

y = np.array([4, 5, 6])

5

​

6

# Addition

7

add = x + y

8

print("Addition:",add)

9

​

10

# Subtraction

11

subtract = x - y

12

print("substration:",subtract)

13

​

14

# Multiplication

15

multiply = x \* y

16

print("multiplication:",multiply)

17

​

18

# Division

19

divide = x / y

20

print("division:", divide)

**Output**

Addition: [5 7 9]

substration: [-3 -3 -3]

multiplication: [ 4 10 18]

division: [0.25 0.4 0.5 ]

* **Unary Operation:**These operations are applied to each individual element in the array, without the need for multiple arrays (as in binary operations).

**Example:**

1

import numpy as np

2

​

3

# Example array with both positive and negative values

4

arr = np.array([-3, -1, 0, 1, 3])

5

​

6

# Applying a unary operation: absolute value

7

result = np.absolute(arr)

8

print("Absolute value:", result)

**Output**

Absolute value: [3 1 0 1 3]

* **Binary Operators:**[Numpy Binary Operations](https://www.geeksforgeeks.org/numpy-binary-operations/" \t "_blank) apply to the array elementwise and a new array is created. We can use all basic arithmetic operators like +, -, /,  etc. In the case of +=, -=, = operators, the existing array is modified.

**Example:**

1

import numpy as np

2

​

3

# Two example arrays

4

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

5

arr2 = np.array([4, 5, 6])

6

​

7

# Applying a binary operation: addition

8

result = np.add(arr1, arr2)

9

​

10

print("Array 1:", arr1)

11

print("Array 2:", arr2)

12

print("Addition Result:", result)

**Output**

Array 1: [1 2 3]

Array 2: [4 5 6]

Addition Result: [5 7 9]

**NumPy ufuncs**

NumPy provides familiar mathematical functions such as **sin, cos, exp, etc**. These functions also operate elementwise on an array, producing an array as output.

**Example:**

**import** **numpy** **as** **np**

*# create an array of sine values*

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

print ("Sine values of array elements:", np.sin(a))

*# exponential values*

a = np.array([0, 1, 2, 3])

print ("Exponent of array elements:", np.exp(a))

*# square root of array values*

print ("Square root of array elements:", np.sqrt(a))

**Output:**

Sine values of array elements: [ 0.00000000e+00 1.00000000e+00 1.22464680e-16]

Exponent of array elements: [ 1. 2.71828183 7.3890561 20.08553692]

Square root of array elements: [ 0. 1. 1.41421356 1.73205081]

**NumPy Sorting Arrays**

We can use a simple **[np.sort()](https://www.geeksforgeeks.org/numpy-sort-in-python/)** method for sorting Python NumPy arrays.

**Example:**

1

import numpy as np

2

​

3

# set alias names for dtypes

4

dtypes = [('name', 'S10'), ('grad\_year', int), ('cgpa', float)]

5

​

6

# Values to be put in array

7

values = [('Hrithik', 2009, 8.5), ('Ajay', 2008, 8.7),

8

('Pankaj', 2008, 7.9), ('Aakash', 2009, 9.0)]

9

10

# Creating array

11

arr = np.array(values, dtype = dtypes)

12

print ("\nArray sorted by names:\n",

13

np.sort(arr, order = 'name'))

14

15

print ("Array sorted by graduation year and then cgpa:\n",

16

np.sort(arr, order = ['grad\_year', 'cgpa']))

**Output**

Array sorted by names:

[(b'Aakash', 2009, 9. ) (b'Ajay', 2008, 8.7) (b'Hrithik', 2009, 8.5)

(b'Pankaj', 2008, 7.9)]

Array sorted by graduation year and then cgpa:

[(b'Pankaj', 2008, 7.9) (b'Ajay',...

**Key Features of NumPy**

NumPy has various features that make it popular over lists.

* **N-Dimensional Arrays**: NumPy’s core feature is ndarray, a powerful N-dimensional array object that supports homogeneous data types.
* **Arrays with High Performance**: Arrays are stored in contiguous memory locations, enabling faster computations than Python lists(Please see [Numpy Array vs Python List](https://www.geeksforgeeks.org/python-lists-vs-numpy-arrays/" \t "_blank) for details).
* [Broadcasting](https://www.geeksforgeeks.org/numpy-array-broadcasting/): This allows element-wise computations between arrays of different shapes. It simplifies operations on arrays of **various shapes** by automatically aligning their dimensions without creating new data.
* [Vectorization](https://www.geeksforgeeks.org/vectorization-in-python/): Eliminates the need for explicit Python loops by applying operations directly on entire arrays.
* **Linear algebra**: NumPy contains routines for linear algebra operations, such as matrix multiplication, decompositions, and determinants.

Basic

[Introduction](https://www.w3schools.com/python/numpy/numpy_intro.asp)

[Getting Started](https://www.w3schools.com/python/numpy/numpy_getting_started.asp)

[Creating Arrays](https://www.w3schools.com/python/numpy/numpy_creating_arrays.asp)

[Array Indexing](https://www.w3schools.com/python/numpy/numpy_array_indexing.asp)

[Array Slicing](https://www.w3schools.com/python/numpy/numpy_array_slicing.asp)

[Data Types](https://www.w3schools.com/python/numpy/numpy_data_types.asp)

[Copy vs View](https://www.w3schools.com/python/numpy/numpy_copy_vs_view.asp)

[Array Shape](https://www.w3schools.com/python/numpy/numpy_array_shape.asp)

[Array Reshape](https://www.w3schools.com/python/numpy/numpy_array_reshape.asp)

[Array Iterating](https://www.w3schools.com/python/numpy/numpy_array_iterating.asp)

[Array Join](https://www.w3schools.com/python/numpy/numpy_array_join.asp)

[Array Split](https://www.w3schools.com/python/numpy/numpy_array_split.asp)

[Array Search](https://www.w3schools.com/python/numpy/numpy_array_search.asp)

[Array Sort](https://www.w3schools.com/python/numpy/numpy_array_sort.asp)

[Array Filter](https://www.w3schools.com/python/numpy/numpy_array_filter.asp)

Random

[Random Intro](https://www.w3schools.com/python/numpy/numpy_random.asp)

[Data Distribution](https://www.w3schools.com/python/numpy/numpy_random_distribution.asp)

[Random Permutation](https://www.w3schools.com/python/numpy/numpy_random_permutation.asp)

[Seaborn Module](https://www.w3schools.com/python/numpy/numpy_random_seaborn.asp)

[Normal Dist.](https://www.w3schools.com/python/numpy/numpy_random_normal.asp)

[Binomial Dist.](https://www.w3schools.com/python/numpy/numpy_random_binomial.asp)

[Poisson Dist.](https://www.w3schools.com/python/numpy/numpy_random_poisson.asp)

[Uniform Dist.](https://www.w3schools.com/python/numpy/numpy_random_uniform.asp)

[Logistic Dist.](https://www.w3schools.com/python/numpy/numpy_random_logistic.asp)

[Multinomial Dist.](https://www.w3schools.com/python/numpy/numpy_random_multinomial.asp)

[Exponential Dis.](https://www.w3schools.com/python/numpy/numpy_random_exponential.asp)

[Chi Square Dist.](https://www.w3schools.com/python/numpy/numpy_random_chisquare.asp)

[Rayleigh Dist.](https://www.w3schools.com/python/numpy/numpy_random_rayleigh.asp)

[Pareto Dist.](https://www.w3schools.com/python/numpy/numpy_random_pareto.asp)

[Zipf Dist.](https://www.w3schools.com/python/numpy/numpy_random_zipf.asp)

ufunc

[ufunc Intro](https://www.w3schools.com/python/numpy/numpy_ufunc.asp)

[Create Function](https://www.w3schools.com/python/numpy/numpy_ufunc_create_function.asp)

[Simple Arithmetic](https://www.w3schools.com/python/numpy/numpy_ufunc_simple_arithmetic.asp)

