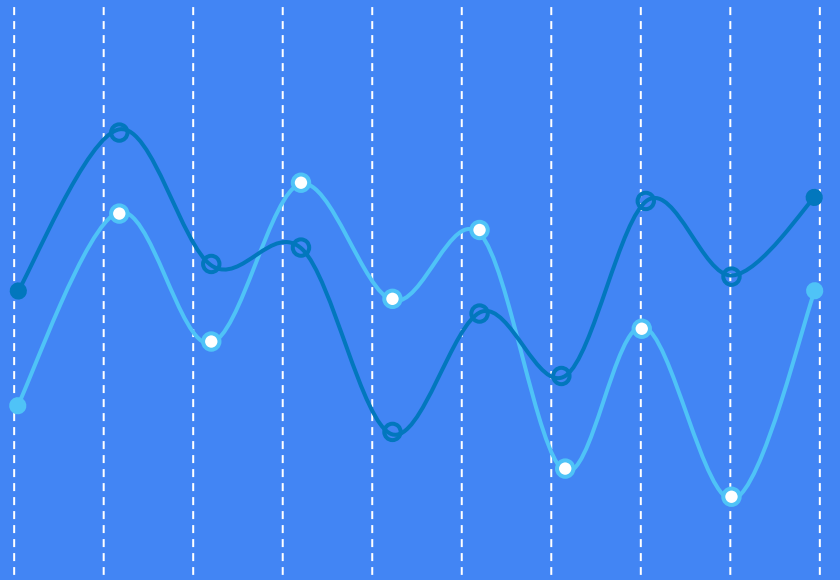
Certification Training Courses for Graduates and Professionals

www.signitivetech.com

NumPy: Numerical Python

Agenda

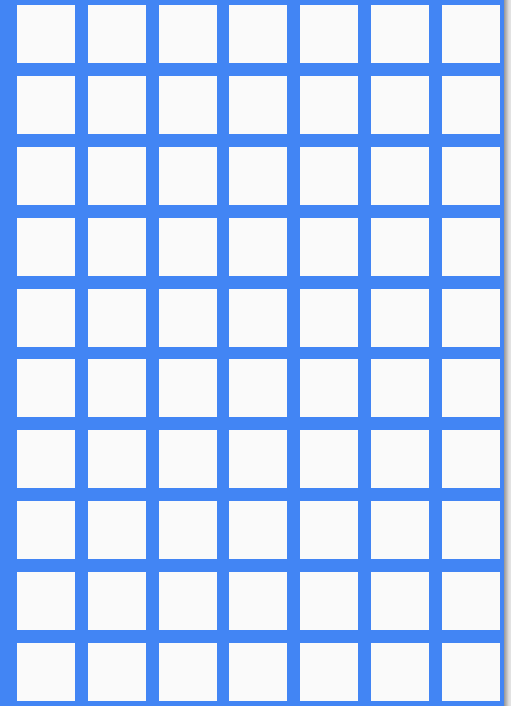
- Introduction
- Ndarray Objects
- Array Creation Routines
- Array from Numerical Ranges
- Indexing & Slicing
- Various Functions
- Array Manipulation
- More Stuffs with Arrays



Introduction

Introduction

- NumPy: NumPy stands for Numerical Python.
- It is a library consisting of multidimensional array objects and a collection of routines for processing of array.
- NumPy is used for mathematical and logical operations on arrays, operations related to linear algebra and random number generation, fourier transforms and routines for shape manipulation.
- NumPy is a Python extension to add support for large, multi-dimensional **arrays** and **matrices**, along with a large library of high-level mathematical **functions**.



Ndarray Objects

Ndarray

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.
- Ndarray: N-dimensional array type i.e ndarray describes the collection of items of the same type and items in the collection can be accessed using zero-based index.

Ndarray

import *numpy* as np

Syntax: np.array(object, dtype= None)

- ➔ object: Any Python nested sequence
- ➔ dtype: Desired data type of an array

```
n1 = np.array([1, 2, 3, 4])
print(n1)

n2 = np.array([1.1, 2.3, 3.3, 4.4])
print(n2)

n3 = np.array(['A', 'B', 'C', 'D'])
print(n3)
```

Output:

```
[1 2 3 4]    ← n1
[1.1  2.2  3.3  4.4] ← n2
['A'  'B'  'C'  'D']  ← n3
```

Ndarray

import *numpy* as np

As Array is a collection of Homogeneous Types of Elements, if any one element belongs to different data type, numpy makes all elements of same data types.
If One element is of float type and other are of integers, all elements will become of float type.
If One element is of str type, all elements will become of str type.

```
n1 = np.array([1.1, 2, 3, 4])
print(n1)

n2 = np.array(['A', 2.3, 3.3, 4])
print(n2)
```

