## What is NumPy

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
5. NumPy provides in-built functions for linear algebra & random number genration.

Nowadays, NumPy in combination with SciPy and Mat-plotlib is used as replacement to MATLAB as Python is more complete and easier programming language than MATLAB.

# NumPy Ndarray

Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define a ndarray as the collection of the data type (dtype) objects.

The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

## Creating a ndarray object

**numpy.array(object, dtype =None, copy = True, order = None, subok = False, ndmin = 0)**

The parameters are described in the following table.

| **Parameter** | **Description** |
| --- | --- |
| object | It represents the collection object. It can be a list, tuple, dictionary, set, etc. |
| dtype | We can change the data type of the array elements by changing this option to the specified type. The default is none. |
| copy | It is optional. By default, it is true which means the object is copied. |
| order | There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any) |
| subok | The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true. |
| ndmin | It represents the minimum dimensions of the resultant array. |

**To create an array using the list:- >>> a = numpy.array([1, 2, 3])**

**To create a multi-dimensional array object,: >>> a = numpy.array([[1, 2, 3], [4, 5, 6]])**

**To change the data type of the array elements, mention the name of the data type along with the collection.**

1. **>>> a = numpy.array([1, 3, 5, 7], complex)**

## Finding the dimensions of the Array

**The ndim function can be used to find the dimensions of the array.**

1. **import numpy as np**
2. **arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])**
3. **print(arr.ndim)**

## Finding the size of each array element

The itemsize function is used to get the size of each array item. It returns the number of bytes taken by each array element.

1. a = np.array([[1,2,3]])
2. **print**("Each item contains",a.itemsize,"bytes")

## Finding the data type of each array item

To check the data type of each array item, the dtype function is used. Consider the following example to check the data type of the array items.

1. a = np.array([[1,2,3]])
2. **print**("Each item is of the type",a.dtype)

## Finding the shape and size of the array

To get the shape and size of the array, the size and shape function associated with the numpy array is used.

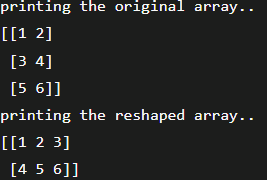
1. a = np.array([[1,2,3,4,5,6,7]])
2. **print**("Array Size:",a.size) **//7**
3. **print**("Shape:",a.shape) **//(1,7)**

## Reshaping the array objects

The numpy module provides us the way to reshape the array by changing the number of rows and columns of the multi-dimensional array.

The reshape() function associated with the ndarray object is used to reshape array. It accepts the two parameters indicating the row and columns of new shape of the array.

1. a = np.array([[1,2],[3,4],[5,6]])
2. **print**("printing the original array..")
3. **print**(a)
4. a=a.reshape(2,3)
5. **print**("printing the reshaped array..")
6. **print**(a)



## Slicing in the Array

Slicing in the NumPy array is the way to extract a range of elements from an array. Slicing in the array is performed in the same way as it is performed in the python list.

1. a = np.array([[1,2],[3,4],[5,6]])
2. **print**(a[0,1]) **//2**
3. **print**(a[2,0]) **//5**

The above program prints the 2nd element from the 0th index and 0th element from the 2nd index of the array.

## Linspace

The linspace() function returns the evenly spaced values over the given interval. The following example returns the 10 evenly separated values over the given interval 5-15

The **numpy.linspace()** function returns number spaces evenly w.r.t interval. Similar to [numpy.arange() function](https://www.geeksforgeeks.org/numpy-arange-python/) but instead of step it uses sample number.

1. a=np.linspace(5,15,10) #prints 10 values which are evenly spaced over the given interval 5-15
2. **print**(a)



## Finding maximum, minimum, & sum of array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum, minimum, and sum of the array elements respectively.

1. a = np.array([1,2,3,10,15,4])
2. **print**("The array:",a) **//[ 1 2 3 10 15 4]**
3. **print**("The maximum element:",a.max()) **//15**
4. **print**("The minimum element:",a.min())  **//1**
5. **print**("The sum of the elements:",a.sum()) **//35**

## NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where axis-0 represents the columns and axis-1 represents the rows. We can mention the axis to perform row-level or column-level calculations like the addition of row or column elements.

To calculate the maximum element among each column, the minimum element among each row, and the addition of all the row elements,

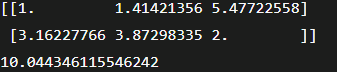
1. a = np.array([[1,2,30],[10,15,4]])
2. **print**("The maximum elements of columns:",a.max(axis = 0)) //[10 15 30]
3. **print**("The minimum element of rows",a.min(axis = 1)) //[1 4]
4. **print**("The sum of all rows",a.sum(axis = 1)) //[33 29]

## Finding square root and standard deviation

The sqrt() and std() functions associated with the numpy array are used to find the square root and standard deviation of the array elements respectively.

Standard deviation means how much each element of the array varies from the mean value of the numpy array.

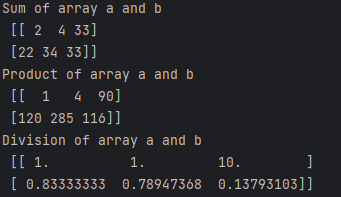
1. a = np.array([[1,2,30],[10,15,4]])
2. **print**(np.sqrt(a))
3. **print**(np.std(a))



## Arithmetic operations on the array

The numpy module allows us to perform the arithmetic operations on multi-dimensional arrays directly.

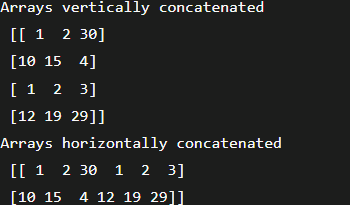
1. a = np.array([[1,2,30],[10,15,4]])
2. b = np.array([[1,2,3],[12, 19, 29]])
3. **print**("Sum of array a and b\n",a+b)
4. **print**("Product of array a and b\n",a\*b)
5. **print**("Division of array a and b\n",a/b)



## Array Concatenation

The numpy provides us with the vertical stacking and horizontal stacking which allows us to concatenate two multi-dimensional arrays vertically or horizontally.

1. a = np.array([[1,2,30],[10,15,4]])
2. b = np.array([[1,2,3],[12, 19, 29]])
3. **print**("Arrays vertically concatenated\n",np.vstack((a,b)));
4. **print**("Arrays horizontally concatenated\n",np.hstack((a,b)))



# NumPy Datatypes

| **Data type** | **Description** |
| --- | --- |
| bool\_ | It represents the boolean value indicating true or false. It is stored as a byte. |
| int\_ | It is default type of integer. It is identical to long type in C that contains 64 bit or 32-bit integer. |
| intc | It is similar to the C integer (c int) as it represents 32 or 64-bit int. |
| intp | It represents the integers which are used for indexing. |
| int8 | It is the 8-bit integer identical to a byte. The range of the value is -128 to 127. |
| int16 | It is the 2-byte (16-bit) integer. The range is -32768 to 32767. |
| int32 | It is the 4-byte (32-bit) integer. The range is -2147483648 to 2147483647. |
| int64 | It is the 8-byte (64-bit) integer. The range is -9223372036854775808 to 9223372036854775807. |
| uint8 | It is the 1-byte (8-bit) unsigned integer. |
| uint16 | It is the 2-byte (16-bit) unsigned integer. |
| uint32 | It is the 4-byte (32-bit) unsigned integer. |
| uint64 | It is the 8 bytes (64-bit) unsigned integer. |
| float\_ | It is identical to float64. |
| float16 | It is the half-precision float. 5 bits are reserved for the exponent. 10 bits are reserved for mantissa, and 1 bit is reserved for the sign. |
| float32 | It is a single precision float. 8 bits are reserved for the exponent, 23 bits are reserved for mantissa, and 1 bit is reserved for the sign. |
| float64 | It is the double precision float. 11 bits are reserved for the exponent, 52 bits are reserved for mantissa, 1 bit is used for the sign. |
| complex\_ | It is identical to complex128. |
| complex64 | It is used to represent the complex number where real and imaginary part shares 32 bits each. |
| complex128 | It is used to represent the complex number where real and imaginary part shares 64 bits each. |

## NumPy dtype

All the items of a numpy array are data type objects also known as numpy dtypes. A data type object implements the fixed size of memory corresponding to an array.

**Syntax:-** numpy.dtype(object, align, copy)

The constructor accepts the following object.

**Object:** It represents the object which is to be converted to the data type.

**Align:** It can be set to any boolean value. If true, then it adds extra padding to make it equivalent to a C struct.

**Copy:** It creates another copy of the dtype object.

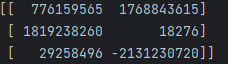
# Numpy Array Creation

The ndarray object can be constructed by using the following routines.

## 1)Numpy.empty:- As the name specifies, The empty routine is used to create an uninitialized array of specified shape and data type.

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

* **Shape:** The desired shape of the specified array.
* **dtype:** The data type of the array items. The default is the float.
* **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.



## 2)NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 0.

**Syntax**: numpy.zeros(shape, dtype = float, order = 'C')

* **Shape:** The desired shape of the specified array.
* **dtype:** The data type of the array items. The default is the float.
* **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

1. arr = np.zeros((3,2), dtype = int)
2. **print**(arr)



## 3)NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 1.

**Syntax**: numpy.ones(shape, dtype = none, order = 'C')

1. arr = np.ones((3,2), dtype = int)
2. **print**(arr)



# Numpy array from existing data

NumPy provides us the way to create an array by using the existing data.

## 1)numpy.asarray

This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

**Syntax**: numpy.asarray(sequence, dtype = None, order = None)

* **sequence:** It is the python sequence which is to be converted into python array.
* **dtype:** It is the data type of each item of the array.
* **order:** It can be set to C or F. The default is C.

1. l=[1,2,3,4,5,6,7]
2. a = np.asarray(l);
3. **print**(type(a))
4. **print**(a)



### Example: creating a numpy array using more than one list

1. l=[[1,2,3,4,5,6,7],[8,9]]
2. a = np.asarray(l);
3. **print**(type(a))
4. **print**(a)

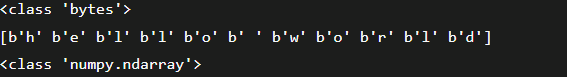
## 2)numpy.frombuffer

This function is used to create an array by using the specified buffer.

**Syntax**:- numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)

* **buffer:** It represents an object that exposes a buffer interface.
* **dtype:** It represents the data type of the returned data type array. The default value is 0.
* **count:** It represents the length of the returned ndarray. The default value is -1.
* **offset:** It represents the starting position to read from. The default value is 0.

1. l = b'hello world'
2. **print**(type(l))
3. a = np.frombuffer(l, dtype = "S1")
4. **print**(a)
5. **print**(type(a))



## 3)numpy.fromiter

This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

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

1. list = [0,2,4,6]
2. it = iter(list)
3. x = np.fromiter(it, dtype = float)
4. **print**(x)
5. **print**(type(x))



# Numpy Arrays within the numerical range

## 1)Numpy.arrange: It creates an array by using the evenly spaced values over the given interval.

**Syntax**: numpy.arrange(start, stop, step, dtype)

1. **start:** The starting of an interval. The default is 0.
2. **stop:** represents the value at which the interval ends excluding this value.
3. **step:** The number by which the interval values change.
4. **dtype:** the data type of the numpy array items.

* arr = np.arange(0,10,2,float)
* **print**(arr) //[0. 2. 4. 6. 8.]

1. arr = np.arange(10,100,5,int)
2. **print**("The array over the given range is ",arr)



## 2)NumPy.linspace

It is similar to the arrange function. However, it doesn?t allow us to specify the step size in the syntax.

Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size.

**Syntax**:- numpy.linspace(start, stop, num, endpoint, retstep, dtype)

1. **start:** It represents the starting value of the interval.
2. **stop:** It represents the stopping value of the interval.
3. **num:** The amount of evenly spaced samples over the interval to be generated. The default is 50.
4. **endpoint:** Its true value indicates that the stopping value is included in interval.
5. **rettstep:** This has to be a boolean value. Represents the steps and samples between the consecutive numbers.
6. **dtype:** It represents the data type of the array items.
7. **import** numpy as np
8. arr = np.linspace(10, 20, 5)
9. **print**("The array over the given range is ",arr) //[10. 12.5 15. 17.5 20.]
10. arr = np.linspace(10, 20, 5, endpoint = False)
11. **print**("The array over the given range is ",arr) [10. 12. 14. 16. 18.]

## 3)numpy.logspace

It creates an array by using the numbers that are evenly separated on a log scale.

**Syntax**: numpy.logspace(start, stop, num, endpoint, base, dtype)

1. **base:** It represents the base of the log space.



1. arr = np.logspace(10, 20, num = 5,base = 2, endpoint = True)
2. **print**("The array over the given range is ",arr)



# NumPy Broadcasting

In Mathematical operations, we may need to consider the arrays of different shapes. NumPy can perform such operations where the array of different shapes are involved.

For example, if we consider the matrix multiplication operation, if the shape of the two matrices is the same then this operation will be easily performed. However, we may also need to operate if the shape is not similar.

Consider the following example to multiply two arrays.

1. a = np.array([1,2,3,4,5,6,7])
2. b = np.array([2,4,6,8,10,12,14])
3. c = a\*b; //[ 2 8 18 32 50 72 98]

However, in the above example, if we consider arrays of different shapes, we will get the errors as shown below.

1. a = np.array([1,2,3,4,5,6,7])
2. b = np.array([2,4,6,8,10,12,14,19])
3. c = a\*b;



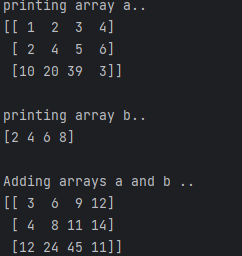
## Broadcasting Rules

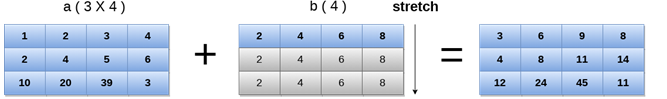
Broadcasting is possible if the following cases are satisfied.

1. The smaller dimension array can be appended with '1' in its shape.
2. Size of each output dimension is the maximum of the input sizes in dimension.
3. An input can be used in the calculation if its size in a particular dimension matches the output size or its value is exactly 1.
4. If input size is 1, then first data entry is used for the calculation along dimension.

**Broadcasting** can be applied to the arrays if the following rules are satisfied.

1. All the input arrays have the same shape.
2. Arrays have the same number of dimensions, and the length of each dimension is either a common length or 1.
3. Array with the fewer dimension can be appended with '1' in its shape.
4. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
5. b = np.array([2,4,6,8])
6. **print**(a)
7. **print**(b)
8. **print**(a+b)

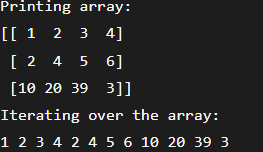




# NumPy Array Iteration

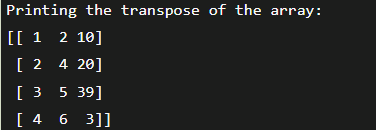
NumPy provides an iterator object, i.e., **nditer** which can be used to iterate over the given array using python standard Iterator interface.

1. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
2. **print**("Printing array:")
3. **print**(a);
4. **print**("Iterating over the array:")
5. **for** x **in** np.nditer(a):
6. **print**(x,end=' ')



Order of the iteration doesn't follow any special ordering like row-major or column-order. However, it is intended to match the memory layout of the array.

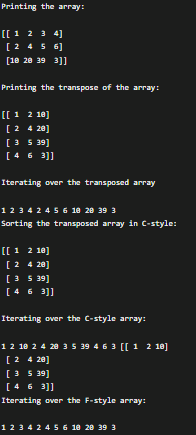
1. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
2. at = a.T //Printing the transpose of the array
3. **print**(at)



## Order of Iteration

As we know, there are two ways of storing values into the numpy arrays:

1. F-style order
2. C-style order
3. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
4. **print**(a)
5. at = a.T //Printing the transpose of the array
6. **print**(at)
7. **for** x **in** np.nditer(at): //Iterating over the transposed array
8. **print**(x, end= ' ')
9. c = at.copy(order = 'C') //Sorting the transposed array in C-style:
10. **print**(c)
11. **for** x **in** np.nditer(c): //Iterating over the C-style array:
12. **print**(x,end=' ')
13. d = at.copy(order = 'F')
14. **print**(d)
15. **for** x **in** np.nditer(d): //Iterating over the F-style array:
16. **print**(x,end=' ')

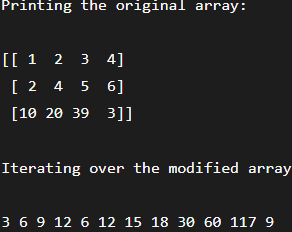


## Array Values Modification

We can not modify the array elements during the iteration since the op-flag associated with the Iterator object is set to readonly.

However, we can set this flag to readwrite or write only to modify the array values.

1. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
2. **print**(a) //Printing the original array:
3. **for** x **in** np.nditer(a, op\_flags = ['readwrite']): // Iterating over the modified array
4. x[...] = 3 \* x;
5. **print**(x,end = ' ')



# NumPy Bitwise Operators

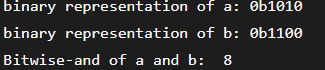
| **Operator** | **Description** |
| --- | --- |
| bitwise\_and | It is used to calculate the bitwise and operation between the corresponding array elements. |
| bitwise\_or | It is used to calculate the bitwise or operation between the corresponding array elements. |
| invert | It is used to calculate the bitwise not the operation of the array elements. |
| left\_shift | It is used to shift the bits of the binary representation of the elements to the left. |
| right\_shift | It is used to shift the bits of the binary representation of the elements to the right. |

## bitwise\_and Operation

The NumPy provides the bitwise\_and() function which is used to calculate the bitwise\_and operation of the two operands.

The bitwise and operation is performed on the corresponding bits of the binary representation of the operands. If both the corresponding bit in the operands is set to 1, then only the resultant bit in the AND result will be set to 1 otherwise it will be set to 0.

1. a,b = 10,12
2. **print**("binary representation of a:",bin(a))
3. **print**("binary representation of b:",bin(b))
4. **print**("Bitwise-and of a and b: ",np.bitwise\_and(a,b)) //1000

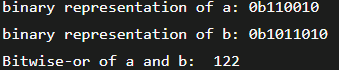


## bitwise\_or Operator

The NumPy provides the bitwise\_or() function which is used to calculate the bitwise or operation of the two operands.

The bitwise or operation is performed on the corresponding bits of the binary representation of the operands. If one of the corresponding bit in the operands is set to 1 then the resultant bit in the OR result will be set to 1; otherwise it will be set to 0.

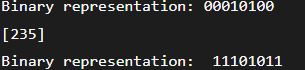
1. a = 50
2. b = 90
3. **print**("binary representation of a:",bin(a))
4. **print**("binary representation of b:",bin(b))
5. **print**("Bitwise-or of a and b: ",np.bitwise\_or(a,b))



## Invert operation

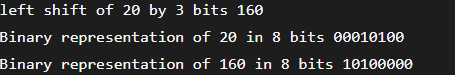
It is used to calculate the bitwise not the operation of the given operand. The 2's complement is returned if the signed integer is passed in the function.

1. arr = np.array([20],dtype = np.uint8)
2. **print**("Binary representation:",np.binary\_repr(20,8))
3. **print**(np.invert(arr))
4. **print**("Binary representation: ", np.binary\_repr(235,8))



It shifts the bits in the binary representation of the operand to the left by the specified position. An equal number of 0s are appended from the right.

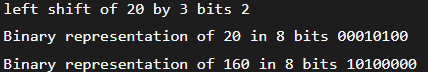
1. **print**("left shift of 20 by 3 bits",np.left\_shift(20, 3))
2. **print**("Binary representation of 20 in 8 bits",np.binary\_repr(20, 8))
3. **print**("Binary representation of 160 in 8 bits",np.binary\_repr(160,8))



## Right Shift Operation

It shifts the bits in the binary representation of the operand to the right by the specified position. An equal number of 0s are appended from the left.

1. **print**("left shift of 20 by 3 bits",np.right\_shift(20, 3))
2. **print**("Binary representation of 20 in 8 bits",np.binary\_repr(20, 8))
3. **print**("Binary representation of 160 in 8 bits",np.binary\_repr(160,8))



# NumPy String Function

| **Function** | **Description** |
| --- | --- |
| add() | It is used to concatenate the corresponding array elements (strings). |
| multiply() | It returns the multiple copies of the specified string, i.e., if a string 'hello' is multiplied by 3 then, a string 'hello hello' is returned. |
| center() | It returns the copy of the string where the original string is centered with the left and right padding filled with the specified number of fill characters. |
| capitalize() | It returns a copy of the original string in which the first letter of the original string is converted to the Upper Case. |
| title() | It returns the title cased version of the string, i.e., the first letter of each word of the string is converted into the upper case. |
| lower() | It returns a copy of the string in which all the letters are converted into the lower case. |
| upper() | It returns a copy of the string in which all the letters are converted into the upper case. |
| split() | It returns a list of words in the string. |
| splitlines() | It returns the list of lines in the string, breaking at line boundaries. |
| strip() | Returns a copy of the string with the leading and trailing white spaces removed. |
| join() | It returns a string which is the concatenation of all the strings specified in the given sequence. |
| replace() | It returns a copy of the string by replacing all occurrences of a particular substring with the specified one. |
| decode() | It is used to decode the specified string element-wise using the specified codec. |
| encode() | It is used to encode the decoded string element-wise. |

### numpy.char.add() method example

1. **print**("Concatenating two string arrays:")
2. **print**(np.char.add(['welcome','Hi'], [' to Javatpoint', ' read python'] ))

//['welcome to Javatpoint' 'Hi read python']

### numpy.char.multiply() method

1. **print**("Printing a string multiple times:")
2. **print**(np.char.multiply("hello ",3)) //hello hello hello

### numpy.char.center() method

1. **print**("Padding the string through left and right with the fill char \*");
2. #np.char.center(string, width, fillchar)
3. **print**(np.char.center("Javatpoint", 20, '\*')) //\*\*\*\*\*Javatpoint\*\*\*\*\*

### numpy.char.capitalize() method

1. **print**("Capitalizing the string using capitalize()...")
2. **print**(np.char.capitalize("welcome to javatpoint")) //Welcome to javatpoint

### numpy.char.title() method

1. **print("Converting string into title cased version...")**
2. **print(np.char.title("welcome to javatpoint")) //Welcome To Javatpoint**

### numpy.char.lower() method

1. **print**("Converting all the characters of the string into lowercase...")
2. **print**(np.char.lower("WELCOME TO JAVATPOINT")) //welcome to javatpoint

### numpy.char.upper() method

1. **print**("Converting all the characters of the string into uppercase...")
2. **print**(np.char.upper("Welcome To Javatpoint")) //WELCOME TO JAVATPOINT

### numpy.char.split() method

1. **print**("Splitting the String word by word..")
2. **print**(np.char.split("Welcome To Java"),sep = " ")//['Welcome', 'To', 'Java']

### numpy.char.splitlines() method

1. **print**("Splitting the String line by line..")
2. **print**(np.char.splitlines("Welcome\nTo\nJavatpoint"))['Welcome', 'To', 'Javatpoint']

### numpy.char.strip() method

1. str = " welcome to javatpoint "
2. **print**("Original String:",str)
3. **print**("Removing the leading and trailing whitespaces from the string")
4. **print**(np.char.strip(str)) //welcome to javatpoint

### numpy.char.join() method

**print**(np.char.join(':','HM')) //H:M

### numpy.char.replace() method

1. str = "Welcome to Javatpoint"
2. **print**("Original String:",str)
3. **print**("Modified Str:",end=" ")
4. **print**(np.char.replace(str, "Welcome to","www.")) //Modified Str: www. Javatpoint

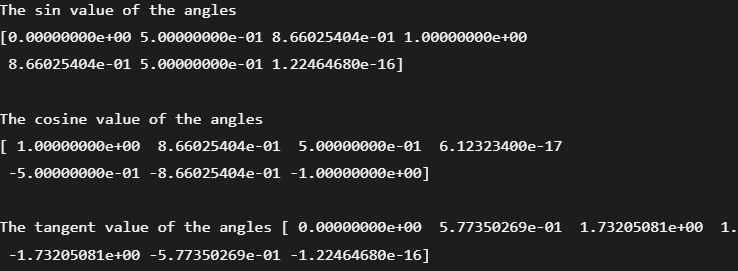
### numpy.char.encode() and decode() method

1. enstr = np.char.encode("welcome to javatpoint", 'cp500')
2. dstr =np.char.decode(enstr, 'cp500')
3. **print**(enstr)
4. **print**(dstr)

# NumPy Mathematical Functions

The mathematical functions include trigonometric functions, arithmetic functions, and functions for handling complex numbers. Let's discuss the mathematical functions.

1. arr = np.array([0, 30, 60, 90, 120, 150, 180])
2. **print**("\nThe sin value of the angles",end = " ")
3. **print**(np.sin(arr \* np.pi/180))
4. **print**("\nThe cosine value of the angles",end = " ")
5. **print**(np.cos(arr \* np.pi/180))
6. **print**("\nThe tangent value of the angles",end = " ")
7. **print**(np.tan(arr \* np.pi/180))



## Rounding Functions

The numpy provides various functions that can be used to truncate the value of a decimal float number rounded to a particular precision of decimal numbers.

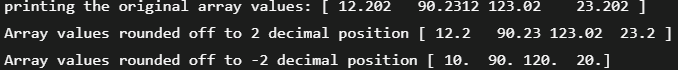
### The numpy.around() function

This function returns a decimal value rounded to a desired position of the decimal.

**Syntax** :- numpy.around(num, decimals)

**Num:-** It is the input number.

**Decimals:** It is the number of decimals which to which the number is to be rounded. The default value is 0. If this value is negative, then the decimal will be moved to the left.



### The numpy.floor() function

This function is used to return the floor value of the input data which is the largest integer not greater than the input value.

1. arr = np.array([12.202, 90.23120, 123.020, 23.202])
2. **print**(np.floor(arr)) //[ 12. 90. 123. 23.]

### The numpy.ceil() function

This function is used to return the ceiling value of the array values which is the smallest integer value greater than the array element.

1. arr = np.array([12.202, 90.23120, 123.020, 23.202])
2. **print**(np.ceil(arr)) //[ 13. 91. 124. 24.]

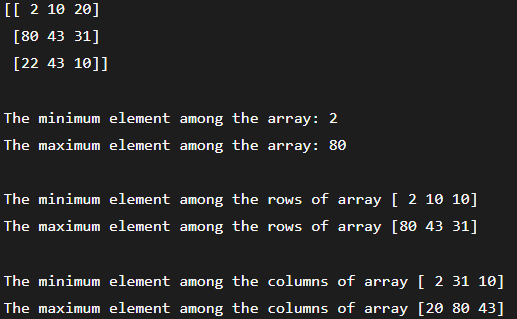
# Numpy statistical functions

It provides many funcs that are used to perform some statistical data analysis.

## Finding minimum & maximum elements from the array

The numpy.amin() and numpy.amax() functions are used to find the minimum and maximum of the array elements along the specified axis respectively.

1. a = np.array([[2,10,20],[80,43,31],[22,43,10]])
2. **print**("\nThe minimum element among the array:",np.amin(a))
3. **print**("The maximum element among the array:",np.amax(a))
4. **print**("\nThe minimum element among the rows of array",np.amin(a,0))
5. **print**("The maximum element among the rows of array",np.amax(a,0))
6. **print**("\nThe minimum element among the columns of array",np.amin(a,1))
7. **print**("The maximum element among the columns of array",np.amax(a,1))



## numpy.ptp() function

The name of the function numpy.ptp() is derived from the name peak-to-peak. It is used to return the range of values along an axis.

1. a = np.array([[2,10,20],[80,43,31],[22,43,10]])
2. **print**("\nptp value along axis 1:",np.ptp(a,1)) //[18 49 33]
3. **print**("ptp value along axis 0:",np.ptp(a,0)) //[78 33 21]

## numpy.percentile() function

**Syntax:** numpy.percentile(input, q, axis)

* **input:** It is the input array.
* **q:** It is the percentile (1-100) which is calculated of the array element.
* **axis:** It is the axis along which the percentile is to be calculated.

1. a = np.array([[2,10,20],[80,43,31],[22,43,10]])
2. **print**("\nPercentile along axis 0",np.percentile(a, 10,0)) //[ 6. 16.6 12. ]
3. **print**("Percentile along axis 1",np.percentile(a, 10, 1)) //[ 3.6 33.4 12.4]

## Calculating median, mean, and average of array items

## The numpy.median() function:

**Median** is defined as the value that is used to separate the higher range of data sample with a lower range of data sample. The function numpy.median() is used to calculate the median of the multi-dimensional or one-dimensional arrays.

## The numpy.mean() function:

The mean can be calculated by adding all the items of the arrays dividing by the number of array elements. We can also mention the axis along which mean can be calculated.

## The numpy.average() function:

The numpy.average() function is used to find the weighted average along the axis of the multi-dimensional arrays where their weights are given in another array.

1. a = np.array([[1,2,3],[4,5,6],[7,8,9]])
2. **print**("\nMedian of array along axis 0:",np.median(a,0)) //[4. 5. 6.]
3. **print**("Mean of array along axis 0:",np.mean(a,0)) //[4. 5. 6.]
4. **print**("Average of array along axis 1:",np.average(a,1)) //[2. 5. 8.]

# NumPy Sorting and Searching

Numpy provides a variety of functions for sorting and searching. There are various sorting algorithms like quicksort, merge sort and heapsort which is implemented using the numpy.sort() function.

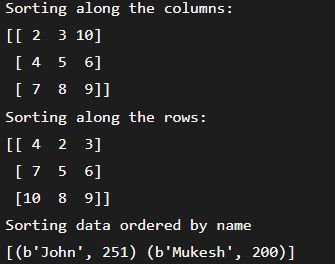
The kind of the sorting algorithm to be used in the sort operation must be mentioned in the function call.

| **SN** | **Algorithm** | **Worst case complexity** |
| --- | --- | --- |
| 1 | Quick Sort | O (n ^ 2) |
| 2 | Merge Sort | O (n \* log(n)) |
| 3 | Heap Sort | O (n \* log(n)) |

**Syntax**: numpy.sort(a, axis, kind, order)

| **Parameter** | **Description** |
| --- | --- |
| input | It represents the input array which is to be sorted. |
| axis | It represents the axis along which the array is to be sorted. If the axis is not mentioned, then the sorting is done along the last available axis. |
| kind | It represents the type of sorting algorithm which is to be used while sorting. The default is quick sort. |
| order | It represents the filed according to which the array is to be sorted in the case if the array contains the fields. |

1. a = np.array([[10,2,3],[4,5,6],[7,8,9]])
2. **print**(np.sort(a)) //Sorting along the columns:
3. **print**(np.sort(a, 0)) //Sorting along the rows:
4. data\_type = np.dtype([('name', 'S10'),('marks',int)])
5. arr = np.array([('Mukesh',200),('John',251)],dtype = data\_type)
6. **print**(np.sort(arr,order = 'name')) //Sorting data ordered by name



### numpy.argsort() function

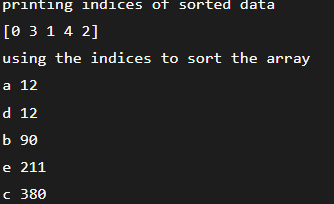
This function is used to perform an indirect sort on an input array that is, it returns an array of indices of data which is used to construct the array of sorted data.

1. a = np.array([90, 29, 89, 12])
2. **print**("Original array:\n",a) //[90 29 89 12]
3. sort\_ind = np.argsort(a)
4. **print**("Printing indices of sorted data\n",sort\_ind) //[3 1 2 0]
5. sort\_a = a[sort\_ind]
6. **for** i **in** sort\_ind: //printing sorted array
7. **print**(a[i],end = " ") //12 29 89 90

### numpy.lexsort() function

This function is used to sort the array using sequence of keys indirectly. This function performs similarly to the numpy.argsort() which returns array of indices of sorted data.

1. a = np.array(['a','b','c','d','e'])
2. b = np.array([12, 90, 380, 12, 211])
3. ind = np.lexsort((a,b))
4. **print**(ind) //printing indices of sorted data
5. **for** i **in** ind: //using the indices to sort the array
6. **print**(a[i],b[i])



### numpy.nonzero() function

This function is used to find the location of the non-zero elements from the array.

1. b = np.array([12, 90, 380, 12, 211])
2. **print**("printing original array",b) //[ 12 90 380 12 211]
3. **print**("printing location of the non-zero elements")
4. **print**(b.nonzero()) //(array([0, 1, 2, 3, 4]),)

### numpy.where() function

This function is used to return the indices of all the elements which satisfies a particular condition.

1. b = np.array([12, 90, 380, 12, 211])
2. **print**(np.where(b>12)) //(array([1, 2, 4]),)
3. c = np.array([[20, 24],[21, 23]])
4. **print**(np.where(c>20)) //(array([0, 1, 1]), array([1, 0, 1]))

# NumPy Copies and Views

The copy of an input array is physically stored at some other location and the content stored at that particular location is returned which is the copy of the input array whereas the different view of the same memory location is returned in the case of view.

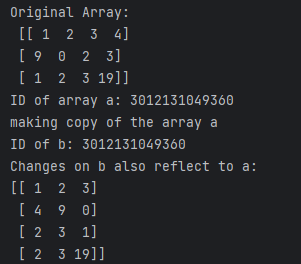
## 

## Array Assignment

The assignment of a numpy array to another array doesn't make the direct copy of the original array, instead, it makes another array with the same content and same id. It represents the reference to the original array. Changes made on this reference are also reflected in the original array.

The id() function returns the universal identifier of the array similar to the pointer in C.