[Rounding Decimals](https://www.w3schools.com/python/numpy/numpy_ufunc_rounding_decimals.asp)

[Logs](https://www.w3schools.com/python/numpy/numpy_ufunc_logs.asp)

[Summations](https://www.w3schools.com/python/numpy/numpy_ufunc_summations.asp)

[Products](https://www.w3schools.com/python/numpy/numpy_ufunc_products.asp)

[Differences](https://www.w3schools.com/python/numpy/numpy_ufunc_differences.asp)

[Finding LCM](https://www.w3schools.com/python/numpy/numpy_ufunc_lcm.asp)

[Finding GCD](https://www.w3schools.com/python/numpy/numpy_ufunc_gcd.asp)

[Trigonometric](https://www.w3schools.com/python/numpy/numpy_ufunc_trigonometric.asp)

[Hyperbolic](https://www.w3schools.com/python/numpy/numpy_ufunc_hyperbolic.asp)

[Set Operations](https://www.w3schools.com/python/numpy/numpy_ufunc_set_operations.asp)

**NumPy**is a powerful [Python library](https://www.geeksforgeeks.org/libraries-in-python/) that can manage different types of data. Here we will explore the **Datatypes in NumPy** and How we can check and create datatypes of the [NumPy array](https://www.geeksforgeeks.org/basics-of-numpy-arrays/).

**DataTypes in NumPy**

A **data type** in [NumPy](https://www.geeksforgeeks.org/python-numpy/) is used to specify the type of data stored in a variable. Here is the list of characters available in NumPy to represent data types.

| **Character** | **Meaning** |
| --- | --- |
| b | Boolean |
| f | Float |
| m | Time Delta |
| O | Object |
| U | Unicode String |
| **i** | Integer |
| u | Unsigned Integer |
| c | Complex Float |
| M | DateTime |
| S | String |
| V | A fixed chunk of memory for other types (void) |

The list of various types of data types provided by NumPy are given below:

| **Data Type** | **Description** |
| --- | --- |
| bool\_ | Boolean |
| int\_ | Default integer type (int64 or int32) |
| intc | Identical to the integer in C (int32 or int64) |
| intp | Integer value used for indexing |
| int8 | 8-bit integer value (-128 to 127) |
| int16 | 16-bit integer value (-32768 to 32767) |
| int32 | 32-bit integer value (-2147483648 to 2147483647) |
| int64 | 64-bit integer value (-9223372036854775808 to 9223372036854775807) |
| uint8 | Unsigned 8-bit integer value (0 to 255) |
| uint16 | Unsigned 16-bit integer value (0 to 65535) |
| uint32 | Unsigned 32-bit integer value (0 to 4294967295) |
| uint64 | Unsigned 64-bit integer value (0 to 18446744073709551615) |
| float\_ | Float values |
| float16 | Half precision float values |
| float32 | Single-precision float values |
| float64 | Double-precision float values |
| complex\_ | Complex values |
| complex64 | Represent two 32-bit float complex values (real and imaginary) |
| complex128 | Represent two 64-bit float complex values (real and imaginary) |

**Checking the Data Type of NumPy Array**

We can check the datatype of [Numpy array](https://www.geeksforgeeks.org/basics-of-numpy-arrays/) by using dtype. Then it returns the data type all the elements in the array.

1

import numpy as np

2

​

3

# Create a NumPy array

4

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

5

​

6

# Check the data type of the array

7

data\_type = arr.dtype

8

print(data\_type)

**Output**

int64

**Create Arrays With a Defined Data Type**

We can create an array with a defined data type by specifying “dtype” attribute in **numpy.array()** method while initializing an array.

1

import numpy as np

2

​

3

arr1 = np.array([1, 2, 3, 4], dtype=np.float64)

4

​

5

# Creating a 3x3 int32 array of zeros

6

arr2 = np.zeros((3, 3), dtype=np.int32)

7

​

8

# Creating a 2x2 complex128 array of ones

9

arr3 = np.ones((2, 2), dtype=np.complex128)

10

​

11

# Creating a 1D bool array

12

arr4 = np.empty((4,), dtype=np.bool\_)

13

​

14

# Print the arrays and their data types

15

print(arr1.dtype)

16

print(arr2.dtype)

17

print(arr3.dtype)

18

print(arr4.dtype)

**Output**

float64

int32

complex128

bool

**Convert Data Type of NumPy Arrays**

We can convert data type of an arrays from one type to another using[astype() function](https://www.geeksforgeeks.org/numpy-maskedarray-astype-function-python/).

1

import numpy as np

2

​

3

arr1 = np.array([1.2, 2.5, 3.7])

4

​

5

# Converting to int32

6

arr2 = arr1.astype(np.int32)

7

​

8

# Print new array and its type

9

print(arr2)

10

print(arr2.dtype)

**Output**

[1 2 3]

int32

NumPy Operations

Python package NumPy stands for "Numerical Python." The logical computation package includes an efficient N-D array object and provides facilities to link C, C++, and other programming languages. Additionally, it helps with arbitrary number capacity, linear-based math, and other subjects. NumPy displays can be used as a powerful multi-dimensional container for general data. Python Array: Rows and columns make up the robust N-dimensional object known as a Numpy array. We may access its elements and initialize NumPy arrays using a nested Python list. On a structural plane, a Numpy array consists of a mixture of:

* The Data reference displays the memory location of the Numpy array's initial byte.
* The name Data type or the dtype argument describes the Python types of elements stored within the particular array.
* The shape denotes the number of rows and columns the array has.
* The strides specify how many bytes in the system's memory should be bypassed before moving on to the next array element.

Numpy Basics

Array of Ones

Creates a NumPy array according to the parameters given, with all elements being 1.

**Code**

1. # Python program to create a numpy array of ones
2. array = np.ones([2,3])
3. **print**(array)

**Output**

Backward Skip 10sPlay VideoForward Skip 10s

*[[1. 1. 1.]*

*[1. 1. 1.]]*

Array of Zeros

This function returns a NumPy array with all entries set to 0 having dimensions as specified.

**Code**

1. # Python program to create a numpy array of zeroes
2. array = np.zeros([2,3])
3. **print**(array)

**Output**

*[[0. 0. 0.]*

*[0. 0. 0.]]*

These functions are simple, and they can be used to create sample arrays which are often needed for various computational purposes.

Eye

Let's now examine how the eye method works. A 2-D array containing ones on its diagonal and zeros everywhere else is this function's output.

**Code**

Advertisement

1. # Python program to create a numpy array having ones on diagonal and zeroes elsewhere
2. array = np.eye(4)
3. **print**(array)# Python program to create a numpy array of ones
4. array = np.ones([2,3])
5. **print**(array)

**Output**

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

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

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

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

Similarly, the diag() method builds a 2D array with all other members set to zero, and the items supplied to function as arguments are set to the diagonal elements.

**Code**

1. **import** numpy as np
2. y = np.array([78,56,89])
3. np.diag(y)

**Output**

*array([[78, 0, 0],*

*[ 0, 56, 0],*

*[ 0, 0, 89]])*

Vstack() and Hstack()

We can use the vstack() function to vertically stack two arrays and hstack() to stack two or more arrays horizontally. Let's practice with several instances.