Output:

```
[1.1  2.  3.  4.]          ← n1
['A'  '2.2'  '3.3'  '4']   ← n2
```

Types of Arrays

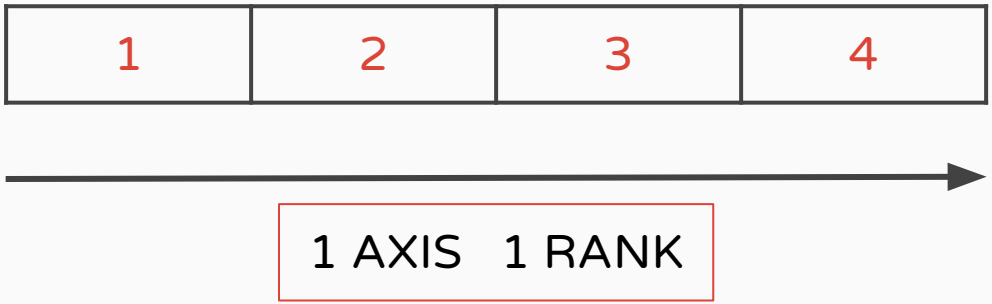
➔ **One dimensional:**

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

print(a)
```

Output: [1 2 3 4]

→ 1 axis 1 rank: x-axis



Interpretation:
Array a is a Horizontal Sequence of 4 Items

Types of Arrays

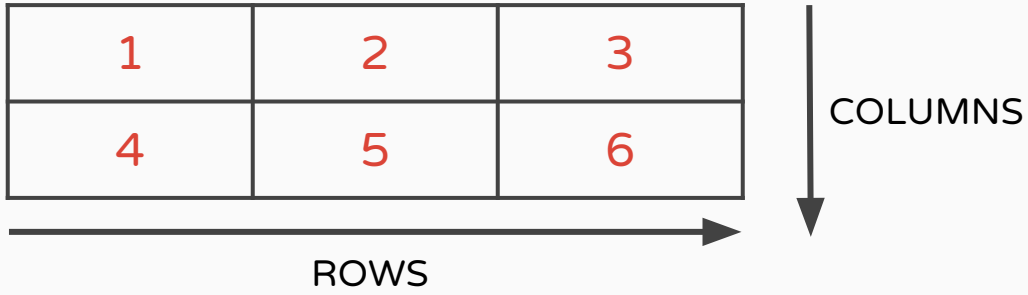
➔ **Two dimensional:**

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

Output:

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

→ 2 axes 2 ranks: x-axis, y-axis



Interpretation:
Array b contains 2 Rows and 3 Columns

Types of Arrays

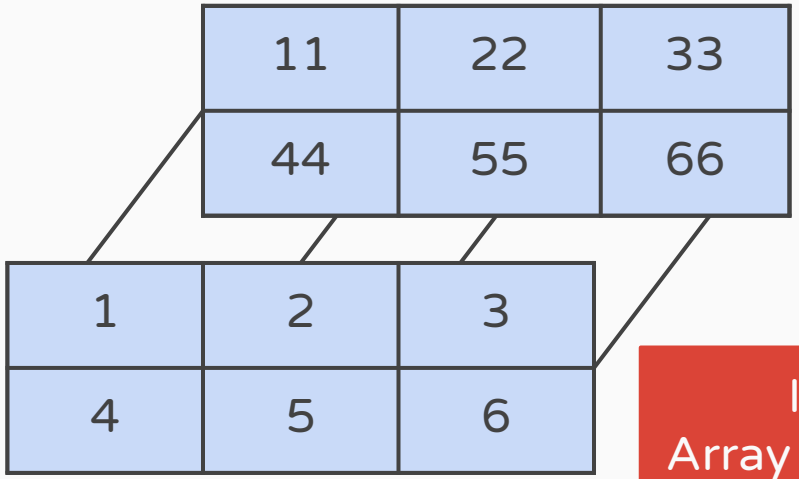
➔ **Three dimensional:**

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

Output:

```
[[[1 2 3]
  [4 5 6]]
[[11 22 33]
  [44 55 66]]]
```

→ 3 axes 3 ranks:
x-axis, y-axis, z-axis



Interpretation:
Array c contains 2 Arrays of 2 X 3 Dimensions

Ndarray: Attributes

n1 = np.array([1, 2, 3, 4]) → 1D Array

- **ndarray.shape**: Returns a tuple consisting of array dimensions or rank.
  - print(n1.shape) → Output: (4,)
- **ndarray.dtype**: Returns the type of the elements in the array.
  - print(n1.dtype) → Output: int32
- **ndarray.size**: Returns the total number of elements of the array. This is equal to the product of the elements of shape.
  - print(n1.size) → Output: 4
- **ndarray.ndim**: Returns the number of axes(dimensions) of the array.
  - print(n1.ndim) → Output: 1
- **ndarray.itemsize**: Returns the size in bytes of each element of the array.
  - print(n1.itemsize) → Output: 4
- **ndarray.nbytes**: Returns the total size of all elements i.e. size of array.
  - print(n1.nbytes) → Output: 16

Ndarray: Attributes

n1 = np.array([[1, 2, 3], [4, 5, 6]]) → 2D Array

- **ndarray.shape**: Returns a tuple consisting of array dimensions or rank.
  - print(n1.shape) → Output: (2, 3)
- **ndarray.dtype**: Returns the type of the elements in the array.
  - print(n1.dtype) → Output: int32
- **ndarray.size**: Returns the total number of elements of the array. This is equal to the product of the elements of shape.
  - print(n1.size) → Output: 6
- **ndarray.ndim**: Returns the number of axes(dimensions) of the array.
  - print(n1.ndim) → Output: 2
- **ndarray.itemsize**: Returns the size in bytes of each element of the array.
  - print(n1.itemsize) → Output: 4
- **ndarray.nbytes**: Returns the total size of all elements i.e. size of array.
  - print(n1.nbytes) → Output: 24

Ndarray: Attributes

n1 = np.array([[[1, 2, 3], [4, 5, 6]], [[11, 22, 33], [44, 55, 66]]]) → 3D Array

- **ndarray.shape**: Returns a tuple consisting of array dimensions or rank.
  - print(n1.shape) → Output: (2, 2, 3)
- **ndarray.dtype**: Returns the type of the elements in the array.
  - print(n1.dtype) → Output: int32
- **ndarray.size**: Returns the total number of elements of the array. This is equal to the product of the elements of shape.
  - print(n1.size) → Output: 12
- **ndarray.ndim**: Returns the number of axes(dimensions) of the array.
  - print(n1.ndim) → Output: 3
- **ndarray.itemsize**: Returns the size in bytes of each element of the array.
  - print(n1.itemsize) → Output: 4
- **ndarray.nbytes**: Returns the total size of all elements i.e. size of array.
  - print(n1.nbytes) → Output: 48

Ndarray

Access items of Array using for loop:

```
n1 = np.array([[1, 2, 3], [4, 5, 6]])
for i in n1:
    print(i)

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

```
n1 = np.array([[1, 2, 3], [4, 5, 6]])
for i in n1:
    for j in i:
        print(j, end=" ")
    print("")

Output:
1 2 3
4 5 6
```

Lists vs NumPy Arrays

Lists

- Arithmetic operations not possible.

```
l1 = [12, 14, 16, 18]
t1 = [2, 7, 8, 3]
print(l1/t1)
```

Output: TypeError: unsupported operand type(s) for /: 'list' and 'list'

- Dynamic appending items is possible.

```
l1 = [12, 14, 16, 18]
l1 = l1 + [4, 5]
print(l1)
```

Output: [12, 14, 16, 18, 4, 5]

Arrays

- Arithmetic operations are possible.