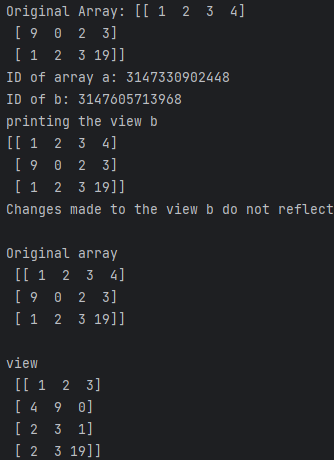
1. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
2. **print**("Original Array:\n",a)
3. **print**("\nID of array a:",id(a))
4. b = a
5. **print**("\nID of b:",id(b)) //making copy of the array a
6. b.shape = 4,3;
7. **print**(a) //Changes on b also reflect to a:



### ndarray.view() method

The ndarray.view() method returns the new array object which contains the same content as the original array does. Since it is a new array object, changes made on this object do not reflect the original array.

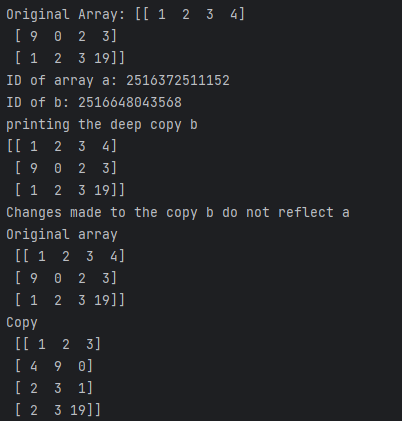
1. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
2. **print**("Original Array:\n",a)
3. **print**("\nID of array a:",id(a))
4. b = a.view()
5. **print**("\nID of b:",id(b))
6. **print**("\nprinting the view b") //printing the view b
7. **print**(b)
8. b.shape = 4,3;
9. **print**("\nOriginal array \n",a) //Changes made to the view b do not reflect a
10. **print**("\nview\n",b)



### ndarray.copy() method

It returns the deep copy of the original array which doesn't share any memory with the original array. The modification made to the deep copy of the original array doesn't reflect the original array.

1. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
2. **print**("Original Array:\n",a)
3. **print**("\nID of array a:",id(a))
4. b = a.copy()
5. **print**("\nID of b:",id(b))
6. **print**(b) //printing the deep copy b
7. b.shape = 4,3;
8. **print**("\nOriginal array \n",a) //Changes made to the copy b do not reflect a
9. **print**("\nCopy\n",b)



# NumPy Matrix Library

NumPy contains a matrix library, i.e. numpy.matlib which is used to configure matrices instead of ndarray objects.

### numpy.matlib.empty() function

This function is used to return a new matrix with the uninitialized entries.

**Syntax**: numpy.matlib.empty(shape, dtype, order)

* shape: It is the tuple defining the shape of the matrix.
* dtype: It is the data type of the matrix.
* order: It is the insertion order of the matrix, i.e. C or F.

1. **print**(numpy.matlib.empty((3,3)))

[[6.90262230e-310 6.90262230e-310 6.90262304e-310]

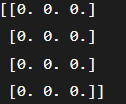
[6.90262304e-310 6.90261674e-310 6.90261552e-310]

[6.90261326e-310 6.90262311e-310 3.95252517e-322]]

### numpy.matlib.zeros() function

This function is used to create the matrix where the entries are initialized to zero.

1. **print**(numpy.matlib.zeros((4,3)))



### numpy.matlib.ones() function

This function returns a matrix with all the elements initialized to 1.

**print**(numpy.matlib.ones((2,2)))



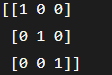
### numpy.matlib.eye() function

This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere.

**Syntax**: numpy.matlib.eye(n, m, k, dtype)

* n: It represents the number of rows in the resulting matrix.
* m: It represents the number of columns, defaults to n.
* k: It is the index of diagonal.
* dtype: It is the data type of the output

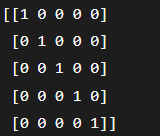
1. **print**(numpy.matlib.eye(n = 3, M = 3, k = 0, dtype = int))



### numpy.matlib.identity() function

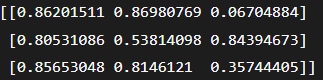
This function is used to return an identity matrix of the given size. An identity matrix is the one with diagonal elements initializes to 1 and all other elements to zero.

1. **print**(numpy.matlib.identity(5, dtype = int))



### numpy.matlib.rand() function

This function is used to generate a matrix where all the entries are initialized with random values.



# NumPy Linear Algebra

Numpy provides the following functions to perform the different algebraic calculations on the input data.

| **SN** | **Function** | **Definition** |
| --- | --- | --- |
| 1 | dot() | It is used to calculate the dot product of two arrays. |
| 2 | vdot() | It is used to calculate the dot product of two vectors. |
| 3 | inner() | It is used to calculate the inner product of two arrays. |
| 4 | matmul() | It is used to calculate the matrix multiplication of two arrays. |
| 5 | det() | It is used to calculate the determinant of a matrix. |
| 6 | solve() | It is used to solve the linear matrix equation. |
| 7 | inv() | It is used to calculate the multiplicative inverse of the matrix. |

## numpy.dot() function

This function is used to return the dot product of the two matrices. It is similar to the matrix multiplication.

1. a = np.array([[100,200],[23,12]])
2. b = np.array([[10,20],[12,21]])
3. dot = np.dot(a,b)
4. **print**(dot) The dot product is calculated as: [100 \* 10 + 200 \* 12, 100 \* 20 + 200 \* 21] [23\*10+12\*12, 23\*20 + 12\*21]



## numpy.vdot() function

This function is used to calculate the dot product of two vectors. It can be defined as the sum of the product of corresponding elements of multi-dimensional arrays.

1. a = np.array([[100,200],[23,12]])
2. b = np.array([[10,20],[12,21]])
3. vdot = np.vdot(a,b)
4. **print**(vdot) //5528 ##np.vdot(a,b) = 100 \*10 + 200 \* 20 + 23 \* 12 + 12 \* 21 =5528

## numpy.inner() function

This function returns the sum of the product of inner elements of one-dimensional array. For n-dimensional arrays, it returns the sum of product of elements over the last axis.

1. a = np.array([1,2,3,4,5,6])
2. b = np.array([23,23,12,2,1,2])
3. inner = np.inner(a,b)
4. **print**(inner) //130

## numpy.matmul() function

It is used to return the multiplication of the two matrices. It gives an error if the shape of both matrices is not aligned for multiplication.

1. a = np.array([[1,2,3],[4,5,6],[7,8,9]])
2. b = np.array([[23,23,12],[2,1,2],[7,8,9]])
3. mul = np.matmul(a,b)
4. **print**(mul)

## numpy determinant

The determinant of the matrix can be calculated using the diagonal elements. The determinant of following 2 X 2 matrix **[[A,B][C ,D]]**

can be calculated as AD - BC.

The numpy.linalg.det() function is used to calculate the determinant of the matrix.

1. a = np.array([[1,2],[3,4]])
2. **print**(np.linalg.det(a)) //-2.0000000000000004

## numpy.linalg.solve() function

This function is used to solve a quadratic equation where values can be given in the form of the matrix.

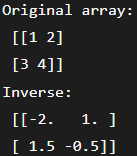
1. a = np.array([[1,2],[3,4]])
2. b = np.array([[1,2],[3,4]])
3. **print**(np.linalg.solve(a, b))



## numpy.linalg.inv() function

This function is used to calculate the multiplicative inverse of the input matrix.

1. a = np.array([[1,2],[3,4]])
2. **print**("Original array:\n",a)
3. b = np.linalg.inv(a)
4. **print**("Inverse:\n",b)



# NumPy Matrix Multiplication in Python

Multiplication of matrix is an operation which produces a single matrix by taking two matrices as input and multiplying rows of the first matrix to the column of the second matrix. Note that we have to ensure that the number of rows in the first matrix should be equal to the number of columns in the second matrix.

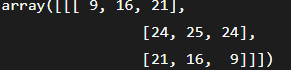
In Python, the process of matrix multiplication using NumPy is known as **vectorization**. The main objective of vectorization is to remove or reduce the **for loops** which we were using explicitly. By reducing 'for' loops from programs gives faster computation. The build-in package NumPy is used for manipulation and array-processing.

These are three methods through which we can perform numpy matrix multiplication.

1. First is the use of multiply() function, which perform element-wise multiplication of the matrix.
2. Second is the use of matmul() function, which performs the matrix product of two arrays.
3. Last is the use of the dot() function, which performs dot product of two arrays.

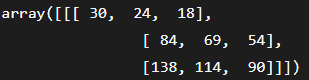
### Example 1: Element-wise matrix multiplication

1. array1=np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin=3)
2. array2=np.array([[9,8,7],[6,5,4],[3,2,1]],ndmin=3)
3. result=np.multiply(array1,array2)



### Example 2: Matrix product

1. array1=np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin=3)
2. array2=np.array([[9,8,7],[6,5,4],[3,2,1]],ndmin=3)
3. result=np.matmul(array1,array2)

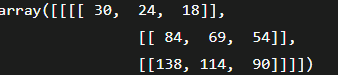


### Example 3: Dot product

These are the following specifications for numpy.dot:

* When both a and b are 1-D (one dimensional) arrays-> Inner product of two vectors (without complex conjugation)
* When both a and b are 2-D (two dimensional) arrays -> Matrix multiplication
* When either a or b is 0-D (also known as a scalar) -> Multiply by using numpy.multiply(a, b) or a \* b.
* When a is an N-D array & b is a 1-D array -> Sum product over last axis of a and b.
* When a is an N-D array and b is an M-D array provided that M>=2 -> Sum product over the last axis of a and the second-to-last axis of b:  
  **Also, dot(a, b)[i,j,k,m] = sum(a[i,j,:] \* b[k,:,m])**

1. array1=np.array([[1,2,3],[4,5,6],[7,8,9]],ndmin=3)
2. array2=np.array([[9,8,7],[6,5,4],[3,2,1]],ndmin=3)
3. result=np.dot(array1,array2)



# numpy.array() in Python

The homogeneous multidimensional array is the main object of **NumPy**. It is basically a table of elements which are all of the same type and indexed by a tuple of positive integers. The dimensions are called axis in NumPy.

The NumPy's array class is known as **ndarray** or **alias array**. The numpy.array is not the same as the standard Python library class **array.array**. The array.array handles only one-dimensional arrays and provides less functionality.

### Syntax:numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

**1) object: array\_like**

Any object, which exposes an array interface whose \_\_array\_\_ method returns any nested sequence or an array.

**2) dtype : optional data-type**

This parameter is used to define the desired parameter for the array element. If we do not define the data type, then it will determine the type as the minimum type which will require to hold the object in sequence. This parameter is used only for upcasting array.

**3) copy: bool(optional)**

If we set copy equals to true, the object is copied else the copy will be made when an

object is a nested sequence, or a copy is needed to satisfy any of the other

requirements such as dtype, order, etc.

**4) order : {'K', 'A', 'C', 'F'}, optional**

The order parameter specifies the memory layout of the array. When the object is not an array, the newly created array will be in C order (row head or row-major) unless 'F' is specified. When F is specified, it will be in Fortran order (column head or column-major). When the object is an array, it holds the following order.

| **Order** | **no copy** | **copy=True** |
| --- | --- | --- |
| 'K' | Unchanged | F and C order preserved. |
| 'A' | Unchanged | When the input is F and not C then F order otherwise C order |
| 'C' | C order | C order |
| 'F' | F order | F order |

When copy=False or the copy is made for the other reason, the result will be the same as copy= True with some exceptions for A. The default order is 'K'.

**5) subok : bool(optional)**

When subok=True, then sub-classes will pass-through; otherwise, the returned array will force to be a base-class array (default).

**6) ndmin : int(optional)**

This parameter specifies the minimum number of dimensions which the resulting array should have. Users can be prepended to the shape as needed to meet this requirement.

### Returns: The numpy.array() method returns an ndarray. The ndarray is an array object which satisfies the specified requirements.

1. arr=np.array([1,2,3]) //array([1, 2, 3])
2. floatArr=np.array([1,2.,3.]) //array([1., 2., 3.])

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



### Example 4: Minimum dimensions: 2

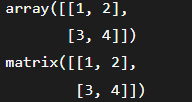
1. arr=np.array([1,2.,3.],ndmin=2) //array([[1., 2., 3.]])

### Example 5: Type provided

1. arr=np.array([12,45.,3.],dtype=complex) //array([12.+0.j, 45.+0.j, 3.+0.j])

### Example 6: Creating an array from sub-classes

1. arr=np.array(np.mat('1 2;3 4'))
2. arr=np.array(np.mat('1 2;3 4'),subok=True)



# numpy.concatenate() in Python

The concatenate() function is a function from the NumPy package. This function essentially combines NumPy arrays together. This function is basically used for joining two or more arrays of the same shape along a specified axis. There are the following things which are essential to keep in mind:

1. NumPy's concatenate() is not like a traditional database join. It is like stacking NumPy arrays.
2. This function can operate both vertically and horizontally. This means we can concatenate arrays together horizontally or vertically.

### Syntax: numpy.concatenate((a1, a2, ...), axis)

**1) (a1, a2, ...)**

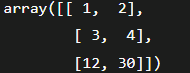
This parameter defines the sequence of arrays. Here, a1, a2, a3 ... are the arrays which have the same shape, except in the dimension corresponding to the axis.

**2) axis : int(optional)**

This parameter defines axis along which array will be joined. By default, its value is 0.

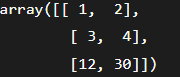
Result: It will return a ndarray containing the elements of both the arrays.

1. x=np.array([[1,2],[3,4]])
2. y=np.array([[12,30]])
3. z=np.concatenate((x,y))



### Example 2: numpy.concatenate() with axis=0

1. x=np.array([[1,2],[3,4]])
2. y=np.array([[12,30]])
3. z=np.concatenate((x,y), axis=0)



### Example 3: numpy.concatenate() with axis=1

1. x=np.array([[1,2],[3,4]])
2. y=np.array([[12,30]])
3. z=np.concatenate((x,y.T), axis=1)



### Example 4: numpy.concatenate() with axis=None

1. x=np.array([[1,2],[3,4]])
2. y=np.array([[12,30]])
3. z=np.concatenate((x,y), axis=None) //array([ 1, 2, 3, 4, 12, 30])

# numpy.append() in Python

The numpy.append() function is available in NumPy package. As the name suggests, append means adding something. The numpy.append() function is used to add or append new values to an existing numpy array. This function adds the new values at the end of the array.

The numpy append() function is used to merge two arrays. It returns a new array, and the original array remains unchanged.

### Syntax: numpy.append(arr, values, axis=None)

**1) arr: array\_like**

This is a ndarray. The new values are appended to a copy of this array. This parameter is required and plays an important role in numpy.append() function.

**2) values: array\_like**

This parameter defines the values which are appended to a copy of a ndarray. One thing is to be noticed here that these values must be of the correct shape as the original ndarray, excluding the axis. If the axis is not defined, then the values can be in any shape and will flatten before use.

**3) axis: int(optional)**

This parameter defines the axis along which values are appended. When the axis is not given to them, both ndarray and values are flattened before use.

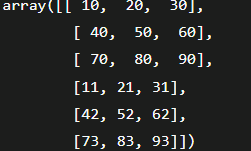
### Returns: This function returns a copy of ndarray with values appended to the axis.

1. a=np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
2. b=np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
3. c=np.append(a,b)

//array([ 10, 20, 30, 40, 50, 60, 70, 80, 90,11,21, 31, 42, 52, 62,73,83,93])

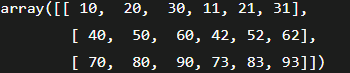
### Example 2: np.append({a1,a2,...}, axis=0)

1. a=np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
2. b=np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
3. c=np.append(a,b,axis=0)



### Example 3: np.append({a1,a2,...}, axis=1)

1. a=np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
2. b=np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
3. c=np.append(a,b,axis=1)



# numpy.reshape() in Python

The numpy.reshape() function is available in NumPy package. As the name suggests, reshape means 'changes in shape'. The numpy.reshape() function helps us to get a new shape to an array without changing its data.

Sometimes, we need to reshape the data from wide to long. So in this situation, we have to reshape the array using reshape() function.

### Syntax: numpy.reshape(arr, new\_shape, order='C')

**1) arr: array\_like**

This is a ndarray. This is the source array which we want to reshape. This parameter is essential and plays a vital role in numpy.reshape() function.

**2) new\_shape: int or tuple of ints**

The shape in which we want to convert our original array should be compatible with the original array. If an integer, the result will be a 1-D array of that length. One shape dimension can be -1. Here, the value is approximated by the length of the array and the remaining dimensions.

**3) order: {'C', 'F', 'A'}, optional**

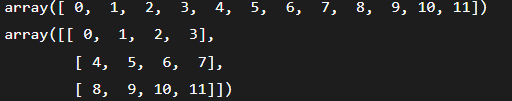
These index orders are used to read the elements of source array and place the elements into the reshaped array using this index order.