**Code**

1. # Python program to stack two arrays
3. array1 = np.array([[2,3,4,5],[4,3,5,3]])
4. array2 = np.array([[6,3,5,2],[8,2,5,8]])
6. # Vertical stacking
7. array = np.vstack((array1, array2))
8. **print**("Vertically stacked: \n", array)
10. # Horizontal stacking
11. array = np.hstack((array1, array2))
12. **print**("Horizontally stacked: \n", array)

**Output**

Advertisement

*Vertically stacked:*

*[[2 3 4 5]*

*[4 3 5 3]*

*[6 3 5 2]*

*[8 2 5 8]]*

*Horizontally stacked:*

*[[2 3 4 5 6 3 5 2]*

*[4 3 5 3 8 2 5 8]]*

Operations on Numpy Array

Arithmetic Operations

**Code**

1. # Python program to perform arithmetic operations on the Numpy arrays
3. **import** numpy as np
5. # Initializing our array
6. array1 = np.arange(9, dtype = np.float\_).reshape(3, 3)
7. **print**('First Array:')
8. **print**(array1)
10. **print**('Second array:')
11. array2 = np.arange(11,20, dtype = np.float\_).reshape(3, 3)
12. **print**(array2)
14. **print**('\nAdding two arrays:')
15. **print**(np.add(array1, array2))
17. **print**('\nSubtracting two arrays:')
18. **print**(np.subtract(array1, array2))
20. **print**('\nMultiplying two arrays:')
21. **print**(np.multiply(array1, array2))
23. **print**('\nDividing two arrays:')
24. **print**(np.divide(array1, array2))

**Output**

*First Array:*

*[[0. 1. 2.]*

*[3. 4. 5.]*

*[6. 7. 8.]]*

*Second array:*

*[[11. 12. 13.]*

*[14. 15. 16.]*

*[17. 18. 19.]]*

*Adding two arrays:*

*[[11. 13. 15.]*

*[17. 19. 21.]*

*[23. 25. 27.]]*

*Subtracting two arrays:*

*[[-11. -11. -11.]*

*[-11. -11. -11.]*

*[-11. -11. -11.]]*

*Multiplying two arrays:*

*[[ 0. 12. 26.]*

*[ 42. 60. 80.]*

*[102. 126. 152.]]*

*Dividing two arrays:*

*[[0. 0.08333333 0.15384615]*

*[0.21428571 0.26666667 0.3125 ]*

*[0.35294118 0.38888889 0.42105263]]*

numpy.reciprocal()

This method returns the argument's element-by-element inverse. When an element's absolute value is greater than 1, the outcome is always 0, and an overflow warning is shown for integer 0.

Advertisement

**Code**

1. # Python program to show how to perform the reciprocal operation on the NumPy arrays
3. # Importing the numpy library
4. **import** numpy as np
6. # Initializing our array
7. array = np.array([23, 14, 63.9, 23.5, 23.7, 13, 7])
9. **print**('The array is:')
10. **print**(array)
12. **print**('\nAfter applying the reciprocal function array is:')
13. **print**(np.reciprocal(array))
15. # Creating another array
16. array2 = np.array([24], dtype = int)
18. **print**()
20. **print**('The second array is: ')
21. **print**(array2)
23. **print**('After applying the reciprocal function the array is: ')
24. **print**(np.reciprocal(array2))

**Output**

*The array is:*

*[23. 14. 63.9 23.5 23.7 13. 7. ]*

*After applying the reciprocal function array is:*

*[0.04347826 0.07142857 0.01564945 0.04255319 0.04219409 0.07692308*

*0.14285714]*

*The second array is:*

*[24]*

*After applying the reciprocal function the array is:*

*[0]*

numpy.power()

This function treats the original array's elements as the base in the exponents' syntax, which then raises them to the power of the adjacent elements provided in the second array argument.

**Code**

1. # Python program to show how to take powers of the numpy array
3. # Importing the numpy library
4. **import** numpy as np
6. # Initializing the array
7. array = np.array([3, 5, 2])
9. **print**('The original array is:')
10. **print**(array)
12. **print**('Applying the numpy power function: ')
13. **print**(np.power(array, 2))
15. # Initializing another array
16. **print**('The second array is:')
17. array\_ = np.array([2, 4, 5])
18. **print**(array\_)
20. # Giving the power as a numpy array
21. **print**('Applying the numpy power function: ')
22. **print**(np.power(array, array\_))

**Output**

*The original array is:*

*[3 5 2]*

*Applying the numpy power function:*

*[ 9 25 4]*

*The second array is:*

*[2 4 5]*

*Applying the numpy power function:*

*[ 9 625 32]*

numpy.mod()

This function returns the remainder of the division of the corresponding elements in the input array. The function numpy.remainder() also produces the same result.

**Code**

1. # Python program to show how to find remainders on dividing an array by an array in numpy
3. # Importing the numpy library
4. **import** numpy as np
6. # Initializing the arrays
7. array = np.array([5, 10, 15])
8. array1 = np.array([2, 4, 5])
10. **print**('The original array:')
11. **print**(array)
13. **print**('The array for diving the original array:')
14. **print**(array1)
16. **print**('Applying the numpy mod function:')
17. **print**(np.mod(array, array1))
19. **print**('Applying the numpy remainder function:')
20. **print**(np.remainder(array, array1))

**Output**

*The original array:*

*[ 5 10 15]*

*The array for diving the original array:*

*[2 4 5]*

*Applying the numpy mod function:*

*[1 2 0]*

*Applying the numpy remainder function:*

*[1 2 0]*

The following functions are used to perform operations on an array with complex numbers.

**numpy.real() ?** This function will return the real part of the given complex argument.

**numpy.imag() ?** This function will return the imaginary part of the complex argument.

**numpy.conj() ?** This function will return the complex conjugate of the given complex argument. It is obtained by swapping the sign of the imaginary part.

**numpy.angle() ?** This function will return the angle of the given complex argument. The function has a parameter having the keyword- degree. If set to true, the function will return the angle in degrees; otherwise, the angle is returned in radians.

**Code**

1. # Python program to show how to perform operations on the complex elements of an array in numpy
3. # Importing the numpy library
4. **import** numpy as np
6. # Initializing the array
7. a = np.array([-6.6j, 0.9j, 14. , 1+9j])
9. **print**('Our complex array is:')
10. **print**(a)
12. **print**('Applying the numpy real function: ')
13. **print**(np.real(a))
15. **print**('Applying the numpy imag function: ')
16. **print**(np.imag(a))
18. **print**('Applying the numpy conj function: ')
19. **print**(np.conj(a))
21. **print**('Applying the numpy angle function: ')
22. **print**(np.angle(a))
24. **print**('Applying the numpy angle function again (result in degrees)')
25. **print**(np.angle(a, deg = True))

**Output**

*Our complex array is:*

*[-0.-6.6j 0.+0.9j 14.+0.j 1.+9.j ]*

*Applying the numpy real function:*

*[-0. 0. 14. 1.]*

*Applying the numpy imag function:*

*[-6.6 0.9 0. 9. ]*

*Applying the numpy conj function:*

*[-0.+6.6j 0.-0.9j 14.-0.j 1.-9.j ]*

*Applying the numpy angle function:*

*[-1.57079633 1.57079633 0. 1.46013911]*

*Applying the numpy angle function again (result in degrees)*

*[-90. 90. 0. 83.65980825]*

Using Numpy Arrays with Conditional Expressions

To identify the values that meet your criterion, utilize conditionals. The outcome of a conditional operating condition is also a numpy array because "array" is also a numpy array. Our conditional check produces an array containing boolean values as its result.

**Code**

1. # Python program to perform condition operations on the array
3. # Initializing the array
4. array = np.array([20, 30, 23, 12, 53, 54, 24])
6. # Values greater than 30
7. array\_30 = array >= 30
8. **print**(array\_30)
10. # Only storing the values that are greater than 30
11. **print**(array[array\_30])
13. # Direct method to do the above task
14. **print**(array[array >= 30])

**Output**

*[False True False False True True False]*

*[30 53 54]*

*[30 53 54]*

numpy.dot()

We will begin with the cases in which both arguments are scalars or one-dimensional arrays.

**Code**

1. # Python program to find the dot product of the numpy vectors
3. # Directly passing the values
4. **print**(np.dot(3, 4))
6. # Initializing the two arrays of a single value
7. a = np.array([3])
8. b = np.array([4])
9. **print**("Dimensions of a: ", a.ndim)
10. **print**(f"The dot product of {a} and {b} is: ", np.dot(a, b))
12. # Initializing the two arrays of two value
13. a = np.array([4, -3])
14. b = np.array([-6, 3])
15. **print**(f"The dot product of {a} and {b} is: ", np.dot(a, b))

**Output**

*12*

*Dimensions of a: 1*

*The dot product of [3] and [4] is: 12*

*The dot product of [ 4 -3] and [-6 3] is: -33*

Logical Operators

The logical operators "or" and "and" also apply to numpy arrays elementwise. For this, we can use the numpy logical\_or and logical\_and methods.