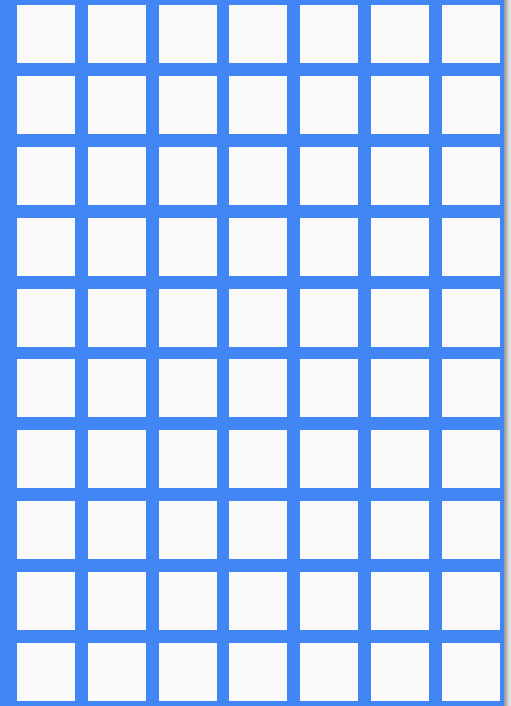
```
l1 = np.array([12, 14, 16, 18])
t1 = np.array([2, 7, 8, 3])
print(l1/t1)
```

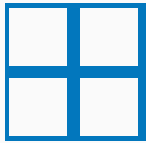
Output: [6. 2. 2. 6.]

- Dynamic appending items isn't possible.

```
l1 = np.array([12, 14, 16, 18])
l1 = l1 + [4, 5]
print(l1)
```

Output: ValueError: operands could not be broadcast together with shapes (4,) (2,)





Array Creation Routines

Array Creation Routines

- numpy.empty: Creates an uninitialized array of specified shape and data type. Elements will be random values.
  - **Syntax**: numpy.empty(shape, dtype)
  - shape: Shape of an empty array in int or tuple or list of int
  - dtype: Desired data type of elements

```
print(np.empty((2,2), dtype='int'))

[[5 0]
 [4 0]]
```

```
print(np.empty((2,2), dtype='float'))

[[1.00685167e-311 9.10805054e-315]
 [9.10805117e-315 9.34601642e-307]]
```

Array Creation Routines

- ➔ numpy.zeros: Returns a new array of specified size, filled with zeros.
  - **Syntax**: numpy.zeros(shape, dtype)
  - shape: Shape of an empty array in int or tuple or list of int
  - dtype: Desired data type of elements

```
print(np.zeros(2)) ← Output: [0. 0.]
```

```
print(np.zeros((2, 2))) ← Output:
                          [[0. 0.]
                           [0. 0.]]
```

```
print(np.zeros([2, 2])) ← Output:
                          [[0. 0.]
                           [0. 0.]]
```

Array Creation Routines

- numpy.ones: Returns a new array of specified size, filled with ones.
  - **Syntax**: numpy.ones(shape, dtype)
  - shape: Shape of an empty array in int or tuple or list of int
  - dtype: Desired data type of elements

```
print(np.ones(2)) ← Output: [1. 1.]
```

```
print(np.ones((2, 2))) ←  Output:
                          [[1. 1.]
                           [1. 1.]]
```

```
print(np.ones([2, 2])) ←  Output:
                          [[1. 1.]
                           [1. 1.]]
```

Array Creation Routines

- numpy.full: Returns a new array of specified size, filled with fill_value.
  - **Syntax**: numpy.full(shape, fill_value, dtype)
  - shape: Shape of an empty array in int or tuple or list of int
  - fill_value: Scalar value to be filled in array
  - dtype: Desired data type of elements

```
print(np.full((2, 2), 5,
dtype='float'))

[[5.0 5.0]
 [5.0 5.0]]
```

```
print(np.full([3, 3], 6))

[[6 6 6]
 [6 6 6]
 [6 6 6]]
```

Array Creation Routines

- numpy.identity: Returns the identity array. The identity array is a square array with ones on the main diagonal.
  - **Syntax**: numpy.identity(n, dtype)
  - n: int → Number of rows and columns in *n x n* output
  - dtype: Desired data type of elements

```
print(np.identity(3))

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

Array Creation Routines

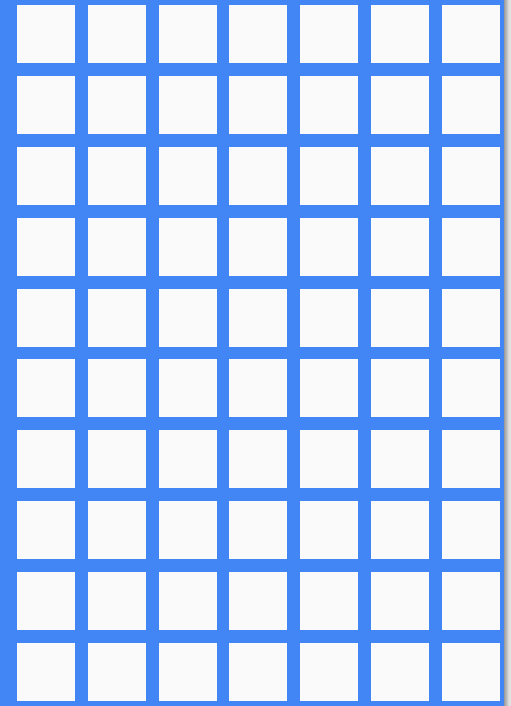
- numpy.eye: Returns a 2D array with ones on the diagonal and zeros elsewhere.
  - **Syntax**: numpy.eye(N, M=None, k=0, dtype)
  - N: Numbers of rows
  - M: Number of columns, if None, defaults to N
  - k: Index of the diagonal
  - dtype: Desired data type of elements

```
print(np.eye(3))

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

```
print(np.eye(3, k = 1))

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



Array from Numerical Ranges

Array from Numerical Ranges

- numpy.arange: Returns evenly spaced values within a given interval.
  - **Syntax**: numpy.arange([start,] stop, [step,] dtype = None)
  - start: Start of interval. Start is inclusive. The default start value is 0.
  - stop: End of interval. End is exclusive.
  - step: Spacing between values i.e. the distance between two adjacent values. The default step size is 1.
  - dtype: Desired data type of elements

```
print(np.arange(3))
```

```
[0 1 2]
```

```
print(np.arange(1, 20, 4))
```

```
[ 1  5  9 13 17]
```

Array from Numerical Ranges

- numpy.linspace: Returns evenly spaced numbers over a specified interval. Returns ***num*** evenly spaced samples, calculated over the interval.
  - **Syntax**: numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
  - start: Starting value of sequence. This is inclusive.
  - stop: The end value of sequence. This is exclusive.
  - num: Number of samples to generate. Default is 50.
  - endpoint: If True, stop is the last sample, otherwise stop is exclusive.
  - retstep: If True, return samples and step, where step is the spacing between samples.
  - dtype: Desired data type of elements

Array from Numerical Ranges

- numpy.linspace: Returns evenly spaced numbers over a specified interval. Returns ***num*** evenly spaced samples, calculated over the interval.
  - **Syntax**: numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