1. The index order 'C' means to read/write the elements which are using a C-like index order where the last axis index is changing fastest, back to the first axis index changing slowest.
2. The index order 'F' means to read/write the elements which are using the Fortran-like index order, where the last axis index changing slowest and the first axis index changing fastest.
3. The 'C' and 'F' order take no amount of the memory layout of the underlying array and only refer to the order of indexing.
4. The index order 'A' means to read/write the elements in Fortran-like index order, when arr is contiguous in memory, otherwise use C-like order.

### Returns:This function returns a ndarray. It is a new view object if possible; otherwise, it will be a copy. There is no guarantee of the memory layout of the returned array.

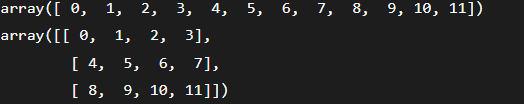
### Example 1: C-like index ordering

1. x=np.arange(12)
2. y=np.reshape(x, (4,3))



### Example 2: Equivalent to C ravel then C reshape

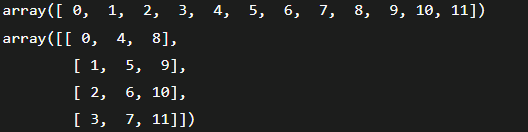
1. x=np.arange(12)
2. y=np.reshape(np.ravel(x),(3,4))



The ravel() function is used for creating a contiguous flattened array. A one-dimensional array that contains elements of input, is returned.A copy is made only when it is needed.

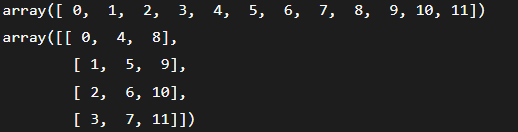
### Example 3: Fortran-like index ordering

1. x=np.arange(12)
2. y=np.reshape(x, (4, 3), order='F')

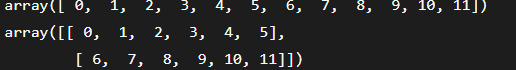


### Example 4: Fortran-like index ordering

1. x=np.arange(12)
2. y=np.reshape(np.ravel(x, order='F'), (4, 3), order='F')



### Example 5: The unspecified value is inferred to be 2

1. x=np.arange(12)
2. y=np.reshape(x, (2, -1)) 

* We have declared the variable 'y' and assigned the returned value of the np.reshape() function.
* We have passed the array 'x' and the shape (unspecified value) in the function.

# numpy.sum() in Python

This function is used to compute the sum of all elements, the sum of each row, and the sum of each column of a given array.

Essentially, this sum ups the elements of an array, takes the elements within a ndarray, and adds them together. It is also possible to add rows and column elements of an array. The output will be in the form of an array object.

### Syntax: numpy.sum(arr, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>)

**1) arr: array\_like**

This is a ndarray. This is the source array whose elements we want to sum. This parameter is essential and plays a vital role in numpy.sum() function.

**2) axis: int or None or tuple of ints(optional)**

This parameter defines the axis along which a sum is performed. The default axis is None, which will sum all the elements of the array. When the axis is negative, it counts from the last to the first axis. In version 1.7.0, a sum is performed on all axis specified in the tuple instead of a single axis or all axis as before when an axis is a tuple of ints.

**3) dtype: dtype(optional)**

This parameter defines the type of the accumulator and the returned array in which the elements are summed. By default, the dtype of arr is used unless arr has an integer dtype of less precision than the default platform integer. In such a case, when arr is signed, then the platform integer is used, and when arr is unsigned, then an unsigned integer of the same precision as the platform integer is used.

**4) out: ndarray(optional)**

This parameter defines the alternative output array in which the result will be placed. This resulting array must have the same shape as the expected output. The type of output values will be cast, when necessary.

**5) keepdims: bool(option)**

This parameter defines a Boolean value. When this parameter is set to True, the axis which is reduced is left in the result as dimensions with size one. With the help of this option, the result will be broadcast correctly against the input array. The keepdims will not be passed to the sum method of sub-classes of a ndarray, when the default value is passed, but not in case of non-default value. If the sub-class method does not implement keepdims, then any exception can be raised.

**6) initial: scalar**

This parameter defines the starting value for the sum.

### Returns: This function returns an array of the same shape as arr with the specified axis removed. When arr is a 0-d array, or when the axis is None, a scalar is returned. A reference to **out** is returned, when an array output is specified.

1. a=np.array([0.4,0.5])
2. b=np.sum(a) **//0.9**

### Example 2: **a=np.array([0.4,0.5,0.9,6.1])**

1. **x=np.sum(a, dtype=np.int32) //6**

### Example 3: **a=np.array([[1,4],[3,5]])**

1. **b=np.sum(a) //13**

### Example 4: **a=np.array([[1,4],[3,5]])**

1. **b=np.sum(a,axis=0) //array([4, 9])**

### Example 5: **a=np.array([[1,4],[3,5]])**

1. **b=np.sum(a,axis=1) //array([5, 8])**

### Example 6:

1. **b=np.sum([15], initial=8) //23**

# numpy.random() in Python

This module contains the functions which are used for generating random numbers. This module contains some simple random data generation methods, some permutation and distribution functions, and random generator functions.

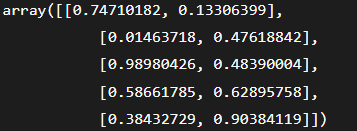
## A)Simple random data

There are the following functions of simple random data:

**1) p.random.rand(d0, d1, ..., dn)**

This function of random module is used to generate random numbers or values in a given shape.

a=np.random.rand(5,2)

****

**2) np.random.randn(d0, d1, ..., dn)**

This function of random module return a sample from "standard normal" distribution.

a=np.random.randn(2,2)

****

**3) np.random.randint(low[, high, size, dtype])**

This function of random module is used to generate random integers from inclusive(low) to exclusive(high).

a=np.random.randint(3, size=10) //array([1, 1, 1, 2, 0, 0, 0, 0, 0, 0])

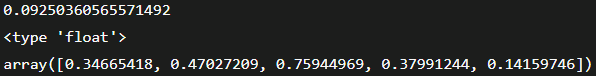
**4) np.random.random\_integers(low[, high, size])**

This function of random module is used to generate random integers number of type np.int between low and high.

**5) np.random.random\_sample([size])**

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

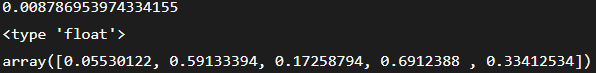
1. a=np.random.random\_sample()
2. b=type(np.random.random\_sample())
3. c=np.random.random\_sample((5,))



**6) np.random.random([size])**

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

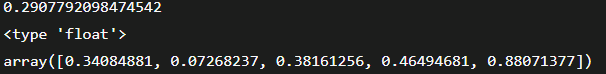
1. a=np.random.random()
2. b=type(np.random.random())
3. c=np.random.random((5,))



**7) np.random.ranf([size])**

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

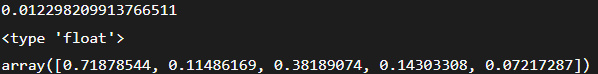
1. a=np.random.ranf()
2. b=type(np.random.ranf())
3. c=np.random.ranf((5,))



**8) np.random.sample([size])**

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

1. a=np.random.sample()
2. b=type(np.random.sample())
3. c=np.random.sample((5,))



**9) np.random.choice(a[, size, replace, p])**

This func of random module is used to generate random sample from a given 1-D array.

1. a=np.random.choice(5,3)
2. b=np.random.choice(5,3, p=[0.2, 0.1, 0.4, 0.2, 0.1])



**10) np.random.bytes(length)**

This function of random module is used to generate random bytes.

a=np.random.bytes(7) //'nQ\x08\x83\xf9\xde\x8a'

## B)Permutations

There are the following functions of permutations:

**1) np.random.shuffle()**

This function is used for modifying a sequence in-place by shuffling its contents.

1. a=np.arange(12)
2. np.random.shuffle(a)



**2) np.random.permutation()**

This function permute a sequence randomly or return a permuted range.

a=np.random.permutation(12) //array([ 8, 7, 3, 11, 6, 0, 9, 10, 2, 5, 4, 1])

## C)Distributions

There are the following functions of permutations:

**2) binomial(n, p[, size])**

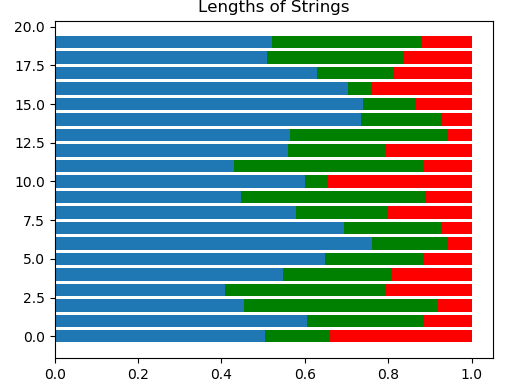
This function is used to draw sample from a binomial distribution.

1. n, p = 10, .6
2. s1= np.random.binomial(n, p, 10) //array([6, 7, 7, 9, 3, 7, 8, 6, 6, 4])

**4) dirichlet(alpha[, size])**

This function is used to draw a sample from the Dirichlet distribution.

1. s1 = np.random.dirichlet((10, 5, 3), 20).transpose()
2. plt.barh(range(20), s1[0])
3. plt.barh(range(20), s1[1], left=s1[0], color='g')
4. plt.barh(range(20), s1[2], left=s1[0]+s1[1], color='r')
5. plt.title("Lengths of Strings")
6. plt.show()



**6) f(dfnum, dfden[, size])**

This function is used to draw sample from an F distribution.

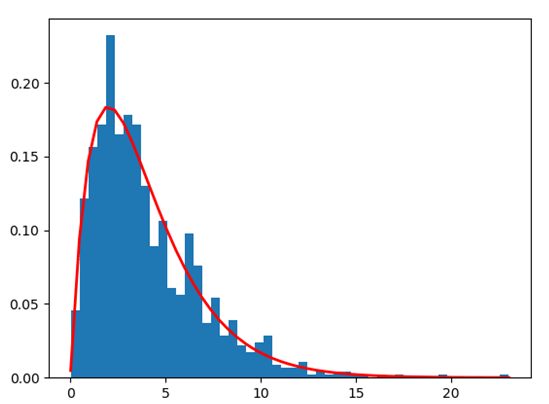
1. dfno= 1.
2. dfden = 48.
3. s1 = np.random.f(dfno, dfden, 10)
4. np.sort(s1)



**7) gamma(shape[, scale, size])**

This function is used to draw sample from a Gamma distribution.

1. shape, scale = 2., 2.
2. s1 = np.random.gamma(shape, scale, 1000)
3. **import** matplotlib.pyplot as plt
4. **import** scipy.special as spss
5. count, bins, ignored = plt.hist(s1, 50, density=True)
6. a = bins\*\*(shape-1)\*(np.exp(-bins/scale) /
7. (spss.gamma(shape)\*scale\*\*shape))
8. plt.plot(bins, a, linewidth=2, color='r')
9. plt.show()



**8) geometric(p[, size])**

This function is used to draw sample from a geometric distribution.

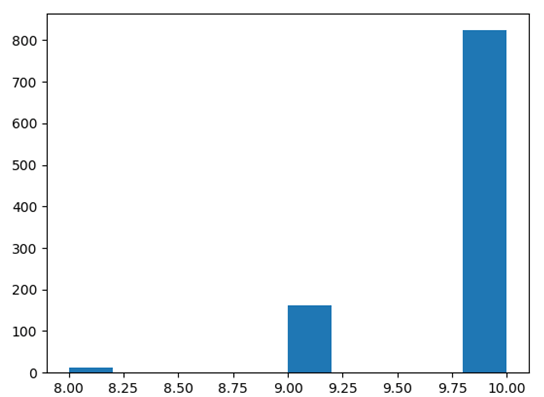
1. a = np.random.geometric(p=0.35, size=10000)
2. (a == 1).sum() / 1000 //3.4

**10) hypergeometric(ngood, nbad, nsample[, size])**

This function is used to draw sample from a Hypergeometric distribution.

1. **good, bad, samp = 100, 2, 10**
2. **s1 = np.random.hypergeometric(good, bad, samp, 1000)**
3. **plt.hist(s1)**
4. **plt.show()**

**//(array([ 13., 0., 0., 0., 0., 163., 0., 0., 0., 824.]), array([ 8. , 8.2, 8.4, 8.6, 8.8, 9. , 9.2, 9.4, 9.6, 9.8, 10. ]), <a list of 10 Patch objects>)**

****

# numpy.zeros() in Python

The numpy.zeros() function is one of the most significant functions which is used in ML programs widely. This function is used to generate an array containing zeros.

The numpy.zeros() function provide a new array of given shape and type, which is filled with zeros.

### Syntax: numpy.zeros(shape, dtype=**float**, order='C'

**shape: int or tuple of ints**

This parameter is used to define the dimensions of the array. This parameter is used for the shape in which we want to create an array, such as (3,2) or 2.

**dtype: data-type(optional)**

This parameter is used to define the desired data-type for the array. By default, the data-type is numpy.float64. This parameter is not essential for defining.

**order: {'C','F'}(optional)**

This parameter is used to define the order in which we want to store data in memory either row-major(C-style) or column-major(Fortran-style)

### Return: This function returns a ndarray. The output array is the array with specified shape, dtype, order, and contains zeros.

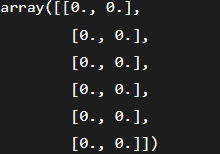
### Example 1: numpy.zeros() without dtype and order

1. a=np.zeros(6) **//array([0., 0., 0., 0., 0., 0.])**

### Example 2: numpy.zeros() without order

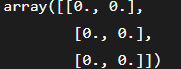
1. a=np.zeros((6,), dtype=**int**) //array([0, 0, 0, 0, 0, 0])

### Example 3: numpy.zeros() with shape

1. a=np.zeros((6,2))
2. 

### Example 4: numpy.zeros() with the shape

1. s1=(3,2)
2. a=np.zeros(s1)



### Example 5: numpy.zeros() with custom dtype

1. a=np.zeros((3,), dtype=[('x', 'i4'), ('y', 'i4')]) //array([(0, 0), (0, 0), (0, 0)], dtype=[('x', '<i4'), ('y', '<i4')])

# numpy.log() in Python

The numpy.log() is a mathematical function that is used to calculate the natural logarithm of x(x belongs to all the input array elements). It is the inverse of the exponential function as well as an element-wise natural logarithm. The natural logarithm log is the reverse of the exponential function, so that log(exp(x))=x. The logarithm in base e is the natural logarithm.

### Syntax: numpy.log(x, /, out=None, \*, where=True, casting='same\_kind', order='K', dtype=None, subok=True[, signature, extobj]) = <ufunc 'log'>

**x: array\_like** : This parameter defines the input value for the numpy.log() function.

**out: ndarray, None, or tuple of ndarray and None(optional)**

This parameter is used to define the location in which the result is stored. If we define this parameter, it must have a shape similar to the input broadcast; otherwise, a freshly-allocated array is returned. A tuple has a length equal to the number of outputs.

**where: array\_like(optional)**

It is a condition that is broadcast over the input. At this location, where the condition is True, the out array will be set to the ufunc(universal function) result; otherwise, it will retain its original value.

**casting: {'no','equiv','safe','same\_kind','unsafe'}(optional)**This parameter controls the kind of data casting that may occur. The 'no' means the data types should not be cast at all.

The 'equiv' means only byte-order changes are allowed. The 'safe' means the only cast, which can allow the preserved value. The 'same\_kind' means only safe casts or casts within a kind. The 'unsafe' means any data conversions may be done.

**order: {'K', 'C', 'F', 'A'}(optional)**

This parameter specifies the calculation iteration order/ memory layout of the output array. By default, the order will be K. The order 'C' means the output should be C-contiguous. The order 'F' means F-contiguous, and 'A' means F-contiguous if the inputs are F-contiguous and if inputs are in C-contiguous, then 'A' means C-contiguous. 'K' means to match the element ordering of the inputs(as closely as possible).

**dtype: data-type(optional):** It overrides the dtype of the calculation and output arrays.

**subok: bool(optional)**

By default, this parameter is set to true. If we set it to false, the output will always be a strict array, not a subtype.

**signature**

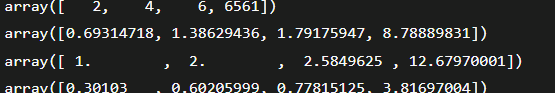
This argument allows us to provide a specific signature to the 1-d loop 'for', used in the underlying calculation.

**extobj**

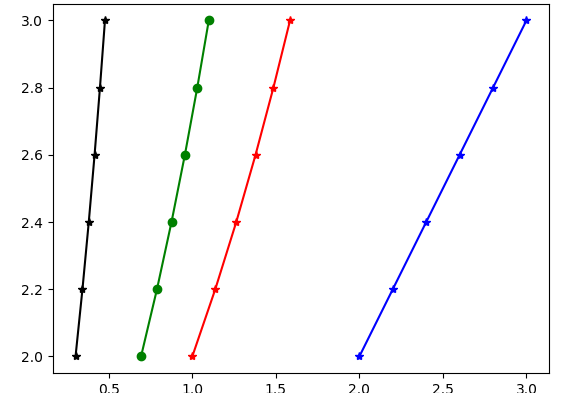
This parameter is a list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error callback function.