**Code**

1. # Python program to use logical operations on numpy arrays
3. # Importing the numpy library
4. **import** numpy as np
6. # Intializing the arrays
7. array = np.array([ [True, False], [True, False]])
8. array1 = np.array([ [False, True], [True, False]])
10. # Applying numpy "or" and "and" operations
11. **print**("After or operation: \n", np.logical\_or(array, array1))
12. **print**("After and operation: \n", np.logical\_and(array, array1))

**Output**

*After or operation:*

*[[ True True]*

*[ True False]]*

*After and operation:*

*[[False False]*

*[ True False]]*

Implementing Operations on Arrays Having Different Shapes

Up until now, we have dealt with two distinct instances using simple operations such as "+" or "\*":

* an operation performed on a scalar and a numpy array
* an operation used to combine two arrays having the same shape.

The next section will demonstrate that we can still apply operators even if an array has a unique shape. But only particular circumstances allow it to function.

Broadcasting

Arrays of different dimensions can be used for arithmetic operations thanks to Numpy's robust Broadcasting technique. This implies that we take a larger dimension array and a smaller dimension array, and we convert or extend the smaller dimension array to the larger dimension array multiple times to carry out an operation. To put this in another way, the smaller array can occasionally be "broadcasted" so that it takes on the same dimension as the larger array.

In our Python software, we can avoid loops with the help of broadcasting. In the C-based Numpy implementations, the iteration happens implicitly. Additionally, we refrain from making duplicates of our data.

A straightforward example is used to illustrate the broadcasting operational principle.

**Code**

1. # Python program to show how to apply operations on two numpy arrays of different shapes
3. # Importing the numpy library
4. **import** numpy as np
6. # Initializing two arrays of different lengths
7. arr1 = np.array([ [3, 2, 4], [1, 3, 2], [5, 3, 3] ])
8. arr2 = np.array([2, 2, 2])
10. # Original arrays
11. **print**("arr1: \n", arr1)
12. **print**("arr2: \n", arr2)
14. # Multiplieng the arrays
15. **print**("Multiplication using broadcasting: ")
16. **print**(arr1 \* arr2)
18. # Adding the arrays
19. **print**("Addition using broadcasting: ")
20. **print**(arr1 + arr2)

**Output**

*arr1:*

*[[3 2 4]*

*[1 3 2]*

*[5 3 3]]*

*arr2:*

*[2 2 2]*

*Multiplication using broadcasting:*

*[[ 6 4 8]*

*[ 2 6 4]*

*[10 6 6]]*

*Addition using broadcasting:*

*[[5 4 6]*

*[3 5 4]*

*[7 5 5]]*

Distance Matrix

A distance matrix in geometry is a matrix or a two-dimensional array that stores the distances between the members of a set, pairwise taken, in mathematics, computer science, and particularly in graph theory. If the set has n elements, this two-dimensional array's size is n x n.

A distance matrix connecting the coordinates, in our case, the alphabet, to a hypothetical point:

**Code**

1. # Python program to find the distance matrix using numpy functions
3. # Importing numpy
4. **import** numpy as np
5. # Initializing the arrays
6. coord = ["A", "B", "C", "D", "E",
7. "F", "G", "H", "I", "J",
8. "K", "L", "M", "N", "O",
9. "P", "Q", "R", "S", "T",
10. "U", "V", "W", "X", "Y", "Z"]
12. distance\_from\_coord = [12, 14, 13, 18, 18,
13. 16, 69, 12, 23, 24,
14. 10, 45, 35, 27, 15,
15. 33, 34, 37, 28, 22,
16. 47, 18, 19, 17, 21, 14]
18. distances =  np.array(distance\_from\_coord[:12])
19. **print**(distances)
20. **print**(np.abs(distances - distances[:, np.newaxis]))

**Output**

*[12 14 13 18 18 16 69 12 23 24 10 45]*

*[[ 0 2 1 6 6 4 57 0 11 12 2 33]*

*[ 2 0 1 4 4 2 55 2 9 10 4 31]*

*[ 1 1 0 5 5 3 56 1 10 11 3 32]*

*[ 6 4 5 0 0 2 51 6 5 6 8 27]*

*[ 6 4 5 0 0 2 51 6 5 6 8 27]*

*[ 4 2 3 2 2 0 53 4 7 8 6 29]*

*[57 55 56 51 51 53 0 57 46 45 59 24]*

*[ 0 2 1 6 6 4 57 0 11 12 2 33]*

*[11 9 10 5 5 7 46 11 0 1 13 22]*

*[12 10 11 6 6 8 45 12 1 0 14 21]*

*[ 2 4 3 8 8 6 59 2 13 14 0 35]*

*[33 31 32 27 27 29 24 33 22 21 35 0]]*

**Numerical Operations on Numpy Arrays**

[NumPy](https://www.geeksforgeeks.org/python-numpy/) is a Python package which means ‘Numerical Python’. It is the library for logical computing, which contains a powerful n-dimensional array object, gives tools to integrate C, C++ and so on. It is likewise helpful in linear based math, arbitrary number capacity and so on. NumPy exhibits can likewise be utilized as an effective multi-dimensional compartment for generic data. **NumPy Array:** Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize **NumPy arrays** from nested Python lists and access it elements. A Numpy array on a structural level is made up of a combination of:

* The **Data**pointer indicates the memory address of the first byte in the array.
* The **Data type** or **dtype** pointer describes the kind of elements that are contained within the array.
* The **shape** indicates the shape of the array.
* The **strides** are the number of bytes that should be skipped in memory to go to the next element.

**Operations on Numpy Array**

**Arithmetic Operations:**

* Python3

|  |
| --- |
| # Python code to perform arithmetic  # operations on NumPy array      **import** numpy as np      # Initializing the array  arr1 **=** np.arange(4, dtype **=** np.float\_).reshape(2, 2)    print('First array:')  **print**(arr1)    **print**('\nSecond array:')  arr2 **=** np.array([12, 12])  print(arr2)    print('\nAdding the two arrays:')  **print**(np.add(arr1, arr2))    **print**('\nSubtracting the two arrays:')  **print**(np.subtract(arr1, arr2))    print('\nMultiplying the two arrays:')  print(np.multiply(arr1, arr2))    print('\nDividing the two arrays:')  print(np.divide(arr1, arr2)) |

**Output:**

First array:

[[ 0. 1.]

[ 2. 3.]]

Second array:

[12 12]

Adding the two arrays:

[[ 12. 13.]

[ 14. 15.]]

Subtracting the two arrays:

[[-12. -11.]

[-10. -9.]]

Multiplying the two arrays:

[[ 0. 12.]

[ 24. 36.]]

Dividing the two arrays:

[[ 0. 0.08333333]

[ 0.16666667 0.25 ]]

**numpy.reciprocal()** This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0 and for integer 0, overflow warning is issued. **Example:**

* Python3

|  |
| --- |
| # Python code to perform reciprocal operation  # on NumPy array  **import** numpy as np  arr **=** np.array([25, 1.33, 1, 1, 100])    **print**('Our array is:')  **print**(arr)    **print**('\nAfter applying reciprocal function:')  **print**(np.reciprocal(arr))    arr2 **=** np.array([25], dtype **=** int)  print('\nThe second array is:')  print(arr2)    **print**('\nAfter applying reciprocal function:')  print(np.reciprocal(arr2)) |

**Output**

Our array is:

[ 25. 1.33 1. 1. 100. ]

After applying reciprocal function:

[ 0.04 0.7518797 1. 1. 0.01 ]

The second array is:

[25]

After applying reciprocal function:

[0]

**numpy.power()** This function treats elements in the first input array as the base and returns it raised to the power of the corresponding element in the second input array.