```
print(np.linspace(2, 10, num=6))

[ 2.   3.6  5.2  6.8  8.4 10. ]
```

```
print(np.linspace(2, 18, num=4, endpoint=False, retstep=True))

(array([ 2.,  6., 10., 14.]), 4.0)
```

Array from Numerical Ranges

- numpy.logspace: Returns numbers spaced evenly spaced on log scale.
  - **Syntax**: numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
  - start: Starting value of sequence. base**start.
  - stop: The end value of sequence. base**stop.
  - num: Number of samples to generate. Default is 50.
  - endpoint: If True, stop is the last sample, otherwise stop is exclusive.
  - base: The base of the log space. Default is 10.0
  - dtype: Desired data type of elements

Array from Numerical Ranges

- numpy.logspace: Returns numbers spaced evenly spaced on log scale.
  - **Syntax**: numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)

```
print(np.logspace(2, 4, num=4))

[  100.          464.15888336  2154.43469003 10000.        ]
```

```
print(np.logspace(2, 5, num=4, endpoint=False, base=2.0))

[ 4.         6.72717132 11.3137085  19.02731384]
```

Repeating Sequences

Repeating Sequences: The numpy.tile will repeat a whole list or array ***n*** times. Whereas, the numpy.repeat repeats each item ***n*** times.

```
a = [1, 2, 3]
print("Tile:", np.tile(a, 2))
```

Output:
Title: [1 2 3 1 2 3]

```
a = [1, 2, 3]
print("Repeat:", np.repeat(a, 2))
```

Output:
Repeat: [1 1 2 2 3 3]

Arrays from Random Number Generation

The random module provides nice functions to generate random numbers.

```
# Random numbers between 0 and 1 of shape 2, 2
print(np.random.rand(2, 2))
```

```
[[0.20968946  0.48999691]
 [0.16162075  0.72010348]]
```

```
# Random numbers between 0 and 1 of shape 5
print(np.random.rand(5))
```

```
[0.11501305 0.55973855 0.32264374 0.97291222
0.49677243]
```

Arrays from Random Number Generation

The random module provides nice functions to generate random numbers.

```
# Random integers between 0 and 10 of shape 2, 5
print(np.random.randint(0, 10, size=[2, 5]))
```

```
[[9 5 5 2 2]
 [5 5 8 2 2]]
```

```
# Random integers between 0 and 10 of shape 5
print(np.random.randint(0, 10, size=[5]))
```

```
[9 5 4 2 2]
```

Arrays from Random Number Generation

The random module provides nice functions to generate random numbers.

```
#Random float values between 2, 8 of shape 5
print(np.random.uniform(2, 8, size=[5]))
```

```
[6.59508273 3.75125558 6.95702465
6.83985282 3.55061172]
```

```
#Random float values between 2, 8 of shape 2, 2
print(np.random.uniform(2, 8, size=[2, 2]))
```

```
[[ 7.83464456  5.83850086]
 [5.06855626  4.72960059]]
```

Arrays from Random Number Generation

The random module provides nice functions to generate random numbers.

```
# Random integers between 0 and 15 of shape 2, 2, 3
print(np.random.randint(0, 15, size=[2, 2, 3]))
```

```
[[[13  4  0]
  [12  8  6]]

 [[ 8  9  8]
  [ 8  2  7]]]
```

Arrays from Random Number Generation

The random module provides nice functions to generate random numbers.

```
# Random float values between 0 and 4 of
shape 2, 2, 3
print(np.random.uniform(0, 4, size=[2, 2, 3]))
```

```
[[[2.27866091 3.94003556 2.20064253]
  [0.19282226 2.81693994 2.49787956]]

 [[3.77347503 2.9234183  3.98247793]
  [3.38088075 1.49626326 3.1825434 ]]]
```

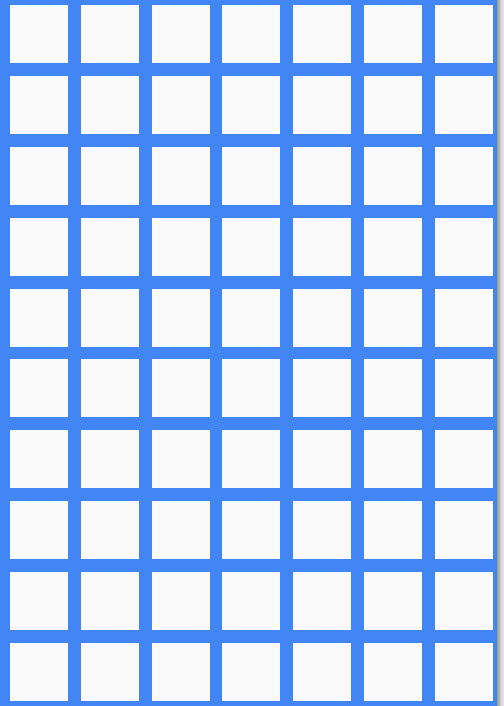
Arrays from Random Number Generation

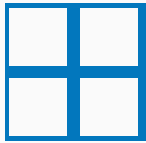
The random module provides nice functions to generate random numbers.