### Returns: This function returns a ndarray that contains the natural logarithmic value of x, which belongs to all elements of the input array.

1. a=np.array([2, 4, 6, 3\*\*8])
2. b=np.log(a)
3. c=np.log2(a)
4. d=np.log10(a)



1. arr = [2, 2.2, 2.4, 2.6,2.8, 3]
2. result1=np.log(arr)
3. result2=np.log2(arr)
4. result3=np.log10(arr)
5. plt.plot(arr,arr, color='blue', marker="\*")
6. plt.plot(result1,arr, color='green', marker="o")
7. plt.plot(result2,arr, color='red', marker="\*")
8. plt.plot(result3,arr, color='black', marker="\*")
9. plt.show()



1. x=np.log([2, np.e, np.e\*\*3, 0])



# numpy.where() in Python

The NumPy module provides a function numpy.where() for selecting elements based on a condition. It returns elements chosen from a or b depending on the condition.

ex., if all arguments -> condition, a & b are passed in numpy.where() then it will return elements selected from a & b depending on values in bool array yielded by condition.

If only the condition is provided, this function is a shorthand to the function np.asarray (condition).nonzero(). Although nonzero should be preferred directly, as it behaves correctly for subclasses.

### Syntax:numpy.where(condition[, x, y])

**condition: array\_like, bool:** If this parameter set to True, yield x otherwise yield y.

**x, y: array\_like:**

This parameter defines the values from which to choose. The x, y, and condition need to be broadcastable to some shape.

### Returns: This function returns the array with elements from x where the condition is True and elements from y elsewhere.

1. a=np.arange(12)
2. b=np.where(a<6,a,5\*a) //array([ 0, 1, 2, 3, 4, 5, 30, 35, 40, 45, 50, 55])

In the output, the values ranging from 0 to 5 remain the same as per the condition, and the other values have been multiplied with 5.

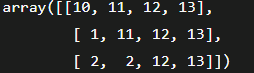
### Example 2: For multidimensional array

1. a=np.arange(12)
2. b=np.where([[True, False], [True, True]],[[1, 2], [3, 4]],[[9, 8], [7, 6]])



### Example 3: Broadcasting x, y, and condition

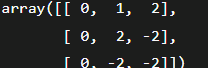
1. x, y = np.ogrid[:3, :4]
2. a=np.where(x > y, x, 10 + y)



In the output, the x value has been compared to y value if it satisfied the condition, then it will be printed x value otherwise, it will print y value, which has passed as an argument in the where() function.

### Example 4: Broadcasting specific value

1. x=np.array([[0,1,2],[0,2,5],[0,4,8]])
2. y=np.where(x<4,x,-2)



# numpy.argsort() in Python

NumPy module provides a function argsort(), returns indices which would sort an array.

The NumPy module provides a function for performing an indirect sort along with the given axis with the help of the algorithm specified by the keyword. This function returns an array of indices of the same shape as 'a', which would sort the array.

### Syntax: numpy.argsort(a, axis=-1, kind=None, order=None)

**a: array\_like:** This parameter defines the source array which we want to sort.

**axis: int or None(optional)**

This parameter defines the axis along which the sorting is performed. By default, the axis is -1. If we set this parameter to None, the flattened array is used.

**kind: {'quicksort','mergesort','heapsort','stable'}(optional)**

This parameter defines the sorting algorithm. By default, the algorithm is **quicksort**. Both **mergesort** and **stable** are using time sort under the covers. The actual implementation will vary with the data type. The **mergesort** option is retained for backward compatibility.

**order: str or list of str(optional)**

If 'a' is an array with defined fields, this argument specifies which fields to compare first, second, etc. The single field can be specified as a string, and not all fields need to be specified. But unspecified fields will still use, in the order in which they come up in the dtype, to break the ties.

### Returns: index\_array: ndarray, int

This function returns an array of indices which sort 'a' along with the specified axis. If 'a' is 1-D, a[index\_array] yields a sorted 'a'. More generally, **np.take\_along\_axis(arr1, index\_array, axis=axis)** always yields the sorted 'a', irrespective of dimensionality.

### Example 1: np.argsort()

1. a=np.array([456,11,63])
2. b=np.argsort(a)

In the output, a ndarray has been shown that contains the indices(indicate the position of the element for the sorted array) and dtype.



### Example 2: For 2-D array( sorts along first axis (down))

1. a = np.array([[0, 5], [3, 2]])
2. indices = np.argsort(a, axis=0)



### Example 4: For 2-D array( sorts along last axis (across))

1. a = np.array([[0, 5], [3, 2]])
2. indices = np.argsort(a, axis=1)



### Example 7: Sorting with keys

1. a= np.array([(0, 5), (3, 2),(1,6)], dtype=[('x', '<i4'), ('y', '<i4')])
2. b=np.argsort(a, order=('x','y'))
3. c=np.argsort(a, order=('y','x'))



# numpy.transpose() in Python

This function permutes or reserves the dimension of the given array and returns the modified array.

The numpy.transpose() function changes the row elements into column elements and the column elements into row elements. The output of this function is a modified array of the original one.

### Syntax: numpy.transpose(arr, axis=None)

**arr: array\_like**

It is an ndarray. It is the source array whose elements we want to transpose. This parameter is essential and plays a vital role in numpy.transpose() function.

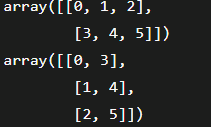
**axis: List of ints()**

If we didn't specify the axis, then by default, it reverses the dimensions otherwise permute the axis according to the given values.

### Return: This function returns a ndarray. The output array is the source array, with its axis permuted. A view is returned whenever possible.

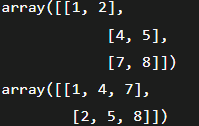
### Example 1: numpy.transpose()

1. a= np.arange(6).reshape((2,3))
2. b=np.transpose(a)



### Example 2: numpy.transpose() with axis

1. a= np.array([[1, 2], [4, 5], [7, 8]])
2. b=np.transpose(a, (1,0))



### Example 3: Reposition elements using numpy.transpose()

1. a=np.ones((12,32,123,64))
2. b=np.transpose(a,(1,3,0,2)).shape
3. c=np.transpose(a,(0,3,1,2)).shape



# numpy.mean() in Python

The sum of elements, along with an axis divided by the number of elements, is known as **arithmetic mean**. The numpy.mean() function is used to compute the arithmetic mean along the specified axis.

This function returns the average of the array elements. By default, the average is taken on the flattened array. Else on the specified axis, float 64 is intermediate as well as return values are used for integer inputs

### Syntax: numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)

**a: array\_like**

This parameter defines the source array containing elements whose mean is desired. In such a case where 'a' is not an array, a conversion is attempted.

**axis: None, int or tuple of ints(optional)**

This parameter defines the axis along which the means are computed. By default, the mean is computed of the flattened array. In version 1.7.0, if this is a tuple of ints, the mean is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype: data-type(optional)**

This parameter is used to define the data-type used in computing the mean. For integer inputs, default is float64, and for floating-point inputs, it is the same as the input dtype.

**out: ndarray(optional)**

This parameter defines an alternative output array in which the result will be placed. The shape of the resulting array should be the same as the shape of the expected output. The type of output values will cast when necessary.

**keepdims: bool(optional)**

When the value is true, the reduced axis is left as dimensions with size one in the output/result. Also, the result broadcasts correctly against the input array. When the default value is set, the keepdims does not pass via the mean method of sub-classes of ndarray, but any non-default value will surely pass. In case the sub-class method does not implement keepdims, then an exception will surely rise.

### Return: If we set the 'out' parameter to **None**, this function returns a new array containing the mean values. Otherwise, it will return the reference to the output array.

1. a = np.array([[1, 2], [3, 4]])
2. b=np.mean(a) //2.5
3. x = np.array([[5, 6], [7, 34]])
4. y=np.mean(x) //13.0
5. a = np.array([[2, 4], [3, 5]])
6. b=np.mean(a,axis=0) //array([2.5, 4.5])
7. c=np.mean(a,axis=1) //array([3., 4.])

### Example 3: In single precision, mean can be inaccurate.

1. a = np.zeros((2, 512\*512), dtype=np.float32)
2. a[0, :] = 23.0
3. a[1, :] = 32.0
4. c=np.mean(a) //27.5

### Example 4: Computing the mean in float64 is more accurate:

1. a[0, :] = 2.0
2. a[1, :] = 0.2
3. c=np.mean(a) //1.0999985
4. d=np.mean(a, dtype=np.float64) //1.1000000014901161

# numpy.unique() in Python

The numpy module of Python provides a function for finding unique elements in a numpy array. The numpy.unique() function finds the unique elements of an array and returns these unique elements as a sorted array. Apart from the unique elements, there are some optional outputs also, which are as follows:

* The output can be the indices of the input array which give the unique values
* The output can be the indices of the unique array which reconstruct input array
* The output can be an array of the number of times each unique value comes in the input array.

### Syntax: numpy.unique(a, return\_index=False, return\_inverse=False, return\_counts=False, axis=None)

**a: array\_like**

This parameter defines the source array containing elements whose unique values are desired. The array will be flattened if it is not 1-D array.

**Return\_index: bool(optional)**

If this parameter is set True, the function will return the indices of the input array(along the specified axis if provided or in the flattened array), which results in the unique array.

**return\_inverse: bool(optional)**

If this parameter is set True, the function will also return the indices of the input array(along the specified axis if provided or in the flattened array), which can be used to reconstruct the input array.

**Return\_counts: bool(optional)**

If this parameter is set True, the function will return the number of times each unique item appeared in the input array 'a'.

**axis: int or None(optional)**

This parameter defines the axis to operate on. If this parameter is not set, then the array 'a' will be flattened. If this parameter is an integer, then the subarrays indexed by the given axis will be flattened and treated as an element of a 1-D array with the dimension of the given axis. Structured arrays or object arrays that contain objects are not supported if the axis 'kwarg' is used.

### Returns: This function returns four types of outputs which are as follows:

**unique: ndarray:** In this, a ndarray will be shown that contain sorted unique values.

**unique\_indices: ndarray(optional)**

In this output, a ndarray will be shown that contains the indices of the first occurrences of unique values in the original array. This output is only provided if return\_index is True.

AD

**unique\_inverse: ndarray(optional)**

In this output, a ndarray will be shown that contains the indices to reconstruct the original array from the unique array. This output is only provided if return\_inverse is True.

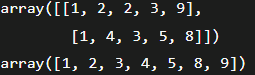
**unique\_counts: ndarray(optional)**

In this output, a ndarray will be shown that contains the number of times each of the unique values comes up in the original array. This output is only provided if return\_counts is True.

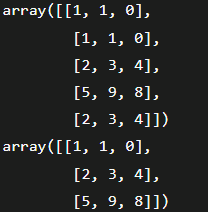
a=np.unique([1,2,3,4,3,6,2,4]) //array([1, 2, 3, 4, 6])

### Example 2:

1. a=np.array([[1,2,2,3,9],[1,4,3,5,8]])
2. b=np.unique(a)

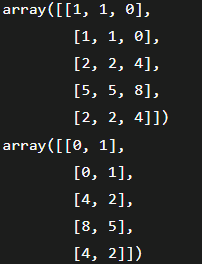


1. a = np.array([[1, 1, 0], [1, 1, 0], [2, 3, 4],[5, 9, 8],[2, 3, 4]])
2. b=np.unique(a, axis=0)



In output, a ndarray has been shown that contains unique rows of the source array 'a'.

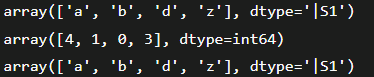
1. a = np.array([[1, 1, 0], [1, 1, 0], [2, 2, 4],[5, 5, 8],[2, 2, 4]])
2. b=np.unique(a, axis=1)



#### Note: When we set axis as 1 then this functn returns the unique columns from source array.

### Example 5: Use return\_index

1. a = np.array(['d', 'b', 'b', 'z', 'a'])
2. result, indices=np.unique(a,return\_index=True)
3. result
4. indices
5. a[indices]

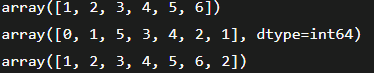


In the output, a ndarray has been shown that contains the indices of the original array that give unique values.

### Ex: Use return\_inverse

We can reconstruct the input array from the unique values in the following way:

1. a = np.array([1, 2, 6, 4, 5, 3, 2])
2. result, indices=np.unique(a,return\_inverse=True)
3. result
4. indices
5. a[indices]



# numpy.ndarray.tolist() in Python

The numpy module provides a function **numpy.ndarray.tolist()**, used to convert the data elements of an array into a list. This function returns the array as an **a.ndim-** levels deep nested list of Python scalars.

In simple words, this function returns a copy of the array elements as a Python list. The elements are converted to the nearest compatible built-in Python type through the item function. When 'a.ndim' is 0, then the depth of the list is 0, and it will be a simple Python scalar, not any list.

### Syntax: ndarray.tolist()

### Parameter: This function has no arguments or parameters.

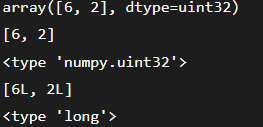
### Returns: y: object, or list of object, or list of object

This function returns the possibly nested list of array elements.

#### Note: We can re-create the array via a=np.array(a.tolist()), however it can sometimes lose precision.

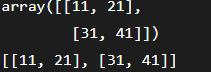
### Ex1: If we will use **a.tolist()** for a 1D array then it will be almost the same as **list(a)**, except that **tolist** converts numpy scalars to Python scalars.

1. a = np.uint32([6, 2])
2. a\_list=list(a)
3. type(a\_list[0])
4. a\_tolist=a.tolist()
5. type(a\_tolist[0])



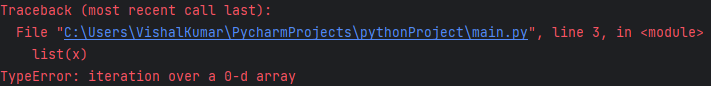
### Example 2: For a 2-dimensional array, **tolist** is applied recursively.

1. a = np.array([[11, 21], [31, 41]])
2. b=a.tolist()



In the output, it shows a list whose elements are transformed from the source array.

1. **import** numpy as np
2. x = np.array(5)
3. list(x)
4. y=x.tolist()



# numpy.dot() in Python

numpy module of Python provides a function to perform the dot product of two arrays.

* If both the arrays 'a' and 'b' are 1-dimensional arrays, the dot() function performs the inner product of vectors (without complex conjugation).
* If both the arrays 'a' and 'b' are 2-dimensional arrays, the dot() function performs the matrix multiplication. But for matrix multiplication use of **matmul** or **'a' @ 'b'** is preferred.
* If either 'a' or 'b' is 0-dimensional (scalar), the dot() function performs multiplication. Also, the use of **numpy.multiply(a, b)** or **a \*b** method is preferred.
* If 'a' is an N-dimensional array and 'b' is a 1-dimensional array, then the dot() function performs the sum-product over the last axis of a and b.
* If 'a' is an M-dimensional array and 'b' is an N-dimensional array (where N>=2), then the dot() function performs the sum-product over the last axis of 'a' and the second-to-last axis of 'b':

1. **dot(a, b)[i,j,k,n] = sum(a[i,j,:] \* b[k,:,n])**

### Syntax: numpy.dot(a, b, out=None)

**a: array\_like:** This parameter defines the first array.

**b: array\_like:** This parameter defines the second array.

**out: ndarray(optional)**

It is an output argument. It should have the exact kind which would be returned in the case when it was not used. Particularly, it should meet the performance feature, i.e., it must contain the right type, i.e., it must be C-contiguous, and its dtype must be the dtype

that would be returned for dot(a,b). Thus, if it does not meet these specified conditions, it raises an exception.

### Returns: This function returns the dot product of 'a' and 'b'. This function returns a scalar if 'a' and 'b' are both scalars or 1-dimensional; otherwise, it returns an array. If 'out' is given, then it is returned.

### Raises: The **ValueError** occurs when the last dimension of 'a' is not having the same size as the second-to-last dimension of 'b'.

1. a=np.dot(6,12) //72
2. a=np.dot([2j, 3j], [5j, 8j]) //(-34+0j)

### Example 3:

1. a = [[1, 2], [4, 1]]
2. b = [[4, 11], [2, 3]]
3. c=np.dot(a, b)



1. x = np.arange(3\*4\*5\*6).reshape((3,4,5,6))
2. y = np.arange(3\*4\*5\*6)[::-1].reshape((5,4,6,3))
3. p=np.dot(x, y)[2,3,2,1,2,2] //499128
4. q=sum(x[2,3,2,:] \* y[1,2,:,2]) //499128

# numpy.loadtxt() in Python

The numpy module of Python provides a function to load data from a text file. The numpy module provides **loadtxt()** function to be a fast reader for simple text files.