* Python3

|  |
| --- |
| # Python code to perform power operation  # on NumPy array      **import** numpy as np      arr **=** np.array([5, 10, 15])    **print**('First array is:')  **print**(arr)    **print**('\nApplying power function:')  print(np.power(arr, 2))    **print**('\nSecond array is:')  arr1 **=** np.array([1, 2, 3])  print(arr1)    print('\nApplying power function again:')  print(np.power(arr, arr1)) |

**Output:**

First array is:

[ 5 10 15]

Applying power function:

[ 25 100 225]

Second array is:

[1 2 3]

Applying power function again:

[ 5 100 3375]

**numpy.mod()** This function returns the remainder of division of the corresponding elements in the input array. The function numpy.remainder() also produces the same result.

* Python3

|  |
| --- |
| # Python code to perform mod function  # on NumPy array      **import** numpy as np      arr **=** np.array([5, 15, 20])  arr1 **=** np.array([2, 5, 9])    print('First array:')  print(arr)    print('\nSecond array:')  **print**(arr1)    print('\nApplying mod() function:')  print(np.mod(arr, arr1))    print('\nApplying remainder() function:')  print(np.remainder(arr, arr1)) |

**Output:**

First array:

[ 5 15 20]

Second array:

[2 5 9]

Applying mod() function:

[1 0 2]

Applying remainder() function:

[1 0 2]

ARRAY FUNCTIONS

In NumPy, "array functions" refer to a wide range of operations that can be performed on NumPy arrays, including mathematical calculations, statistical analysis, manipulation of array elements, and more, allowing for efficient processing of large datasets across multiple dimensions; key functions include creating arrays like np.zeros, np.ones, arithmetic operations like addition and multiplication, statistical functions like mean, std, and indexing/slicing operations to access specific elements within an array.

Key points about NumPy array functions:

* **N-dimensional arrays:**

NumPy's core data structure is the "ndarray" (n-dimensional array), enabling operations on multidimensional data like matrices and tensors.

* **Vectorized operations:**

NumPy performs operations element-wise on entire arrays, providing significant performance advantages over traditional Python loops.

* **Universal functions (ufuncs):**

Many common mathematical functions like sin, cos, exp, log are implemented as ufuncs, allowing them to operate directly on arrays.

Example array functions in NumPy:

* **Array creation:**
  + np.array: Creates a NumPy array from a list or other iterable.
  + np.zeros: Creates an array filled with zeros
  + np.ones: Creates an array filled with ones
  + np.arange: Generates a sequence of numbers
* **Arithmetic operations:**
  + +: Addition
  + -: Subtraction
  + \*: Multiplication
  + /: Division
* **Statistical functions:**
  + np.mean: Calculates the mean of array elements
  + np.std: Calculates the standard deviation
  + np.min: Finds the minimum value in an array
  + np.max: Finds the maximum value in an array
  + np.sum: Calculates the sum of array elements
* **Indexing and slicing:**
  + array[i]: Accesses the element at index i
  + array[i:j]: Extracts a slice from index i to j (exclusive)
* **Boolean indexing:**
  + array[condition] : Selects elements where a boolean condition is True
* **Linear algebra operations:**
  + np.dot: Matrix multiplication
  + np.transpose: Transposes a matrix

[**NumPy**](https://www.geeksforgeeks.org/python-numpy/) stands for Numerical Python. It is a Python library used for working with an array. In Python, we use the list for the array but it’s slow to process. NumPy array is a powerful N-dimensional array object and is used in linear algebra, Fourier transform, and random number capabilities. It provides an array object much faster than traditional Python lists.

**Types of Array:**

1. One Dimensional Array
2. Multi-Dimensional Array

**One Dimensional Array:**

A one-dimensional array is a type of linear array.



*One Dimensional Array*

**Example:**

1

# importing numpy module

2

import numpy as np

3

​

4

# creating list

5

list = [1, 2, 3, 4]

6

​

7

# creating numpy array

8

sample\_array = np.array(list)

9

​

10

print("List in python : ", list)

11

​

12

print("Numpy Array in python :",

13

sample\_array)

**Output:**

List in python : [1, 2, 3, 4]  
Numpy Array in python : [1 2 3 4]

Check data type for list and array:

1

print(type(list\_1))

2

​

3

print(type(sample\_array))

**Output:**

<class 'list'>  
<class 'numpy.ndarray'>

**Multi-Dimensional Array:**

Data in multidimensional arrays are stored in tabular form.

A white square with black numbers

Description automatically generated

*Two Dimensional Array*

**Example:**

1

# importing numpy module

2

import numpy as np

3

​

4

# creating list

5

list\_1 = [1, 2, 3, 4]

6

list\_2 = [5, 6, 7, 8]

7

list\_3 = [9, 10, 11, 12]

8

​

9

# creating numpy array

10

sample\_array = np.array([list\_1,

11

list\_2,

12

list\_3])

13

​

14

print("Numpy multi dimensional array in python\n",

15

sample\_array)

**Output:**

Numpy multi dimensional array in python  
[[ 1 2 3 4]  
 [ 5 6 7 8]  
 [ 9 10 11 12]]

**Note:** use **[ ]** operators inside numpy.array() for multi-dimensional

**Anatomy of an array :**

**1. Axis:**The Axis of an array describes the order of the indexing into the array.

*Axis 0 = one dimensional*

*Axis 1 = Two dimensional*

*Axis 2 = Three dimensional*

**2. Shape:**The number of elements along with each axis. It is from a tuple.

**Example:**

1

# importing numpy module

2

import numpy as np

3

​

4

# creating list

5

list\_1 = [1, 2, 3, 4]

6

list\_2 = [5, 6, 7, 8]

7

list\_3 = [9, 10, 11, 12]

8

​

9

# creating numpy array

10

sample\_array = np.array([list\_1,

11

list\_2,

12

list\_3])

13

​

14

print("Numpy array :")

15

print(sample\_array)

16

​

17

# print shape of the array

18

print("Shape of the array :",

19

sample\_array.shape)

**Output:**

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

**Example:**

1

import numpy as np

2

​

3

sample\_array = np.array([[0, 4, 2],

4

[3, 4, 5],

5

[23, 4, 5],

6

[2, 34, 5],

7

[5, 6, 7]])

8

​

9

print("shape of the array :",

10

sample\_array.shape)

**Output:**

shape of the array : (5, 3)

**3. Rank:**The rank of an array is simply the number of axes (or dimensions) it has.

**The one-dimensional array has rank 1.**



*Rank 1*

**The two-dimensional array has rank 2.**

A white square with black numbers

Description automatically generated

*Rank 2*

**4. Data type objects (dtype):**Data type objects (dtype) is an instance of **numpy.dtype** class. It describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted.

**Example:**

1

# Import module

2

import numpy as np

3

​

4

# Creating the array

5

sample\_array\_1 = np.array([[0, 4, 2]])

6

​

7

sample\_array\_2 = np.array([0.2, 0.4, 2.4])

8

​

9

# display data type

10

print("Data type of the array 1 :",

11

sample\_array\_1.dtype)

12

​

13

print("Data type of array 2 :",

14

sample\_array\_2.dtype)

**Output:**

Data type of the array 1 : int32  
Data type of array 2 : float64

**Some different way of creating Numpy Array :**

**1.**[**numpy.array()**](https://www.geeksforgeeks.org/array-python-set-1-introduction-functions/)**:**The Numpy array object in Numpy is called ndarray. We can create ndarray using **numpy.array()**function.

***Syntax:*** *numpy.array(parameter)*

**Example:**

1

# import module

2

import numpy as np

3

​

4

#creating a array

5

​

6

arr = np.array([3,4,5,5])

7

​

8

print("Array :",arr)

**Output:**

Array : [3 4 5 5]

**2. [numpy.fromiter()](https://www.geeksforgeeks.org/numpy-fromiter-function-python/):**The fromiter() function create a new one-dimensional array from an iterable object.