```
# Random values between 0 and 1 of shape 2, 3, 2
print(np.random.random(size=[2,3,2]))
```

```
[[[0.91982951 0.19839739]
  [0.56538633 0.98950531]
  [0.47856919 0.47385302]]

 [[0.00343517 0.11322972]
  [0.42218032 0.46444301]
  [0.8663744  0.23991606]]]
```





Indexing & Slicing

Indexing and Slicing

➔ Contents of ndarray object can be accessed and modified by indexing or slicing. Items in ndarray object follows zero-based index. Three types of indexing methods are available – field access, basic slicing and advanced indexing.

Field Access(Indexing):

```
n = np.arange(15)
print(n) → Output: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

print(n[2]) → Output: 2

print(n[-5]) → Output: 10
```

Indexing and Slicing

Slicing:

```
n = np.arange(15)
print(n) → Output: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
print(n[2: 7: 3]) → Output: [2 5]
print(n[-2: -10: -1]) → Output: [13 12 11 10 9 8 7 6]
print(n[5:]) → Output: [5 6 7 8 9 10 11 12 13 14]
print(n[5: 8]) → Output: [5 6 7]
print(n[:6]) → Output: [0 1 2 3 4 5]
print(n[: -6]) → Output: [0 1 2 3 4 5 6 7 8]
print(n[: -6: 2]) → Output: [0 2 4 6 8]
print(n[0::4]) → Output: [0 4 8 12]
```

Indexing and Slicing

Slicing:

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
print(a[1:])
```

```
[[ 5  6  7  8  9]
 [10 11 12 13 14]]
```

← Items from 2nd row onwards

```
print(a[1: 2])
```

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

← Items from 2nd row only as 2 is exclusive

```
print(a[0: :2])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]]
```

← Items from 1st row and 3rd row, as step is 2

Indexing and Slicing

Slicing:

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
print(a[: ,1:])
```

```
[[ 2  3  4]
 [ 7  8  9]
 [12 13 14]]
```

← Items from 2nd column onwards

```
print(a[: ,1::2])
```

```
[[ 1  3]
 [ 6  8]
 [11 13]]
```

← Items from 2nd column onwards step by 2

```
print(a[: , :])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

← All items

Indexing and Slicing

Slicing:

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
print(a[2, 2])
```

```
12
```

← Item from 3rd row and 3rd column

```
print(a[2, 2:])
```

```
[12 13 14]
```

← Items 3rd row and 3rd column onwards

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

```
[[ 0  2  4]
 [10 12 14]]
```

← Items rows and columns alternate by 2

Indexing and Slicing

Slicing:

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
print(a[[0, 1, 2], [0, 1, 3]])
```

```
[0  6 13]
```

← Items from positions (0, 0), (1, 1), (2, 3)

```
print(a[1:3, 0:5])
```

```
[[ 5  6  7  8  9]
 [10 11 12 13 14]]
```

← Items 2nd and 3rd rows

```
print(a[::2, :])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]]
```

← Items from 1st and 3rd rows as step is 2

Indexing and Slicing

Advanced Indexing:

Boolean Indexing

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
print(a[a > 5])
```

```
[ 6  7  8  9 10 11 12 13 14]
```

← Items whose values are greater than 5

```
print(a[(a > 1) & (a < 9)])
```

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

← Items whose values are greater than 1 and less than 9

```
print(a[a%2==0])
```

```
[ 0  2  4  6  8 10 12 14]
```

← Items whose mod value is zero

Array Broadcasting

Broadcasting: Broadcasting actually refers to Arithmetic Operations to be performed on arrays on corresponding elements. If the dimensions of two arrays are different, element-to-element operations is not possible.

```
a = np.array([1, 2, 3, 4])
b = np.array([1, 2, 3, 4])
c = a + b
print(c)
```

```
[2 4 6 8]
```

```
a = np.array([1, 2, 3, 4])
b = np.array([1, 2, 3])
c = a + b
print(c)
```

```
ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

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

```
[6 12 18 24]
```

Array Broadcasting

Broadcasting:

```
n = np.arange(15)
a = n.reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
c = a + [1, 2, 3, 4, 5]
print(c)
# it adds each element from list into each element from each row of array
```

```
[[ 1  3  5  7  9]
 [ 6  8  10  12  14]
 [11 13 15 17 19]]
```

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

```
[6 12 18 24]
```

Iterating Over Array

numpy.nditer: It is an efficient multidimensional iterator object using which it is possible to iterate over an array.

Order: 'C' → row-major or 'F' → column-major

```
a = np.array([[1, 2], [3, 4]])
print(a)
for x in np.nditer(a):
    print(x, end=" ")
```

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

1 2 3 4
```

```
a = np.array([[1, 2], [3, 4]])
print(a)
for x in np.nditer(a, order='C'):
    print(x, end=" ")
```

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

1 2 3 4
```

```
a = np.array([[1, 2], [3, 4]])
print(a)
for x in np.nditer(a, order='F'):
    print(x, end=" ")
```

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

1 3 2 4
```

3D Array Slicing & Indexing

```
c = np.array([[[1, 2, 3], [4, 5, 6]],
              [[11, 22, 33], [44, 55, 66]],
              [[7, 8, 9], [77, 88, 99]]])
print(c)
```

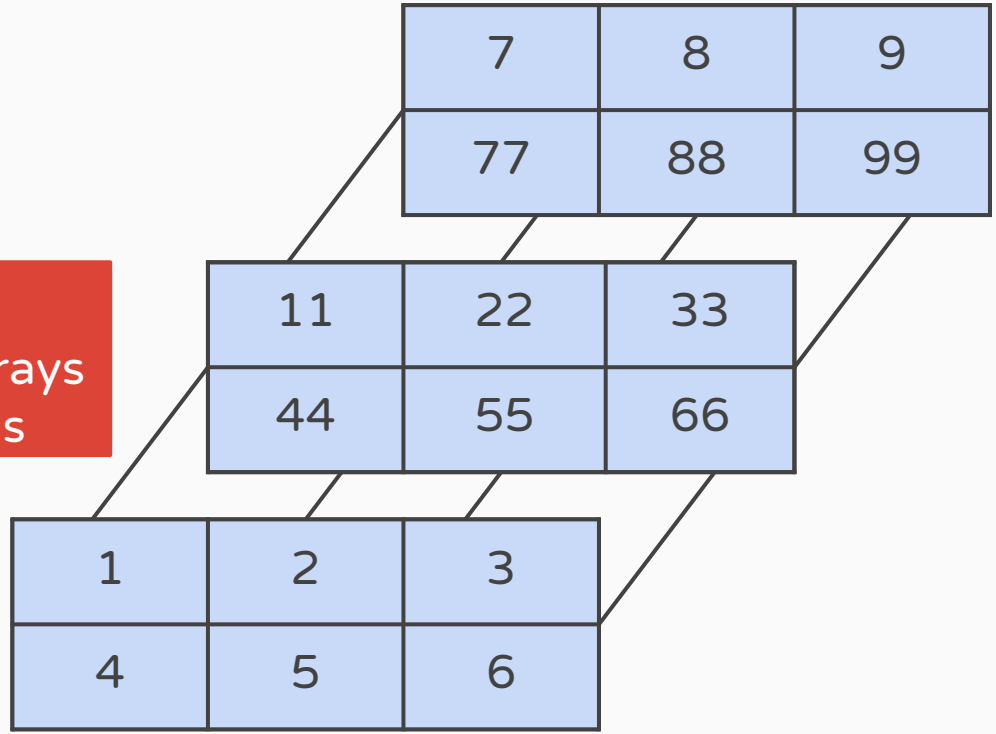
Output:

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

 [[11 22 33]
  [44 55 66]]

 [[ 7  8  9]
  [77 88 99]]]
```