#### Note: In the text file, each row must have the same number of values.

### Syntax: numpy.loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)

**fname: file, str, or pathlib.Path:**

This parameter defines the file, filename, or generator to read. Firstly, we will decompose the file, if the filename extension is **.gz** and **.bz2**. After that the generators will return byte strings for [Python](https://www.javatpoint.com/python-tutorial) 3k.

**dtype: data-type(optional)**

This parameter defines the data type for the resulting array, and by default, the data type will be the float. The resulting array will be 1-dimensional when it is a structured data-type. Each row is interpreted as an array element, and the number of columns used must match with the number of fields in the data-type.

**comments: str or sequence(optional)**

This parameter defines the characters or list of characters used for indicating the start of the comment. By default, it will be '**#**'.

**delimiter: str(optional):** This parameter defines the string used for separating values. By default, it will be any whitespace.

**converters: dict(optional)**

This parameter defines a dictionary mapping column number to a function that will convert the mapped column to the float. When column() is a date string then **converters={0:datestr2num}**. This parameter is also used to provide a default value for missing data as **converters= {3: lambda s: float(s.strip() or 0)}**.

**skiprows: int(optional):** This parameter is used to skip the first 'skiprows', and by default, it will be 0.

**usecols: int or sequence(optional)**

This parameter defines the columns to read, with 0 being the first. For example, usecols=(0, 3, 5) will extract the 1st, 4th, and 5th column. By default, its value is None, which results in all columns being read. In the new version, we can use an integer instead of a tuple if we want to read a single column.

**unpack: bool(optional)**

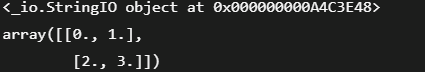
If this parameter is set to true, then the returned array is transposed, so that arguments may be unpacked using **x, y, z =loadtxt(...)**. The arrays are returned for each field when using it with the structured data-type. By default, it will be set to False.

**ndim: int(optional)**

The returned array will have 'ndmin' dimensions. Otherwise, it will squeeze the mono-dimensional axis. Legal values: 0 (default), 1 or 2.

### Returns: out(ndarray): It reads data from the text file in the form of a ndarray.

1. from io **import** StringIO
2. c = StringIO(u"0 1\n2 3")
3. np.loadtxt(c)



In the output, it shows the content of the file in the form of [ndarray](https://www.javatpoint.com/numpy-ndarray).

1. c = StringIO(u"1,3,2\n3,5,4")
2. x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
3. p,q,R = np.loadtxt(c, delimiter=',', unpack=True)





How much column we provide in useCols, it will return same values.

# numpy.clip() in Python

For clipping the values in an array, the numpy module of Python provides a function called **numpy.clip()**. In the clip() function, we will pass the interval, and the values which are outside the interval will be clipped for the interval edges.

If we specify an interval of [1, 2] then the values smaller than 1 become 1 and larger than 2 is 2. This function is similar to **numpy.maximum(x\_min, numpy.maximum(x, x\_max))**. But it is faster than np.maximum(). In **numpy.clip()**, there is no need to perform check for ensuring **x\_min < x\_max**.

### Syntax: numpy.clip(a, a\_min, a\_max, out=None)

**x: array\_like:** This parameter defines the source array whose elements we want to clip.

**x\_min: None, scalar, or array\_like**

This parameter defines the minimum value for clipping values. On the lower interval edge, clipping is not required.

**x\_max: None, scalar, or array\_like**

This parameter defines the maximum value for clipping values. On the upper interval edge, clipping is not required. The three arrays are broadcasted for matching their shapes with x\_min and x\_max arrays. This will be done only when x\_min and x\_max are array\_like.

**out: ndaaray(optional)**

This parameter defines the ndarray in which the result will be stored. For in-place clipping, this can be an input array. The data type of this 'out' arrays have the right shape for holding the output.

### Returns: **clip\_arr: ndarray**

This function returns an array that contains the elements of 'x' but the values which are less than the **x\_min,** they get replaced with **x\_min**, and those which are greater than **x\_max**, they get replaced with **x\_max**.

1. x= np.arange(12)
2. y=np.clip(x, 3, 10)



In the output, a ndarray is shown, which contains elements ranging from 3 to 10.

1. a = np.arange(12)
2. np.clip(a, 3, 9, out=a)
3. a



**If we use out=a then It will assign the result to “a”.**

1. a = np.arange(12)
2. np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4, 5, 6], 8) //array([3, 4, 3, 3, 4, 5, 6, 7, 8, 8, 8, 8])

# numpy.ndarray.flatten() in Python

In Python, for some cases, we need a one-dimensional array rather than a 2-D or multi-dimensional array. For this purpose, the numpy module provides a function called **numpy.ndarray.flatten(),** which returns a copy of the array in one dimensional rather than in 2-D or a multi-dimensional array.

### Syntax: ndarray.flatten(order='C')

**order: {'C', 'F', 'A', 'K'}(optional)**

If we set the order parameter to 'C', it means that the array gets flattened in row-major order. If 'F' is set, the array gets flattened in column-major order. The array is flattened in column-major order only when 'a' is Fortran contiguous in memory, and when we set the order parameter to 'A'. The last order is 'K', which flatten the array in same order in which the elements occurred in the memory. By default, this parameter is set to 'C'.

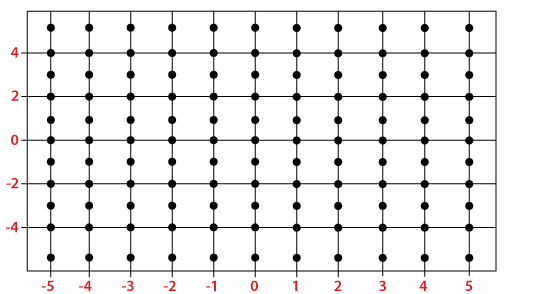
### Returns: **y: ndarray**

This function returns a copy of source array, which gets flattened into one-dimensional.

1. a = np.array([[1,4,7], [2,5,8],[3,6,9]])
2. b=a.flatten() //array([1, 4, 7, 2, 5, 8, 3, 6, 9])
3. b=a.flatten('C') //array([1, 4, 7, 2, 5, 8, 3, 6, 9])
4. b=a.flatten('F') //array([1, 2, 3, 4, 5, 6, 7, 8, 9])
5. b=a.flatten('A') //array([1, 4, 7, 2, 5, 8, 3, 6, 9])
6. b=a.flatten('K') //array([1, 4, 7, 2, 5, 8, 3, 6, 9])

# numpy.meshgrid() in Python

The numpy module of Python provides **meshgrid()** function for creating a rectangular grid with the help of the given 1-D arrays that represent the **Matrix indexing** or **Cartesian indexing**. MATLAB somewhat inspires the meshgrid() function. From the coordinate vectors, the meshgrid() function returns the coordinate matrices.



In the above figure, the x-axis is ranging from -5 to 5, and the y-axis is ranging from -5 to 5. So, there is a total of 121 points marked in the figure, each with x-coordinate and y-coordinate. For any line parallel to the x-axis, the x-coordinates of the marked points are -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, and 5 respectively. On the other hand, for any line parallel to the y-axis, the y-coordinates of the marked points from bottom to top are -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, and 5 respectively.

### Syntax: numpy.meshgrid(\*xi, \*\*kwargs)

**x1, x2,…, xn : array\_like:** This parameter defines the 1-dimensional array, which represents the coordinates of a grid.

**indexing: {'xy', 'ij'}(optional)**

This is an optional argument which defines the Cartesian 'xy'(by default) or matrix ('ij') indexing of output.

**sparse: bool(optional)**

This parameter is also optional. If we need a sparse grid for conserving memory, we have to set this parameter to True. By default, it is set to False.

**copy: bool(optional)**

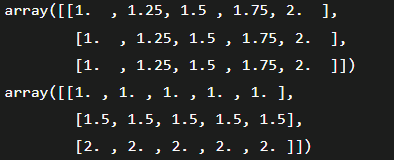
The aim of this optional argument is that it returns a copy of the original array for conserving memory. By default, it is set to False.

If both **sparse** and **copy** parameters are set to False, then it will return non-contiguous arrays. In addition, more than one element of a broadcast array can refer to a single memory location. If we need to write into the arrays, then we have to make copies first.

### Returns : **X1, X2, …, Xn**

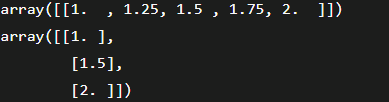
The coordinate length from the coordinate vector is returned from this function.

1. na, nb = (5, 3)
2. a = np.linspace(1, 2, na)
3. b = np.linspace(1, 2, nb)
4. xa, xb = np.meshgrid(a, b)

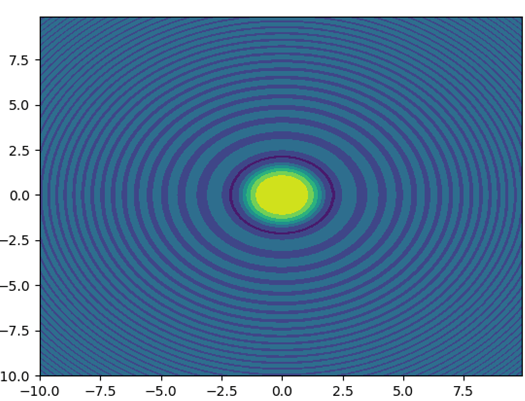


In the output, two arrays have been shown which contain the coordinate length from the coordinate vectors.

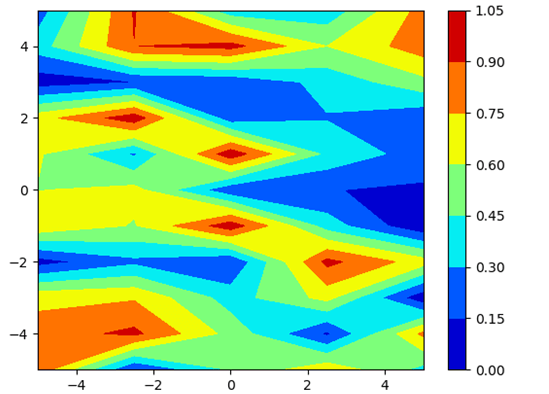
1. na, nb = (5, 3)
2. a = np.linspace(1, 2, na)
3. b = np.linspace(1, 2, nb)
4. xa, xb = np.meshgrid(a, b, sparse=True)



1. a = np.arange(-10, 10, 0.1)
2. b = np.arange(-10, 10, 0.1)
3. xa, xb = np.meshgrid(a, b, sparse=True)
4. z = np.sin(xa\*\*2 + xb\*\*2) / (xa\*\*2 + xb\*\*2)
5. h = plt.contourf(a,b,z)
6. plt.show()



1. a = np.linspace(-5, 5, 5)
2. b = np.linspace(-5, 5, 11)
3. random\_data = np.random.random((11, 5))
4. xa, xb = np.meshgrid(a, b)
5. plt.contourf(xa, xb, random\_data, cmap = 'jet')
6. plt.colorbar()
7. plt.show()



# numpy standard deviation

The numpy module of Python provides a function called **numpy.std()**, used to compute the standard deviation along the specified axis. This function returns the standard deviation of the array elements. The square root of the average square deviation (computed from the mean), is known as the standard deviation. By default, the standard deviation is calculated for the flattened array. With the help of the **x.sum()/N**, the average square deviation is normally calculated, and here, N=len(x).

Standard Deviation=sqrt(mean(abs(x-x.mean( ))\*\*2

### Syntax: numpy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<**class** numpy.\_globals.\_NoValue>)

**a: array\_like:** This parameter defines the source array whose elements standard deviation is calculated.

**axis: None, int, or tuple of ints(optional)**

It is the axis along which the standard deviation is calculated. The standard deviation of the flattened array is computed by default. If it is a tuple of ints, performs standard deviation over multiple axis instead of a single axis or all axis as before.

**dtype : data\_type(optional)**

This parameter defines the data type, which is used in computing the standard deviation. By default, the data type is float64 for integer type arrays, and, for float types array, it will be the same as the array type.

**out : ndarray(optional)**

This parameter defines the alternative output array in which the result is to be placed. This alternative ndarray has the same shape as the expected output. But we cast the type when necessary.

**ddof : int(optional)**

This parameter defines the Delta Degrees of Freedom. The N-ddof divisor is used in calculations, where N is the number of elements. By default, the value of this parameter is set to 0.

**keepdims : bool(optional)**

It is optional, whose value, when true, will leave the reduced axis as dimensions with size one in the resultant. When it passes the default value, it will allow the non-default values to pass via the mean method of sub-classes of ndarray, but the keepdims will not pass. Also, the output or the result will broadcast against the input array correctly.

### Returns: This function will return a new array that contains the standard deviation. If we do not set the 'out' parameter to None, it returns the output array's reference.

1. a=np.array([[1,4,7,10],[2,5,8,11]])
2. b=np.std(a) //3.391164991562634
3. b=np.std(a, axis=0) //array([0.5, 0.5, 0.5, 0.5])
4. b=np.std(a, axis=1) //array([3.35410197, 3.35410197])
5. a = np.zeros((2, 512\*512), dtype=np.float32)
6. a[1, :] = 1.0
7. a[0, :] = 0.1
8. b=np.std(a) //0.45000008
9. b=np.std(a ,dtype=np.float64)) //0.4499999992549418

# numpy.argmax in Python

In many cases, where the size of the array is too large, it takes too much time to find the maximum elements from them. For this purpose, the numpy module of Python provides a function called **numpy.argmax()**. This function returns indices of the maximum values are returned along with the specified axis.

### Syntax: numpy.argmax(a, axis=None, out=None)

**x: array\_like** This parameter defines the source array whose maximum value we want to know.

**axis: int(optional)**

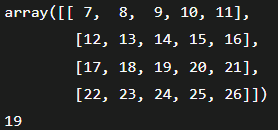
This parameter defines the axis along which the index is present, and by default, it is into the flattened array.

**out: array(optional)**

This parameter defines the ndarray in which the result is going to be inserted. This will be of the same type and shape, which is appropriate for storing the result.

### Returns: This parameter defines a ndarray, which contains the indices of the array. The shape is the same as **x.shape** with the dimension along the axis removed.

1. x = np.arange(20).reshape(4,5) + 7
2. y=np.argmax(x)

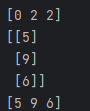


In the output, it shows the indices of the maximum element in the array.

1. x = np.arange(20).reshape(4,5) + 7
2. y=np.argmax(x, axis=0)
3. z=np.argmax(x, axis=1)



1. x = np.arange(20).reshape(4,5) + 7
2. indices = np.unravel\_index(np.argmax(x, axis=None), x.shape)
3. indices //(3, 4)
4. x[indices] //26
5. a = np.array([[5,2,1], [3,7,9],[0, 4, 6]])
6. index\_arr = np.argmax(a, axis=-1)
7. print(index\_arr) # Same as np.max(a, axis=-1, keepdims=True)
8. result1 = np.take\_along\_axis(a, np.expand\_dims(index\_arr, axis=-1), axis=-1)
9. print(result1) # Same as np.max(a, axis=-1)
10. result2 = np.take\_along\_axis(a, np.expand\_dims(index\_arr, axis=-1), axis=-1).squeeze(axis=-1)
11. print(result2)



In the output, it shows indices of the maximum elements in the array and the values which are present on that indices.

# numpy.diff() in Python

The numpy module of **Python** provides a function called **numpy.diff** for calculating the nth discrete difference along the given axis. If **'x'** is the input array, then the first difference is given by out[i]=x[i+1]-a[i]. We can calculate the higher difference by using diff recursively.

### Syntax: numpy.diff(a, n=1, axis=-1, prepend=<no value>, append=<no value>)

**x: array\_like**

This parameter defines the source array whose elements nth discrete deference are those which we want to calculate.

**n: int(optional)**

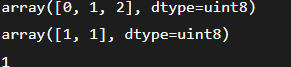
This parameter defines the number of times the values are differenced. If it is 0, then the source array is returned as it is.

**append, prepend: array\_like(optional)**

This parameter defines a ndarray, which defines the values going to append or prepend to **'x'**, along the axis before computing differences.

### Returns: This function returns a ndarray containing nth differences having the same shape as **'x,'** and the dimension is smaller from **n**. The type of difference between any two elements of **'x'** is the type of the output.

1. arr = np.array([0, 1, 2], dtype=np.uint8)
2. b=np.diff(arr)
3. arr[2,...] - arr[1,...] - arr[0,...]

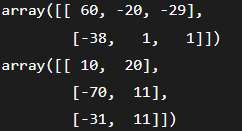


In the output, it shows the discrete differences of elements.

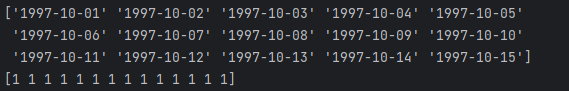
1. x = np.array([11, 21, 41, 71, 1, 12, 33, 2])
2. y = np.diff(x)



1. x = np.array([[11, 21, 41], [71, 1, 12], [33, 2, 13]])
2. y = np.diff(x, axis=0)
3. z = np.diff(x, axis=1)



1. x = np.arange('1997-10-01', '1997-10-16', dtype=np.datetime64)
2. y = np.diff(x)



# numpy.empty() in Python

The numpy module of Python provides a function called **numpy.empty()**. This function is used to create an array without initializing the entries of given shape and type.