***Syntax:*** *numpy.fromiter(iterable, dtype, count=-1)*

**Example 1:**

1

#Import numpy module

2

import numpy as np

3

​

4

# iterable

5

iterable = (a\*a for a in range(8))

6

​

7

arr = np.fromiter(iterable, float)

8

​

9

print("fromiter() array :",arr)

**Output:**

*fromiter() array :  [ 0.  1.  4.  9. 16. 25. 36. 49.]*

**Example 2:**

1

import numpy as np

2

​

3

var = "Geekforgeeks"

4

​

5

arr = np.fromiter(var, dtype = 'U2')

6

​

7

print("fromiter() array :",

8

arr)

**Output:**

*fromiter() array :  [‘G’ ‘e’ ‘e’ ‘k’ ‘f’ ‘o’ ‘r’ ‘g’ ‘e’ ‘e’ ‘k’ ‘s’]*

**3. [numpy.arange()](https://www.geeksforgeeks.org/numpy-arange-python/):**This is an inbuilt NumPy function that returns evenly spaced values within a given interval.

***Syntax:*** *numpy.arange( start , stop, step , dtype=None )*

**Example:**

1

import numpy as np

2

​

3

np.arange(1, 20 , 2,

4

dtype = np.float32)

**Output:**

*array([ 1.,  3.,  5.,  7.,  9., 11., 13., 15., 17., 19.], dtype=float32)*

**4. [numpy.linspace()](https://www.geeksforgeeks.org/numpy-linspace-python/):**This function returns evenly spaced numbers over a specified between two limits.

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

**Example 1:**

1

import numpy as np

2

​

3

np.linspace(3.5, 10, 3)

**Output:**

array([ 3.5 , 6.75, 10. ])

**Example 2:**

1

import numpy as np

2

​

3

np.linspace(3.5, 10, 3,

4

dtype = np.int32)

**Output:**

array([ 3, 6, 10])

**5. [numpy.empty()](https://www.geeksforgeeks.org/numpy-empty-python/):**This function create a new array of given shape and type, without initializing value.

***Syntax:*** *numpy.empty(shape, dtype=float, order=’C’)*

**Example:**

1

import numpy as np

2

​

3

np.empty([4, 3],

4

dtype = np.int32,

5

order = 'f')

**Output:**

array([[ 1, 5, 9],  
 [ 2, 6, 10],  
 [ 3, 7, 11],  
 [ 4, 8, 12]])

**6. [numpy.ones():](https://www.geeksforgeeks.org/numpy-ones-python/)**This function is used to get a new array of given shape and type, filled with ones(1).

***Syntax:*** *numpy.ones(shape, dtype=None, order=’C’)*

**Example:**

1

import numpy as np

2

​

3

np.ones([4, 3],

4

dtype = np.int32,

5

order = 'f')

**Output:**

array([[1, 1, 1],  
 [1, 1, 1],  
 [1, 1, 1],  
 [1, 1, 1]])

**7. [numpy.zeros()](https://www.geeksforgeeks.org/numpy-zeros-python/):**This function is used to get a new array of given shape and type, filled with zeros(0).

***Syntax:*** *numpy.ones(shape, dtype=None)*

**Example:**

1

import numpy as np

2

np.zeros([4, 3],

3

dtype = np.int32,

4

order = 'f')

**Output:**

array([[0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0]])

**Numpy - Array Creation**

Last Updated : 24 Jan, 2025

[**Numpy Arrays**](https://www.geeksforgeeks.org/basics-of-numpy-arrays/) are grid-like structures similar to lists in [Python](https://www.geeksforgeeks.org/python-programming-language-tutorial/) but optimized for numerical operations. The most straightforward way to create a NumPy array is by converting a regular Python list into an array using the np.array() function.

Let's understand this with the help of an example:

1

import numpy as np

2

​

3

# One-dimensional array

4

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

5

print(arr1)

6

​

7

# Two-dimensional array

8

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

9

print(arr2)

**Output**

[1 2 3 4 5]

[[1 2]

[3 4]]

**Creating Arrays with Specific Values**

For assigning a specific values. NumPy provides several function to create arrays filled with zeros, ones, or a specific constant value.

* **Zeros Array:**[np.zeros() function](https://www.geeksforgeeks.org/numpy-zeros-python/" \t "_blank) creates an array filled with zeros. It requires the shape of the array as a parameter.

**Example:**

1

import numpy as np

2

​

3

# 3x4 array filled with zeros

4

arr\_zero = np.zeros((3, 4))

5

print(arr\_zero)

**Output**

[[0. 0. 0. 0.]

[0. 0. 0. 0.]

[0. 0. 0. 0.]]

* **Ones Array:**[np.ones()](https://www.geeksforgeeks.org/numpy-ones-python/" \t "_blank) creates an array filled with ones.

**Example:**

1

import numpy as np

2

​

3

# 2x3 array filled with ones

4

arr\_one = np.ones((2, 3))

5

print(arr\_one)

**Output**

[[1. 1. 1.]

[1. 1. 1.]]

* **Full Array :**[np.full()](https://www.geeksforgeeks.org/numpy-full-python/" \t "_blank) function allows you to create an array filled with a specific value.

**Example:**

1

import numpy as np

2

​

3

# 2x2 array filled with 7

4

arr\_full = np.full((2, 2), 7)

5

print(arr\_full)

**Output**

[[7 7]

[7 7]]

**Creating Arrays with Random Values**

NumPy also has functions for generating arrays with random values, useful for simulations and testing.

* **Random Float Array :** [np.random.rand()](https://www.geeksforgeeks.org/numpy-random-rand-python/" \t "_blank) function generates an array of random values between 0 and 1.

**Example:**

1

import numpy as np

2

​

3

# 2x3 array of random floats

4

arr\_rand = np.random.rand(2, 3)

5

print(arr\_rand)

**Output**

[[0.67820861 0.64484802 0.48673431]

[0.00263043 0.55383721 0.43240166]]

* **Random Integers :**If we need random integers, we can use **np.random.randint()** to create arrays with integer values in a specified range.

**Example:**

1

import numpy as np

2

​

3

# 3x3 array of random integers from 1 to 9

4

arr\_int = np.random.randint(1, 10, size=(3, 3))

5

print(arr\_int)

**Output**

[[4 6 5]

[7 4 8]

[8 5 2]]

**Creating Arrays with a Range of Values**

Another common method of creating arrays is using a range of values. NumPy provides functions like np.arange() and np.linspace() for this purpose.

* **Using np.arange() :**[np.arange()](https://www.geeksforgeeks.org/numpy-arrange-in-python/" \t "_blank) creates arrays with values spaced according to a given interval. It’s similar to Python’s built-in range() but returns a NumPy array.

**Example:**

1

import numpy as np

2

​

3

# Array from 0 to 10 with step 2

4

arr\_range = np.arange(0, 10, 2)

5

print(arr\_range)

**Output**

[0 2 4 6 8]

* **Using np.linspace():** [np.linspace()](https://www.geeksforgeeks.org/numpy-linspace/" \t "_blank)generates an array with a specified number of evenly spaced values over a specified range.

**Example:**

1

import numpy as np

2

​

3

# 5 values from 0 to 1

4

arr\_linspace = np.linspace(0, 1, 5)

5

print(arr\_linspace)

**Output**

[0. 0.25 0.5 0.75 1. ]

**Identity and Diagonal Matrices**

NumPy also provides functions for creating identity matrices and diagonal matrices, which are often used in linear algebra.

* **Identity Matrix :**[np.eye()](https://www.geeksforgeeks.org/numpy-eye-python/" \t "_blank) function creates an identity matrix, a square matrix with ones on the diagonal and zeros elsewhere.

**Example:**

1

import numpy as np

2

​

3

# 3x3 identity matrix

4

identity\_matrix = np.eye(3)

5

print(identity\_matrix)

**Output**

[[1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.]]

* **Diagonal Matrix :**Use [np.diag()](https://www.geeksforgeeks.org/numpy-diag-python/" \t "_blank) to create a diagonal matrix, where the provided array elements form the diagonal.