Interpretation:
Array c contains 3 Arrays of 2 X 3 Dimensions



3D Array Slicing & Indexing

Slicing:

c[0]

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

# 1st Element
```

c[-1]

```
array([[ 7,  8,  9],
       [77, 88, 99]])

# Last Element
```

c[0:, 0]

```
array([[ 1,  2,  3],
       [11, 22, 33],
       [ 7,  8,  9]])
# 1st Row from each Array
```

c[0:, 1]

```
array([[ 4,  5,  6],
       [44, 55, 66],
       [77, 88, 99]])
# 2nd Row from each Array
```

c[1:, 1]

```
array([[44, 55, 66],
       [77, 88, 99]])
# 1st Row from 2nd and 3rd
Array
```

3D Array Slicing & Indexing

Slicing:

```
c[0:, 0:, 0]

array([[ 1,  4],
       [11, 44],
       [ 7, 77]])

# 1st Column from each Array
```

```
c[0:, 0:, -1]

array([[ 3,  6],
       [33, 66],
       [ 9, 99]])

# Last Column from each Array
```

```
c[0:, 0:, 1:]

array([[[ 2,  3],
        [ 5,  6]],

       [[22, 33],
        [55, 66]],

       [[ 8,  9],
        [88, 99]]])

# Items from 2nd and 3rd columns from each Array
```

Various Functions

Mathematical Functions

Rounding Functions:

numpy.around(data, decimals): Returns the value rounded to the desired precision.

```
a = np.array([-2.333, 45.666, 3.54, -14.344, 56.7878])
np.around(a)
```

```
array([ -2.,  46.,   4., -14.,  57.])
```

Mathematical Functions

Rounding Functions:

numpy.floor(data): Returns the largest integer not greater than input data.

```
a = np.array([-2.333, 45.666, 3.54, -14.344, 56.7878])
np.floor(a)
```

```
array([ -3.,  45.,   3., -15.,  56.])
```

numpy.ceil(data): Returns the smallest integer not less than input data.

```
a = np.array([-2.333, 45.666, 3.54, -14.344, 56.7878])
np.ceil(a)
```

```
array([ -2.,  46.,   4., -14.,  57.])
```

Statistical Functions

numpy.amin(data, axis): Returns the minimum from the elements in the given array along the specified axis. 0 → Columns & 1 → Rows

numpy.amax(data, axis): Returns the maximum from the elements in the given array along the specified axis. 0 → Columns & 1 → Rows

a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]]) print(a)	print(np.amin(a)) print(np.amax(a))	print(np.amin(a, 0)) print(np.amax(a, 0))	print(np.amin(a, 1)) print(np.amax(a, 1))
[[1 3 6] [9 11 -5] [3 -2 15]]	-5 ← Minimum 15 ← Maximum	[1 -2 -5] [9 11 15]	[1 -5 -2] [6 11 15]

Statistical Functions

numpy.ptp(data): Returns the range (maximum - minimum) of values along an axis. 0 → Columns & 1 → Rows

a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]]) print(a)	np.ptp(a)	np.ptp(a, 0)	np.ptp(a, 1)
[[1 3 6] [9 11 -5] [3 -2 15]]	20 (Max - Min) → (15 - (-5))	array([8, 13, 20]) → Column-wise	array([5, 16, 17]) → Row-wise

Statistical Functions

numpy.percentile(data, *percentile*): Percentile is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. The percentile should be between 0 - 100

a = np.array([[20, 30, 60], [90, 10, 50], [70, 40, 80]]) print(a)	np.percentile(a, 50)	print(np.percentile(a, 50, 0)) print(np.percentile(a, 50, 1))
[[20 30 60] [90 10 50] [70 40 80]]	50.0 50th Percentile	[70. 30. 60.] → Columns [30. 50. 70.] → Rows 0 → Columns & 1 → Rows

Sort the data first in ascending order: 10 20 30 40 50 60 70 80 90

Statistical Functions

numpy.median(data): Returns the middle value between the input array which separates the data in two parts equally having lower half and higher half.

0 → Columns & 1 → Rows

a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]]) print(a)	np.median(a)	np.median(a, 0)	np.median(a, 1)
[[1 3 6] [9 11 -5] [3 -2 15]]	3.0 → Median	array([3., 3., 6.]) → Column-wise	array([3., 9., 3.]) → Row-wise

Statistical Functions

numpy.mean(data): Returns mean which is the sum of all elements along the axis divided by the number of elements.

0 → Columns & 1 → Rows

a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]]) print(a)	np.mean(a)	np.mean(a, 0)	np.mean(a, 1)
[[1 3 6] [9 11 -5] [3 -2 15]]	4.55 → Mean	array([4.33, 4., 5.33])	array([3.33, 5., 5.33])

Statistical Functions

numpy.std(data): Returns the standard deviation. Standard Deviation is the square root of the average the squared deviations from the mean.

Formula: std = sqrt(mean(abs(x - x.mean()**2)))

numpy.var(data): Returns the variance which is square of standard deviation.

Formula: var = mean(abs(x - x.mean()**2))

a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]]) print(a)	np.std(a)	np.var(a)
[[1 3 6] [9 11 -5] [3 -2 15]]	6.00205 → Standard Deviation	36.0246 → Variance

Sort, Search, Counting Functions

numpy.sort(data, axis, order): Returns a sorted copy of the input array along the specified ***axis*** if given and if ***order*** is given, array will be sorted over the fields. 0 → Columns & 1 → Rows