Just like **numpy.zeros()**, the **numpy.empty()** function doesn't set the array values to zero, and it is quite faster than the **numpy.zeros()**. This function requires the user to set all the values in the array manually and should be used with caution.

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

**shape: int or tuple of ints:** This parameter defines the shape of the empty array, such as (3, 2) or (3, 3).

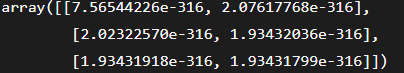
**dtype: data-type(optional):** This parameter defines the data type, which is desired for the output array.

**order: {'C', 'F'}(optional)**

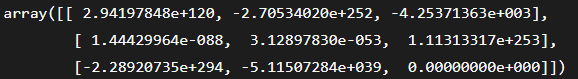
This parameter defines the order in which the multi-dimensional array is going to be stored either in **row-major** or **column-major**. By default, the order parameter is set to **'C'**.

### Returns: This function returns the array of uninitialized data that have the shape, dtype, and order defined in the function.

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



1. x = np.empty([3, 3], dtype=**float**, order='C')



In output, it shows an array of uninitialized values of defined shape, data type, and order.

# numpy.histogram() in Python

The numpy module of Python provides a function called **numpy.histogram()**. This function represents the frequency of the number of values that are compared with a set of values ranges. This function is similar to the **hist()** function of **matplotlib.pyplot**.

In simple words, this function is used to compute the histogram of the set of data.

### Syntax: numpy.histogram(x, bins=10, range=None, normed=None, weights=None, density=None)

**x:array\_like:**This parameter defines a flattened array over which histogram is computed.

**bins: int or sequence of str or scalars(optional)**

int or sequence of str defines number of equal width bins in a range, default is 10.

**range : (float, float)(optional)**

This parameter defines the lower-upper ranges of the bins. By default, the range is **(x.min(), x.max())**. The values are ignored, which are outside the range. The ranges of the first element should be equal to or less than the second element.

**normed : bool(optional)**

This parameter is the same as the density argument, but it can give the wrong output for unequal bin widths.

**weights : array\_like(optional)**

This parameter defines an array which contains weights and has the same shape as **'x'**.

**density : bool(optional)**

If it is set to True, will result in the number of samples in every bin. If its value is False, the density function will result in the value of the probability density function in the bin.

### Returns: **hist: array** The density function returns the values of the histogram.

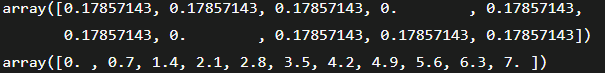
**edge\_bin: an array of float dtype:**ADThis function returns the bin edges **(length(hist+1))**.

In the output, it shows a ndarray that contain the values of the histogram.

1. x=np.histogram(np.arange(6), bins=np.arange(7), density=True)



1. a = np.arange(8)
2. hist, bin\_edges = np.histogram(a, density=True)
3. hist
4. bin\_edges



# numpy.sort in Python

In some cases, we require a sorted array for computation. For this purpose, the numpy module of Python provides a function called **numpy.sort()**. This function gives a sorted copy of the source array or input array.

### Syntax: numpy.sort(a, axis=-1, kind='quicksort', order=None)

**x: array\_like** This parameter defines the source array, which is going to be sorted.

**axis: int or None(optional)**

This parameter defines the axis along which the sorting is performed. If this parameter is **None**, the array will be flattened before sorting, and by default, this parameter is set to -1, which sorts the array along the last axis.

**kind: {quicksort, heapsort, mergesort}(optional)**

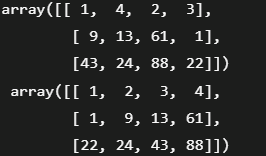
This parameter is used to define the sorting algorithm, and by default, the sorting is performed using **'quicksort'**.

**order: str or list of str(optional)**

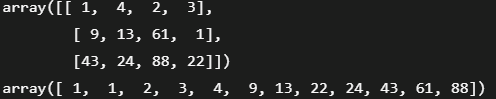
When an array is defined with fields, its order defines the fields for making a comparison in first, second, etc. Only the single field can be specified as a string, and not necessarily for all fields. However, the unspecified fields will still be used, in the order in which they come up in the dtype, to break the ties.

### Returns: This function returns a sorted copy of the source array, which will have the same shape and type as a source array.

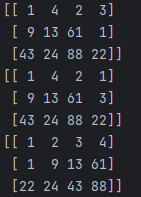
1. x=np.array([[1,4,2,3],[9,13,61,1],[43,24,88,22]])
2. y=np.sort(x)



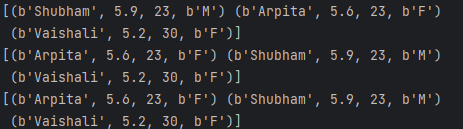
1. x=np.array([[1,4,2,3],[9,13,61,1],[43,24,88,22]])
2. y=np.sort(x, axis=None)



1. x=np.array([[1,4,2,3],[9,13,61,1],[43,24,88,22]])
2. y=np.sort(x,axis=0)
3. z=np.sort(x,axis=1)



1. dtype = [('name', 'S10'), ('height', **float**), ('age', **int**),('gender','S10')]
2. values = [('Shubham', 5.9, 23, 'M'), ('Arpita', 5.6, 23, 'F'),('Vaishali', 5.2, 30, 'F')]
3. x=np.array(values, dtype=dtype)
4. y=np.sort(x, order='age')
5. z=np.sort(x, order=['age','height'])



# numpy.average() in Python

The numpy module of Python provides a function called numpy.average(), used for calculating the weighted average along the specified axis.

### Syntax: numpy.average(a, axis=None, weights=None, returned=False)

**x: array\_like**

This parameter defines the source array whose element's average we want to calculate. The conversion will be attempted if 'x' is an array.

**axis: int or None or tuple of ints(optional)**

**weights : array\_like(optional)**

This parameter defines an array containing weights associated with the array values. Each value of array elements together makes the average according to its associated weight. The weighted array can be one-dimensional or of the same shape as the input array. When there is no weight associated with the array element, the weight will be treated as 1 for all elements.

**returned: bool(optional)**

By default, this parameter is set to False. If we set it as True, then a tuple of average and sum\_of\_weights, is returned. If it is False, the average is returned. The weighted sum is equivalent to the number of elements if there are no values for weights.

### Returns: **retval, [sum\_of\_weights]: array\_type or double**

This function returns either the average or both the average and the sum\_of\_weights that depend on the returned parameter.

### Raises:

**ZeroDivisionError:** This error is raised when all weights along the axis are set to zero.

**TypeError:** This error is raised when the length of the weighted array is not the same as the shape of the input array.

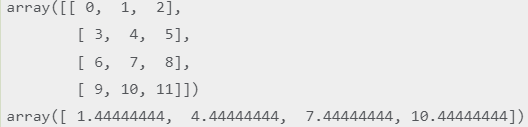
1. data = list(range(1,6))
2. output=np.average(data)



1. output=np.average(range(1,16), weights=range(15,0,-1))



1. data=np.arange(12).reshape((4,3))
2. output = np.average(data, axis=1, weights=[1./4, 3./4, 5./4])



# numpy.pad() in Python

The numpy module of Python provides a function called numpy.pad() to perform padding in the array. This function has several required and optional parameters.

### Syntax: numpy.pad(array, pad\_width, mode='constant', \*\*kwargs)

### array: array\_like This is the source array which we want to pad.

### pad\_width: int, sequence, or array\_like

This parameter defines the number of values that are padded to the edges of each axis. The unique pad widths for each axis are defined as (before\_1, after\_1), (before\_2, after\_2), ... (before\_N, after\_N)). For each axis, ((before, after),) will be treated as same as before and after pad. For all axes, the int, or (pad,) is a shortcut to before = after = pad width.

### mode: str or function(optional) This parameter has one of the following string values:

**'constant'(Default)** If we assign a constant value to the mode parameter, padding will be done with a constant value.

**'Edge'** It is the edge value of the array. The padding will be done with this edge value.

**'Linear\_ramp'** This value is used to perform padding with the linear ramp between the edge value and the end value.

**'Maximum'** This parameter value performs padding by using the max value of a vector part or all, along each axis.

**'Mean'** This parameter value performs padding via the mean value of a vector part or all, along each axis.

**'Median'** This parameter value performs padding via the median value of a vector part or all, along each axis.

**'Minimum'** This parameter value performs padding via the min value of a vector part or all, along each axis.

**'Reflect'** This value pads the array via vector reflection, which is mirrored on the starting and ending vector values, along each axis.

**'Symmetric'** This value is used to pad the array via vector reflection, which is mirrored along the edge of the array.

**'Wrap'** This value is used to perform padding of the array via the wrap of the vector along the axis. The starting values are used for padding the end, and the ending values pad the beginning.

**'Empty'** This value is used to pad the array with undefined values.

### stat\_length: int or sequence(optional)

This parameter is used in 'maximum', 'minimum', 'mean', 'median'. It defines the number of values at each edge axis, used for calculating the static value.

### constant\_values: scalar or sequence(optional)

This parameter is used in 'constant'. It defines the values for setting the padded values to each axis.

### end\_values: scalar or sequence(optional)

This parameter is used in 'linear\_ramp'. It defines the values which are used for the last value of the linear\_ramp and will form the edge of the padded array.

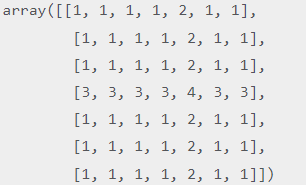
### reflect\_type: even or odd(optional)

This parameter is used in 'symmetric' and 'reflect'. By default, the reflect\_type is 'even' with an unaltered reflection around the edge value. By subtracting the reflected values from two times the edge value, the array's extended part is created for the 'odd' style.

### Returns: **pad: ndarray**

This function returns the padded array of rank equal to the array, whose shape increase according to pad\_width.

1. x = [1, 3, 2, 5, 4]
2. y = np.pad(x, (3, 2), 'constant', constant\_values=(6, 4)) //array([6, 6, 6, 1, 3, 2, 5, 4, 4, 4])
3. y = np.pad(x, (3, 2), 'edge') //array([1, 1, 1, 1, 3, 2, 5, 4, 4, 4])
4. y = np.pad(x, (3, 2), 'linear\_ramp', end\_values=(-4, 5)) //array([-4, -2, 0, 1, 3, 2, 5, 4, 4, 5])
5. y = np.pad(x, (3,), 'maximum') //array([5, 5, 5, 1, 3, 2, 5, 4, 5, 5, 5])
6. y = np.pad(x, (3,), 'mean') //array([3, 3, 3, 1, 3, 2, 5, 4, 3, 3, 3])
7. y = np.pad(x, (3,), 'median') //array([3, 3, 3, 1, 3, 2, 5, 4, 3, 3, 3])
8. a = [[1, 2], [3, 4]]
9. y = np.pad(x, (3,), 'minimum')



# numpy.ravel() in Python

The numpy module of Python provides a function called numpy.ravel, which is used to change a 2-dimensional array or a multi-dimensional array into a contiguous flattened array. The returned array has the same data type as the source array or input array. If the input array is a masked array, the returned array will also be a masked array.

### Syntax: numpy.ravel(x, order='C')

**x: array\_like**

This parameter defines the input array, which we want to change in a contiguous flattened array. The array elements are read in the order specified by the order parameter and packed as a 1-D array.

**order: {'C','F', 'A', 'K'}(optional)**

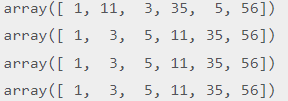
If we set the order parameter to 'C', it means that the array gets flattened in row-major order. If 'F' is set, the array gets flattened in column-major order. The array is flattened in column-major order only when 'A' is Fortran contiguous in memory, and when we set the order parameter to 'A'. The last order is 'K', which flatten the array in same order in which the elements occurred in the memory. By default, this parameter is set to 'C'.

### Returns: This function returns a contiguous flatten array with the same data type as an input array and has shape equal to (**x.size**).

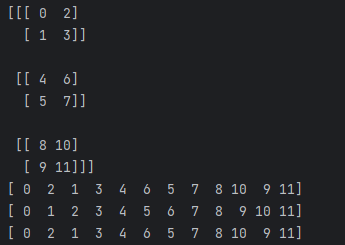
1. x = np.array([[1, 3, 5], [11, 35, 56]])
2. y=np.ravel(x)



1. x = np.array([[1, 3, 5], [11, 35, 56]])
2. y = np.ravel(x, order='F')
3. z = np.ravel(x, order='C')
4. p = np.ravel(x, order='A')
5. q = np.ravel(x, order='K')
6. y, z, p, q



1. x = np.arange(12).reshape(3,2,2).swapaxes(1,2)
2. y=np.ravel(x, order='C')
3. z=np.ravel(x, order='K')
4. q=np.ravel(x, order='A')



# numpy.save() in Python

The numpy module of Python provides a function called numpy.save() to save an array into a binary file in .npy format. In many of the cases, we require data in binary format to manipulate it.

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

**file: str, file, or pathlib.path**

This parameter defines the file or filename in which the data is going to be saved. If this parameter is an object of a file, the filename will be unchanged. If the **file** parameter is a path or a string, the .npy extension will be added to the file name, and it will be done when it doesn't have one.

**allow\_pickle : bool(optional)**

This parameter is used to allow the saving of objects into the pickle. The security and the probability are the reason for disallowing pickles.

**fix\_imports : bool(optional)**

If fix\_imports is set to True, then pickle does the mapping of the new Python3 names to the old module names, which are used in Python2. This makes the pickle data stream readable with Python2.

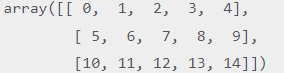
1. from tempfile **import** TemporaryFile
2. out\_file = TemporaryFile()
3. x=np.arange(15)
4. np.save(out\_file, x)
5. \_=out\_file.seek(0) # Only needed here to simulate closing & reopening file
6. np.load(outfile)



* We have saved the array's elements as binary in **npy** file using **np.save()** function.
* We have passed the array **'x'** and **filename** in the function.
* We have closed and reopened the file using **seek(0)** function.
* Lastly, we tried to load the **out\_file**.

In output, an array has been shown which contain elements present in the **out\_file.npy**.

1. from tempfile **import** TemporaryFile
2. outfile = TemporaryFile()
3. x=np.arange(15)
4. np.save(outfile, x, allow\_pickle=False)
5. \_=outfile.seek(0) # Only needed here to simulate closing & reopening file
6. np.load(outfile)



# Numpy arccos() method

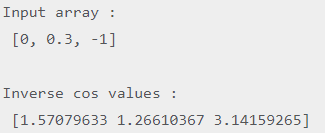
This function is used to calculate the inverse cos of the array elements.

### Syntax: numpy.arccos(array, out)

1. array: these are array elements of which, inverse cos values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing the inverse cos for all the array elements, x.

1. arr = [0, 0.3, -1]
2. **print** ("Input array : \n", arr)
3. arccos\_val = np.arccos(arr)
4. **print** ("\nInverse cos values : \n", arccos\_val)



# Numpy arcsin() method

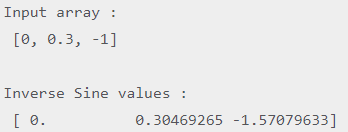
This function is used to calculate the inverse sin of the array elements.

### Syntax: numpy.arcsin(array, out)

1. array: these are array elements of which, inverse sin values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing the inverse sin for all the array elements, x.

1. arr = [0, 0.3, -1]
2. **print** ("Input array : \n", arr)
3. arcsine = np.arcsin(arr)
4. **print** ("\nInverse Sine values : \n", arcsine)



# Numpy arctan() method

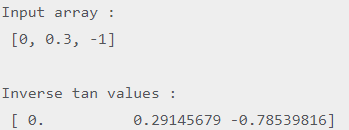
This function is used to calculate the inverse tangent of the array elements.

### Syntax: numpy.arcttan(array, out)

1. array: these are array elmnts of which, inverse tangent valus are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing inverse tangent values for all array elements, x.

1. arr = [0, 0.3, -1]
2. **print** ("Input array : \n", arr)
3. arctan\_val = np.arctan(arr)
4. **print** ("\nInverse tan values : \n", arctan\_val)



# Numpy degrees() method

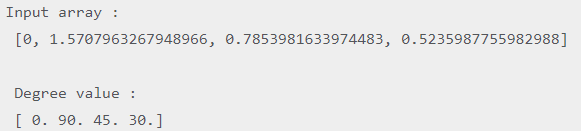
This function is used to convert the angles from radian to degrees.

### Syntax: numpy.degrees(array, out)

1. array: these are the angles whose degree values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing equivalent degree angles of the radians given in the input array.