**Example:**

1

import numpy as np

2

​

3

# Diagonal matrix with [1, 2, 3] on the diagonal

4

diag\_matrix = np.diag([1, 2, 3])

5

print(diag\_matrix)

**Output**

[[1 0 0]

[0 2 0]

[0 0 3]]

**Methods for array creation in Numpy**

| **Function** | **Description** |
| --- | --- |
| [**empty()**](https://www.geeksforgeeks.org/numpy-empty-python/) | Return a new array of given shape and type, without initializing entries |
| [**empty\_like()**](https://www.geeksforgeeks.org/numpy-empty_like-python/) | Return a new array with the same shape and type as a given array |
| [**eye()**](https://www.geeksforgeeks.org/numpy-eye-python/) | Return a 2-D array with ones on the diagonal and zeros elsewhere. |
| [**identity()**](https://www.geeksforgeeks.org/numpy-identity-python/) | Return the identity array |
| [**ones()**](https://www.geeksforgeeks.org/numpy-ones-python/) | Return a new array of given shape and type, filled with ones |
| [**ones\_like()**](https://www.geeksforgeeks.org/numpy-ones_like-python/) | Return an array of ones with the same shape and type as a given array |
| [**zeros()**](https://www.geeksforgeeks.org/numpy-zeros-python/) | Return a new array of given shape and type, filled with zeros |
| [**zeros\_like()**](https://www.geeksforgeeks.org/numpy-zeros_like-python/) | Return an array of zeros with the same shape and type as a given array |
| [**full\_like()**](https://www.geeksforgeeks.org/numpy-full_like-python/) | Return a full array with the same shape and type as a given array. |
| [**array()**](https://www.geeksforgeeks.org/python-array-length/) | Create an array |
| [**asarray()**](https://www.geeksforgeeks.org/numpy-asarray-in-python/) | Convert the input to an array |
| [**asanyarray()**](https://www.geeksforgeeks.org/numpy-asanyarray-in-python/) | Convert the input to an ndarray, but pass ndarray subclasses through |
| [**ascontiguousarray()**](https://www.geeksforgeeks.org/numpy-ascontiguousarray-in-python/) | Return a contiguous array in memory (C order) |
| [**asmatrix()**](https://www.geeksforgeeks.org/numpy-asmatrix-python/) | Interpret the input as a matrix |
| **copy()** | Return an array copy of the given object |
| [**frombuffer()**](https://www.geeksforgeeks.org/numpy-frombuffer-function-python/) | Interpret a buffer as a 1-dimensional array |
| **fromfile()** | Construct an array from data in a text or binary file |
| [**fromfunction()**](https://www.geeksforgeeks.org/numpy-fromfunction-function-python/) | Construct an array by executing a function over each coordinate |
| [**fromiter()**](https://www.geeksforgeeks.org/numpy-fromiter-function-python/) | Create a new 1-dimensional array from an iterable object |
| [**fromstring()**](https://www.geeksforgeeks.org/numpy-fromstring-function-python/) | A new 1-D array initialized from text data in a string |
| [**loadtxt()**](https://www.geeksforgeeks.org/numpy-loadtxt-in-python/) | Load data from a text file |
| [**arange()**](https://www.geeksforgeeks.org/numpy-arange-python/) | Return evenly spaced values within a given interval |
| [**linspace()**](https://www.geeksforgeeks.org/numpy-linspace-python/) | Return evenly spaced numbers over a specified interval |
| [**logspace()**](https://www.geeksforgeeks.org/numpy-logspace-python/) | Return numbers spaced evenly on a log scale |
| [**geomspace()**](https://www.geeksforgeeks.org/numpy-geomspace-in-python/) | Return numbers spaced evenly on a log scale (a geometric progression) |
| [**meshgrid()**](https://www.geeksforgeeks.org/numpy-meshgrid-function/) | Return coordinate matrices from coordinate vectors |
| **mgrid()** | nd\_grid instance which returns a dense multi-dimensional “meshgrid |
| **ogrid()** | nd\_grid instance which returns an open multi-dimensional “meshgrid |
| [**diag()**](https://www.geeksforgeeks.org/numpy-diag-python/) | Extract a diagonal or construct a diagonal array |
| [**diagflat()**](https://www.geeksforgeeks.org/numpy-diagflat-python/) | Create a two-dimensional array with the flattened input as a diagonal |
| [**tri()**](https://www.geeksforgeeks.org/numpy-tri-python/) | An array with ones at and below the given diagonal and zeros elsewhere |
| [**tril()**](https://www.geeksforgeeks.org/numpy-tril-python/) | Lower triangle of an array |
| [**triu()**](https://www.geeksforgeeks.org/numpy-triu-python/) | Upper triangle of an array |
| [**vander()**](https://www.geeksforgeeks.org/numpy-vander-function-python/) | Generate a Vandermonde matrix |
| **mat()** | Interpret the input as a matrix |
| [**bmat()**](https://www.geeksforgeeks.org/numpy-bmat-python/) | Build a matrix object from a string, nested sequence, or array |

**Saving and loading NumPy Arrays**

Last Updated : 26 Dec, 2023

The savetxt() and loadtxt() functions in NumPy are primarily designed for 1D and 2D arrays (text files with row/column format). When dealing with a 3D NumPy array, these functions can be a bit limited because they cannot directly handle the 3D structure. However, you can reshape the 3D array into a 2D array, save it, and then reshape it back to its original form upon loading. In this article, we will see how to load and save 3D [NumPy Array](https://www.geeksforgeeks.org/basics-of-numpy-arrays/) to file using [savetxt()](https://www.geeksforgeeks.org/numpy-savetxt/) and [loadtxt()](https://www.geeksforgeeks.org/numpy-loadtxt-in-python/) functions and NumPy loadtxt and savetxt usage guide.

**Load and Save 3D Numpy Array to File**

Below are the ways by which we can load and save 3D [NumPy](https://www.geeksforgeeks.org/python-numpy/)array to file using savetxt() and loadtxt() functions in [Python](https://www.geeksforgeeks.org/python-programming-language/):

* Utilize the savetxt() and loadtxt() functions for TXT files
* Saving and loading the 3D arrays(reshaped) into CSV files

**Example 1: Saving a 3D Numpy Array as a Text File**

In this example, a 3D NumPy array arr is reshaped into a 2D format and saved to a text file named “geekfile.txt” using savetxt(). Later, the data is retrieved from the file, reshaped back to its original 3D form, and compared with the original array to verify its equality.

* Python3

|  |
| --- |
| **import** numpy as gfg    arr **=** gfg.random.rand(5, 4, 3)    # reshaping the array from 3D  # matrice to 2D matrice.  arr\_reshaped **=** arr.reshape(arr.shape[0], **-**1)    # saving reshaped array to file.  gfg.savetxt("geekfile.txt", arr\_reshaped)    # retrieving data from file.  loaded\_arr **=** gfg.loadtxt("geekfile.txt")    load\_original\_arr **=** loaded\_arr.reshape(      loaded\_arr.shape[0], loaded\_arr.shape[1] **//** arr.shape[2], arr.shape[2])    # check the shapes:  print("shape of arr: ", arr.shape)  print("shape of load\_original\_arr: ", load\_original\_arr.shape)    # check if both arrays are same or not:  **if** (load\_original\_arr **==** arr).all():  **print**("Yes, both the arrays are same")  **else**:      print("No, both the arrays are not same") |

**Output:**

shape of arr: (5, 4, 3)

shape of load\_original\_arr: (5, 4, 3)

Yes, both the arrays are same

**Example 2: Saving and loading the 3D arrays(reshaped) into CSV files**

In this example, we will perform saving and loading the 3D arrays(reshaped) into [CSV files](https://www.geeksforgeeks.org/working-csv-files-python/) by using savetxt and loadtxt functions respectively. Here, a random 3D NumPy array arr is reshaped into a 2D format, saved as a CSV file, and then loaded back into a 2D array. The loaded data is reshaped back to its original 3D form, and a comparison is made with the original array to confirm their equality.