```
a = np.array([[1, 3, 6], [9, 11, -5], [3, -2, 15]])
print(a)
```

```
[[ 1  3  6]
 [ 9 11 -5]
 [ 3 -2 15]]
```

```
print(np.sort(a))
```

```
[[ 1  3  6]
 [-5  9 11]
 [-2  3 15]]
```

```
print(np.sort(a, 0))
```

```
[[ 1 -2 -5]
 [ 3  3  6]
 [ 9 11 15]]
```

```
print(np.sort(a, 1))
```

```
[[ 1  3  6]
 [-5  9 11]
 [-2  3 15]]
```

Sort, Search, Counting Functions

numpy.argsort(data): Performs a sort along the given axis and returns the array of indices of data.

numpy.argmax(data, axis): Returns the indices of maximum elements along the given axis.

numpy.argmin(data, axis): Returns the indices of minimum elements along the given axis.

a = np.array([-2, 8, 4, 0, 1]); print(a)	print(np.argsort(a))	print(np.argmax(a))	print(np.argmin(a))
[-2 8 4 0 1]	[0 3 4 2 1]	1 → 1st position of the array contains maximum item: 8	0 → 0th position of the array contains minimum item: -2

Sort, Search, Counting Functions

numpy.nonzero(data): Returns the indices of non-zero elements in the array.

numpy.where(condition): Returns the indices of elements in the array where the given condition is satisfied.

numpy.extract(condition, data): Returns the elements that satisfy the given condition.

a = np.array([-2, 8, 4, 0, 1]); print(a)	np.nonzero(a)	np.where(a > 1)	print(np.extract(a > 1, a))
[-2 8 4 0 1]	(array([0, 1, 2, 4], dtype=int64),)	(array([1, 2], dtype=int64),) → 1st & 2nd positioned elements satisfy the condition	[8 4]

More Useful Functions

Use numpy.clip to cap the numbers within a given cutoff range. All number lesser than the lower limit will be replaced by the lower limit. Same applies to the upper limit also.

```
x = np.arange(10)
print(x)
print(np.clip(x, 3, 8))
```

```
[0 1 2 3 4 5 6 7 8 9]

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

Array Manipulation

Array Manipulation

Changing shape:

ndarray.reshape(shape): Gives a new shape to array without changing the data.

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

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

```
print(a.reshape(4, 2))
```

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

Array Manipulation

Changing shape:

ndarray.reshape(shape): We can convert 1D Array into 2D Array having each element from 1D array will treat as a one Row in 2D array.

To convert this type array we generally we reshape function like this:

np.reshape (*array_name*, (-1, 1))

```
a = np.arange(5)
print(a)
```

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

```
print(a.reshape(5, 1))
```

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

```
print(a.reshape(-1, 1))
```

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

Array Manipulation

Changing shape:

ndarray.flat[index]: Returns a 1D iterator over the array.

ndarray.flatten(): Returns a copy of array changing into one dimension.

ndarray.ravel(): Returns a flattened 1D array. No copy will be made.

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

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

```
a.flat[3]
```

4 → 4th positioned item

```
print(a.flatten())
```

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

```
print(a.ravel())
```

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

Array Manipulation

Transpose:

np.transpose(data): Transpose the array changing rows to columns and vice-versa.

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

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

```
print(np.transpose(a))
```

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

Array Manipulation

Splitting Arrays:

numpy.split(data, sections): Splits the array into subarrays (given sections) along a specified axis.

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
print(np.split(a, 3))
```

```
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
```

Array Manipulation

Manipulation:

numpy.resize(data, shape): Returns a new array with the specified size. If new array is greater than original array, the repeated copies of entries in the original array are added.

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

```
[0 1 2 3 4 5 6]
```

```
a = np.resize(a, (3, 3))
print(a)
```

```
[[0 1 2]
 [3 4 5]
 [6 0 1]]
```

Array Manipulation

Manipulation:

numpy.append(array, values, axis): Adds values at the end of the input array.

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

```
[0 1 2 3 4 5 6]
```

```
a = np.append(a, (3, 3))
print(a)
```

```
[0 1 2 3 4 5 6 3 3]
```

Array Manipulation

Manipulation:

numpy.delete(array, index, axis): Deletes items from array.

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

```
[0 1 2 3 4 5 6]
```

```
a = np.delete(a, 3)
print(a)
```

```
[0 1 2 4 5 6]
```

Array Manipulation

Manipulation:

numpy.insert(array, index, values, axis): Insert values in the input array along the given axis and before the given index.

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

```
[0 1 2 3 4 5 6]
```

```
a = np.insert(a, 3, [8, 8])
print(a)
```

```
[0 1 2 8 8 3 4 5 6]
```

Array Manipulation

Manipulation:

numpy.hstack((tuple_of_arrays)): Stack arrays in sequence horizontally i.e. column wise.

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

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

```
c = np.hstack((a,b))
print(c)
```

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

Array Manipulation

Manipulation:

numpy.vstack((tuple_of_arrays)): Stack arrays in sequence vertically i.e. row wise.

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

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

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

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

Array Manipulation

Manipulation:

numpy.flip(array, axis): Reverse the order of elements in an array along the given axis

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

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

```
a = np.flip(a, axis=0)
print(a)
```

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

```
a = np.flip(a, axis=1)
print(a)
```

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

Array Manipulation

Manipulation:

numpy.flipud(array): Flips the given array in up down direction.

numpy.fliplr(array): Flips the given array in left right direction.

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

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

```
a = np.flipud(a)
print(a)
```

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

```
a = np.fliplr(a)
print(a)
```

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

Array Manipulation

Manipulation:

numpy.put(array, [index], [values]): Replaces specified elements of an array with given values.

```
a = np.arange(6,12)
a = a.reshape(2,3)
print(a)
```

```
[[ 6  7  8]
 [ 9 10 11]]
```

```
np.put(a, [0, 5], [44, 55])
print(a)
```

```
[[44  7  8]
 [ 9 10 55]]
```

Array Manipulation

Manipulation:

numpy.place(array, mask/condition, [values]): Change elements of an array based on conditional and input values.

```
a = np.arange(6,12)
a = a.reshape(2,3)
print(a)
```

```
[[ 6  7  8]
 [ 9 10 11]]
```

```
np.place(a, a > 8, [11, 22, 33])
print(a)
```

```
[[ 6  7  8]
 [11 22 33]]
```

Array Manipulation

Manipulation:

numpy.take(array, [indices]): Take elements from an array present at given indices.