1. arr = [0, math.pi/2, math.pi/4, math.pi/6 ]
2. **print** ("Input array : \n", arr)
3. degval = np.degrees(arr)
4. **print** ("\n Degree value : \n", degval)



# Numpy tan() method

This function is used to calculate the trigonometric tangent for all the elements of the array passed as the argument.

### Syntax: numpy.tan(array[,out] )

1. array: Array elements whose tangent values are to be calculated.
2. out: shape of the output array.

### Return: An array with trigonometric tangent sins are returned.

1. arr = np.array([0, math.pi/4, 3\*math.pi/2, math.pi/6])
2. **print**("Input Array:",arr)
3. **print**("tan Array:",end=" ")
4. tanarr = np.tan(arr)
5. **print**(tanarr)



# Numpy deg2rad() method

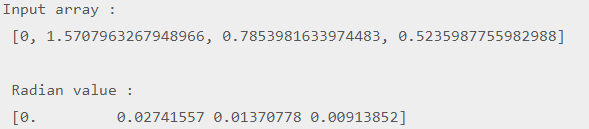
This function is used to convert the angles from degree to radian.

### Syntax: numpy.deg2rad(array, out)

1. array: these are the angles whose degree values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing equivalent degree angles to the radian given in the input array.

1. arr = [0, math.pi/2, math.pi/4, math.pi/6 ]
2. **print** ("Input array : \n", arr)
3. radval = np.deg2rad(arr)
4. **print** ("\n Radian value : \n", radval)



# Numpy hypot() method

This function is used to calculate the hypotenuse for the right angled triangle. The hypotenuse of the right angle is calculated as:

1. h = sqrt(b\*\*2 + p\*\*2)

where b and p are the base and perpendicular of the right angled triangle.

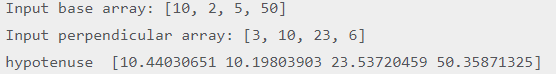
Each value of h is copied into the output array.

### Syntax: numpy.hypot(array1, array2 out)

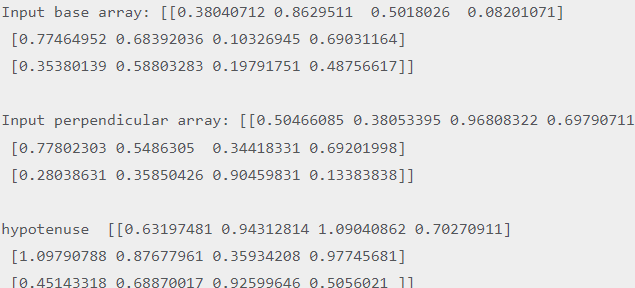
1. array1: It is the input array of bases of the given right angled triangles.
2. array2: It is the input array of perpendiculars of the given right angled triangles.
3. Out: It is the shape of the output array.

### Return: It returns an array containing the hypotenuse of given right angled triangles.

1. base = [10,2,5,50]
2. per= [3,10,23,6]
3. **print**("Input base array:",base)
4. **print**("Input perpendicular array:",per)
5. hyp = np.hypot(base,per)
6. **print**("hypotenuse ",hyp)



1. base = np.random.rand(3,4)
2. per= np.random.rand(3, 4)
3. **print**("Input base array:",base)
4. **print**("Input perpendicular array:",per)
5. hyp = np.hypot(base,per)
6. **print**("hypotenuse ",hyp)



# Numpy rad2deg() method

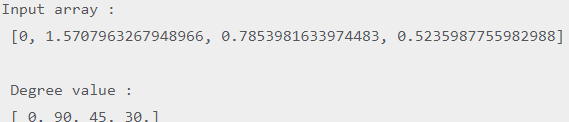
This function is used to convert the angles from radians to degrees.

### Syntax: numpy.rad2deg(array, out)

1. array: these are the angles whose degree values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing equivalent degree angles of the radians given in the input array.

1. arr = [0, math.pi/2, math.pi/4, math.pi/6 ]
2. **print** ("Input array : \n", arr)
3. degval = np.rad2deg(arr)
4. **print** ("\n Degree value : \n", degval)



# Numpy radians() method

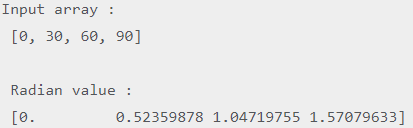
This function is used to convert the angles from degrees to radians.

### Syntax: numpy.radians(array, out)

1. array: these are the angles whose radians values are to be calculated.
2. Out: It is the shape of the output array.

### Return: It returns an array containing equivalent radians angles of the degrees given in the input array.

1. arr = [0, 30, 60, 90 ]
2. **print** ("Input array : \n", arr)
3. radval = np.radians(arr)
4. **print** ("\n Radian value : \n", radval)



# Numpy arcsinh()

This function is used to calculate the hyperbolic inverse sine of the array elements.

### Syntax: numpy.arcsinh(array[,out] )

1. **array:** Array elements (in radians) whose hyperbolic inverse sine values are to be calculated.
2. **out:** shape of the output array.

### Return: An array containing hyperbolic inverse sine values are returned.

1. arr = np.array([0, math.pi/4, 3\*math.pi/2, math.pi/6])
2. **print**("Input Array:",arr)
3. **print**("tanh Array:",end=" ")
4. arcsinharr = np.arcsinh(arr)
5. **print**(arcsinharr)



# Numpy arctanh()

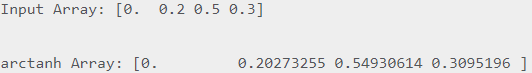
This function is used to calculate the hyperbolic inverse tangent of the array elements.

### Syntax: numpy.arctanh(array[,out] )

1. **array:** Array elements (in radians) whose hyperbolic inverse tangent values are to be calculated.
2. **out:** shape of the output array.

### Return: An array containing hyperbolic inverse tangent values are returned.

1. arr = np.array([0,0.2, 0.5, 0.3])
2. **print**("Input Array:",arr)
3. **print**("arctanh Array:",end=" ")
4. arctanharr = np.arctanh(arr)
5. **print**(arctanharr)



# Numpy ceil()

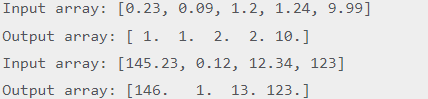
This function returns the ceil value of the input array elements. The floor of a number x is i if i is the smallest integer such that, i>=x.

### Syntax: numpy.ceil(array)

1. **array:** Array elements whose ceil values are to be calculated.

### Return: An array containing the ceil values is returned.

1. arr = [0.23, 0.09, 1.2, 1.24, 9.99]
2. **print**("Input array:",arr)
3. r\_arr = np.ceil(arr)
4. **print**("Output array:",r\_arr)
5. arr2 = [145.23, 0.12, 12.34, 123]
6. r\_arr2=np.ceil(arr2)
7. **print**("Input array:",arr2)
8. **print**("Output array:",r\_arr2)



# Numpy fix()

This function is used to round the array values to the nearest integers towards zero.

### Syntax: numpy.fix(array,b = None)

1. **array:** Array elements which are to be rounded.
2. **b:** it is an Ndarray which is optional.

### Return: An array containing the rounded values is returned.

1. arr = [0.23, 0.09, 1.2, 1.24, 9.99]
2. **print**("Input array:",arr)
3. r\_arr = np.fix(arr)
4. **print**("Output array:",r\_arr)



# 

# Numpy floor()

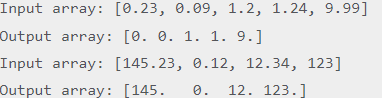
This function returns the floor value of the input array elements. The floor of a number x is i if i is the largest integer such that, i<=x.

### Syntax: numpy.floor(array)

1. **array:** Array elements whose floor values are to be calculated.

### Return: An array containing the floor values is returned.

1. arr = [0.23, 0.09, 1.2, 1.24, 9.99]
2. **print**("Input array:",arr)
3. r\_arr = np.floor(arr)
4. **print**("Output array:",r\_arr)
5. arr2 = [145.23, 0.12, 12.34, 123]
6. r\_arr2=np.floor(arr2)
7. **print**("Input array:",arr2)
8. **print**("Output array:",r\_arr2)



# Numpy rint()

This function is used to round the array elements to the nearest integer.

### Syntax: numpy.rint(array)

### **array:** Array elements whose nearest integers are to be calculated.

### Return: An array containing the rounded values is returned.

1. arr = [0.23, 0.09, 1.2, 1.24, 9.99]
2. **print**("Input array:",arr)
3. r\_arr = np.rint(arr)
4. **print**("Output array:",r\_arr)



# Numpy tanh()

This function is used to calculate the hyperbolic tangent for all the elements of the array passed as the argument.

### Syntax: numpy.tanh(array[,out] )

1. **array:** Array elements whose tangent values are to be calculated (in radians).
2. **out:** shape of the output array.

### Return: An array with trigonometric tangent sins are returned.

1. arr = np.array([0, math.pi/4, 3\*math.pi/2, math.pi/6])
2. **print**("Input Array:",arr)
3. **print**("tanh Array:",end=" ")
4. tanharr = np.tanh(arr)
5. **print**(tanharr)



# Numpy trunc()

This function returns the truncated value of the input array elements. The truncated value t of input value x is the nearest integer which is closer to zero than x.

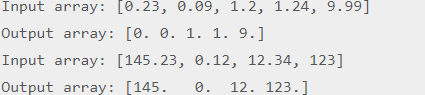
The fractional part of the input number is discarded by this function.

### Syntax: numpy.trunc(array)

1. **array:** Array elements whose truncated values are to be returned.

### Return: An array containing the truncated values are to be returned.

1. arr = [0.23, 0.09, 1.2, 1.24, 9.99]
2. **print**("Input array:",arr)
3. r\_arr = np.trunc(arr)
4. **print**("Output array:",r\_arr)
5. arr2 = [145.23, 0.12, 12.34, 123]
6. r\_arr2=np.trunc(arr2)
7. **print**("Input array:",arr2)
8. **print**("Output array:",r\_arr2)



# numpy.matlib.empty()

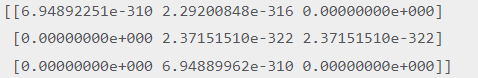
This function is used to return a new matrix with the uninitialized entries.

### Syntax: numpy.matlib.empty(shape,dtype,order)

1. shape: It is the Tuple defining the shape of the matrix.
2. dtype: It is the data type of the matrix.
3. order: It is the insertion order of the matrix.

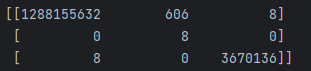
### Return: A matrix with uninitialized entries is returned.

1. **import** numpy.matlib
2. **print**(numpy.matlib.empty((3,3)))



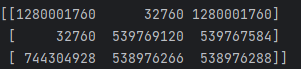
### Example: initializing integer values

1. **print**(numpy.matlib.empty((3,3),int))



### Example: specifying Insertion order

**print**(numpy.matlib.empty((3,3),int,'C'))



# numpy.matlib.eye()

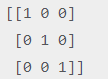
This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere.

### Syntax: numpy.matlib.eye(n, m, k, dtype)

1. n: It represents the number of rows in the resulting matrix.
2. m: It represents the number columns in the resulting matrix.
3. k: It is the index of the diagonal.
4. dtype: It is the data type of the output.

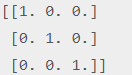
### Return: A matrix with uninitialized entries is returned.

**print**(numpy.matlib.eye(n=3,k=0,dtype=int))



### Example: Initializing float values

1. **print**(numpy.matlib.eye(n=3,k=0,dtype=float))



# numpy.matlib.identity()

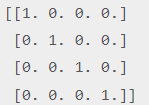
This function is used to return an identity matrix of the given size. An identity matrix is the one with diagonal elements initializes to 1 and all other elements to zero.

### Syntax: numpy.matlib.ones(size,dtype)

1. shape: It is the number of rows and columns in the resulting identity matrix.
2. dtype: It is the data type of the identity matrix.

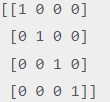
### Return: It returns an identity matrix of the specified size and specified data type.

**print**(numpy.matlib.identity(4))



### Example: Identity matrix with integer values

1. **print**(numpy.matlib.identity(4,int))



# numpy.matlib.ones()

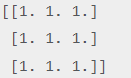
This function is used to return a new matrix with the values initialized to ones.

### Syntax: numpy.matlib.ones(shape,dtype,order)

1. shape: It is the Tuple defining the shape of the matrix.
2. dtype: It is the data type of the matrix.
3. order: It is the insertion order of the matrix.

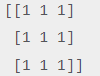
### Return: A matrix is returned with all the entries initialized to 1.

1. **print**(numpy.matlib.ones((3,3)))



### Example: initializing integer values

**print**(numpy.matlib.ones((3,3),int))



### Example: specifying Insertion order

**print**(numpy.matlib.ones((3,3),int,'C'))



# numpy.matlib.zeros()

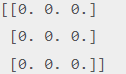
This function is used to return a new matrix with the values initialized to zeros.

### Syntax: numpy.matlib.zeros(shape,dtype,order)

1. shape: It is the Tuple defining the shape of the matrix.
2. dtype: It is the data type of the matrix.
3. order: It is the insertion order of the matrix.

### Return: A matrix with uninitialized entries is returned.

**print**(numpy.matlib.zeros((3,3)))



### Example: initializing integer values

**print**(numpy.matlib.zeros((3,3),int))



### Example: specifying Insertion order



# numpy.arrange()

It creates an array by using the evenly spaced values over the given interval. The interval mentioned is half opened i.e. [Start, Stop]).

### Syntax: numpy.arrange(start, stop, step, dtype)

1. start: The starting of an interval. The default is 0.
2. stop: represents the value at which the interval ends excluding this value.
3. step: The number by which the interval values change.
4. dtype: the data type of the numpy array items.

### Return: An array within the specified range is returned.

arr = np.arange(0,10,2,float) //[0. 2. 4. 6. 8.]

arr = np.arange(10,100,5,int) //[10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]

# numpy.asarray()

This function is used to create an array by using the existing data in the form of lists, or tuples. This function is useful in the scenario where we need to convert a python sequence into the numpy array object.

### Syntax: numpy.asarray(sequence, dtype = None, order = None)

1. shape: It is the Tuple defining the shape of the matrix.
2. dtype: It is the data type each item of the array
3. order: It is the insertion order of the array. The default is C.

### Return: An array with the equivalent values to the sequence is returned.

1. l=[1,2,3,4,5,6,7]
2. a = np.asarray(l) //[1 2 3 4 5 6 7]

### Example: Creating a numpy array from the Tuple

1. l=(1,2,3,4,5,6,7)
2. a = np.asarray(l) //[1 2 3 4 5 6 7]

### Example: creating a numpy array using more than one list

1. l=[[1,2,3,4,5,6,7],[8,9]]
2. a = np.asarray(l) //[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]

# numpy.frombuffer()

This function is used to create an array by using the specified buffer.

### Syntax: numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)

1. buffer: It represents an object that exposes a buffer interface.
2. dtype: It represents the data type of the returned data type array. The default value is 0.
3. count: It represents the length of the returned ndarray. The default value is -1.
4. offset: It represents the starting position to read from. The default value is 0.

### Return: The array version of the buffer is returned.

1. l = b'hello world'
2. a = np.frombuffer(l, dtype = "S1") //[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']

# numpy.fromiter()

This function is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

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

1. Iterable: It represents an iterable object.
2. dtype: It represents the data type of the resultant array items.
3. count: It represents the number of items to read from the buffer in the array.

### Return: An array created by using the iterable object is returned.

1. list = [0,2,4,6]
2. it = iter(list)
3. x = np.fromiter(it, dtype = float) //[0. 2. 4. 6.]

# numpy.linspace()

It is similar to the arrange function. However, it doesn?t allow us to specify the step size in the syntax.

Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size.

### Syntax: numpy.linspace(start, stop, num, endpoint, retstep, dtype)

1. start: It represents the starting value of the interval.
2. stop:It represents the stopping value of the interval.
3. num: The amount of evenly spaced samples over the interval to be generated. The default is 50.
4. endpoint: Its true value indicates that the stopping value is included in interval.
5. rettstep: This has to be a boolean value. Represents the steps and samples between the consecutive numbers.
6. dtype: It represents the data type of the array items.

### Return: An array within the specified range is returned.

1. arr = np.linspace(10, 20, 5)
2. **print**("The array over the given range is ",arr) //[10. 12.5 15. 17.5 20.]
3. arr = np.linspace(10, 20, 5, endpoint = False) //[10. 12. 14. 16. 18.]

# numpy.logspace()

It creates an array by using the numbers that are evenly separated on a log scale.

### Syntax: numpy.logspace(start, stop, num, endpoint, base, dtype)

1. start: It represents the starting value of the interval in the base.
2. stop:It represents the stopping value of the interval in the base.
3. num:The number of values between the range.
4. endpoint:It is a boolean type value. It makes the value represented by stop as the last value of the interval.
5. base:It represents the base of the log space.
6. dtype:It represents the data type of the array items.

### Return: An array within the specified range is returned.

arr = np.logspace(10, 20, num = 5, endpoint = True) //[1.00000000e+10 3.16227766e+12 1.00000000e+15 3.16227766e+17 1.00000000e+20]