* Python3

|  |
| --- |
| **import** numpy as np    # Create a sample 3D array  arr **=** np.random.rand(5, 4, 3)    # Reshape the 3D array to 2D  arr\_reshaped **=** arr.reshape(arr.shape[0], **-**1)    # Save the 2D array to a CSV file  np.savetxt("3d\_array.csv", arr\_reshaped, delimiter**=**",")    # Load the 2D array from the CSV file  loaded\_arr **=** np.loadtxt("3d\_array.csv", delimiter**=**",")    # Reshape the 2D array back to its original 3D shape  load\_original\_arr **=** loaded\_arr.reshape((arr.shape[0], arr.shape[1], arr.shape[2]))    # Verify if the loaded array matches the original  **if** np.array\_equal(load\_original\_arr, arr):      print("Yes, both the arrays are the same")  **else**:      print("No, both the arrays are not the same") |

**Output:**

Yes, both the arrays are same

**NumPy save() Method | Save Array to a File**

Last Updated : 02 Feb, 2024

The NumPy**save()**method is used to **store the input array**in a binary file with the ‘**npy extension’** (.npy).

**Example:**

* Python3

|  |
| --- |
| **import** numpy as np  a **=** np.arange(5)  np.save('array\_file', a) |

**Syntax**

***Syntax:*** *numpy.save(file, arr, allow\_pickle=True, fix\_imports=True)*

***Parameters:***

* ***file:*** *File or filename to which the data is saved. If the file is a string or Path, a .npy extension will be appended to the file name if it does not already have one. If the file is a file object, then the filename is unchanged.*
* ***allow\_pickle :*** *Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations). Default: True*
* ***fix\_imports :*** *Only useful in forcing objects in object arrays on Python 3 to be pickled in a Python 2 compatible way.*
* ***arr :*** *Array data to be saved.*

***Returns:*** *Stores the input array in a disk file with ‘.npy’ extension.*

**Examples**

Let’s understand the workings of numpy.save() method in these Python code and know how to use save() method of [NumPy library](https://www.geeksforgeeks.org/python-numpy/).

To use numpy.save() function, you just need to pass the file name and array in the function.

**Example 1**

* Python3

|  |
| --- |
| # Python program explaining  # save() function    **import** numpy as geek    a **=** geek.arange(5)    # a is printed.  **print**("a is:")  **print**(a)    # the array is saved in the file geekfile.npy  geek.save('geekfile', a)    print("the array is saved in the file geekfile.npy") |

**Output :**

a is:

[0 1 2 3 4]

the array is saved in the file geekfile.npy

**Example 2**

* Python3

|  |
| --- |
| # Python program explaining  # save() function    **import** numpy as geek    # the array is loaded into b  b **=** geek.load('geekfile.npy')    print("b is:")  print(b)    # b is printed from geekfile.npy  print("b is printed from geekfile.npy") |

**Output :**

b is:

[0 1 2 3 4]

b is printed from geekfile.npy

NUMPY RANDOM NUMBER

What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

Pseudo Random and True Random.

Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well.

If there is a program to generate random number it can be predicted, thus it is not truly random.

Random numbers generated through a generation algorithm are called *pseudo random*.

Can we make truly random numbers?

Yes. In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc.

We do not need truly random numbers, unless it is related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

Generate Random Number

NumPy offers the random module to work with random numbers.

Example[Get your own Python Server](https://www.w3schools.com/python/python_server.asp)

Generate a random integer from 0 to 100:

from numpy import random  
  
x = random.randint(100)  
  
print(x)

Generate Random Float

The random module's rand() method returns a random float between 0 and 1.

Example

Generate a random float from 0 to 1:

from numpy import random  
  
x = random.rand()  
  
print(x)

Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers

The randint() method takes a size parameter where you can specify the shape of an array.

Example

Generate a 1-D array containing 5 random integers from 0 to 100:

from numpy import random  
  
x=random.randint(100, size=(5))  
  
print(x)

Example

Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

from numpy import random  
  
x = random.randint(100, size=(3, 5))  
  
print(x)

Floats

The rand() method also allows you to specify the shape of the array.

Example

Generate a 1-D array containing 5 random floats:

from numpy import random  
  
x = random.rand(5)  
  
print(x)

Example

Generate a 2-D array with 3 rows, each row containing 5 random numbers:

from numpy import random  
  
x = random.rand(3, 5)  
  
print(x)

Generate Random Number From Array

The choice() method allows you to generate a random value based on an array of values.

The choice() method takes an array as a parameter and randomly returns one of the values.

Example

Return one of the values in an array:

from numpy import random  
  
x = random.choice([3, 5, 7, 9])  
  
print(x)

The choice() method also allows you to return an *array* of values.

Add a size parameter to specify the shape of the array.

Example

Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, and 9):

from numpy import random  
  
x = random.choice([3, 5, 7, 9], size=(3, 5))  
  
print(x)

**Numpy Random**

In NumPy, we have a module called random which provides functions for generating random numbers.

These functions can be useful for generating random inputs for testing algorithms.

**Generate Random Integer in NumPy**

As discussed earlier, we use the random module to work with random numbers in NumPy.

Let's see an example.

import numpy as np

# generate random integer from 0 to 9

random\_number = np.random.randint(0, 10)

print(random\_number)

# Output: 7

In this example, we have used the random module to generate a random number. The random.randint() function takes two arguments,

* **0** - a lower bound (inclusive)
* **10** - an upper bound (exclusive)

Here, random.randint() returns a random integer between **0** and **9**.

Since the output will be a randomly generated integer between **0** and **9**, we will see different outputs each time the code is run.

**Note**: We can also import and use the random module like this:

from numpy import random

random\_number = random.randint(0, 10)

print(random\_number)

Here, the syntax is slightly different but the output will be the same as above, we will get a random integer between **0** and **9**.

**Generate Random Float in NumPy**

We can also generate a random floating-point number between **0** and **1**. For that we use the random.rand() function. For example,

import numpy as np

# generate random float-point number between 0 and 1

random\_number = np.random.rand()

print(random\_number)

# Output: 0.7696638323107154

Here, random.rand() generates a random floating-point number between **0** and **1**.

Since the number is generated randomly, the output value can vary each time the code is run.

**Generate Random Array in NumPy**

NumPy's random module can also be used to generate an array of random numbers. For example,

import numpy as np

# generate 1D array of 5 random integers between 0 and 9

integer\_array = np.random.randint(0, 10, 5)

print("1D Random Integer Array:\n",integer\_array)

# generate 1D array of 5 random numbers between 0 and 1

float\_array = np.random.rand(5)

print("\n1D Random Float Array:\n",float\_array)

# generate 2D array of shape (3, 4) with random integers

result = np.random.randint(0, 10, (3,4))

print("\n2D Random Integer Array:\n",result)

**Output**

1D Random Integer Array:

[9 7 8 4 2]

1D Random Float Array:

[0.7877579 0.01723754 0.93995075 0.17126388 0.69913594]

2D Random Integer Array:

[[0 5 3 8]

[3 9 2 1]

[8 7 1 2]]

Here,

* np.random.randint(0, 10, 5) - generates a 1D array of **5** random integers between **0** and **9**.
* np.random.rand(5) - generates a 1D array of **5** random numbers between **0** and **1**.
* np.random.randint(0, 10, (3,4)) - generates a 2D array of shape **(3, 4)** with random integers between **0** and **9**.

**Choose Random Number from NumPy Array**

To choose a random number from a NumPy array, we can use the random.choice() function.

Let's see an example.

import numpy as np

# create an array of integers from 1 to 5

array1 = np.array([1, 2, 3, 4, 5])

# choose a random number from array1

random\_choice = np.random.choice(array1)

print(random\_choice)

# Output: 3

In the above example, the np.random.choice(array1) function chooses a random number from the array1 array.

It is important to note that the output will be a single random number from array1, which will be different each time the code is run.