```
a = np.arange(11,31)
a = a.reshape(5,4)
print(a)
```

```
[[11 12 13 14]
 [15 16 17 18]
 [19 20 21 22]
 [23 24 25 26]
 [27 28 29 30]]
```

```
b = np.take(a, [6,9,11])
print(b)
```

```
[17 20 22]
```

Array Manipulation

Manipulation:

numpy.unique(array, return_index, return_inverse, return_counts):
Returns an array of unique elements in the input array.

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

```
[2 2 4 2 9 9 2 5 6 7 2 2 8 9 9 2 3 1 2 3]
```

```
print(np.unique(a))
```

```
[1 2 3 4 5 6 7 8 9] → Unique Array
```

Array Manipulation

Manipulation:

numpy.unique(array, return_index, return_inverse, return_counts):
Returns an array of unique elements in the input array.

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

```
[2 2 4 2 9 9 2 5 6 7 2 2 8 9 9 2 3 1 2 3]
```

```
np.unique(a, return_counts= True)
```

```
(array([1, 2, 3, 4, 5, 6, 7, 8, 9]), array([1, 8, 2, 1, 1, 1, 1, 1, 4],
dtype=int64))
```

→ Returns Count of each unique item

Array Manipulation

Manipulation:

numpy.unique(array, return_index, return_inverse, return_counts):
Returns an array of unique elements in the input array.

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

```
[2 2 4 2 9 9 2 5 6 7 2 2 8 9 9 2 3 1 2 3]
```

```
np.unique(a, return_counts= True, return_index= True)
```

```
(array([1, 2, 3, 4, 5, 6, 7, 8, 9]), array([17,  0, 16,  2,  7,  8,  9, 12,  4],
dtype=int64), array([1, 8, 2, 1, 1, 1, 1, 1, 4], dtype=int64))
```

→ Returns count and index of each unique item

Arithmetic Functions

NumPy provides arithmetic operations such as numpy.add(), numpy.subtract(), numpy.multiply(), numpy.divide()

```
a = np.array([2, 4, 6]); print(a)
b = np.array([1, 3, 7]); print(b)
```

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

```
np.add(a, b)
```

```
array([3, 7, 13])
```

```
np.subtract(a, b)
```

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

```
np.multiply(a, b)
```

```
array([2, 12, 42])
```

```
np.divide(a, b)
```

```
array([2, 1.33, 0.85])
```

Arithmetic Functions

NumPy provides arithmetic operations such as numpy.reciprocal(), numpy.mod(), numpy.remainder(), numpy.power()

```
a = np.array([2, 4, 6]); print(a)
b = np.array([1, 3, 7]); print(b)
```

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

```
np.mod(a, b)
```

```
array([0, 1, 6])
```

```
np.remainder(a, b)
```

```
array([0, 1, 6])
```

```
np.power(a, b)
```

```
array([2, 64, 279936])
```

```
np.reciprocal(a, dtype='float')
```

```
array([0.5, 0.25, 0.16])
```

Arithmetic Functions

NumPy provides arithmetic operations such as numpy.sqrt(), numpy.log(), numpy.sin(), numpy.cos()

```
a = np.array([2, 4, 6]); print(a)
```

```
[2 4 6]
```

```
np.sqrt(a)
```

```
array([1.41421356, 2. , 2.44948974])
```

```
np.sin(a)
```

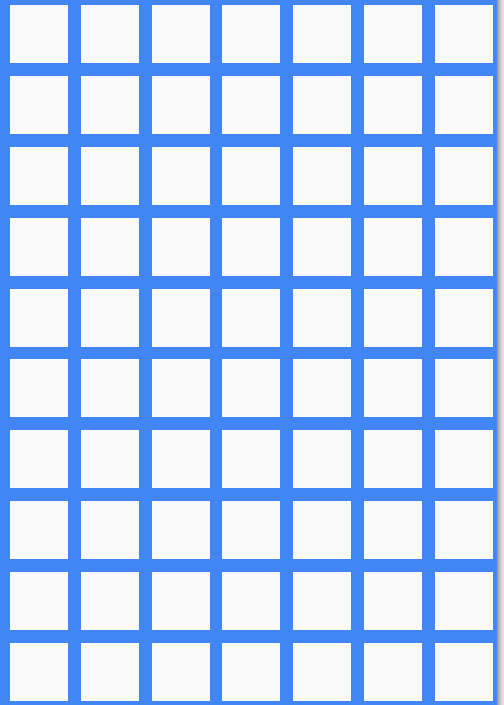
```
array([ 0.90929743, -0.7568025 , -0.2794155 ])
```

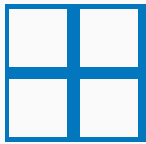
```
np.log(a)
```

```
array([0.69314718, 1.38629436, 1.79175947])
```

```
np.cos(a)
```

```
array([-0.41614684, -0.65364362, 0.96017029])
```





More Stuffs with Arrays

Linear Algebra

numpy.dot(array1, array2): Returns the dot product of two arrays.

```
a = np.array([2, 4, 6]); print(a)
b = np.array([1, 3, 7]); print(b)
```

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

```
print(np.dot(a, b))
```

```
56
```

→ 2*1 + 4*3 + 6*7

```
p = np.array([[1, 2], [3, 4]]); print(p)
q = np.array([[5, 6], [7, 8]]); print(q)
```

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

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

```
print(np.dot(p, q))
```

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

→ [1*5+2*7, 1*6+2*8],
[3*5+4*7, 3*6+4*8]

Linear Algebra

numpy.vdot(array1, array2): Returns the dot product of two vectors.

```
a = np.array([2, 4, 6]); print(a)
b = np.array([1, 3, 7]); print(b)
```

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

```
print(np.vdot(a, b))
```

```
56
```

→ 2*1 + 4*3 + 6*7

```
p = np.array([[1, 2], [3, 4]]); print(p)
q = np.array([[5, 6], [7, 8]]); print(q)
```

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

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

```
print(np.vdot(p, q))
```

```
70
```

→ 1*5 + 2*7 + 3*6 + 4*8

Linear Algebra

numpy.inner(array1, array2): Returns the inner product of vectors.

```
a = np.array([2, 4, 6]); print(a)
b = np.array([1, 3, 7]); print(b)
```

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

```
print(np.inner(a, b))
```

```
56
```

→ 2*1 + 4*3 + 6*7

```
p = np.array([[1, 2], [3, 4]]); print(p)
q = np.array([[5, 6], [7, 8]]); print(q)
```

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

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

```
print(np.inner(p , q))
```

```
[[17 23]
 [39 53]]
```

→ [1*5+2*6, 1*7+2*8],
[3*5+4*6, 3*7+4*8]

Linear Algebra

numpy.linalg.det(array): Returns the determinant of 2D array.

```
q = np.array([[5, 6], [7, 8]]); print(q)
```

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

```
np.linalg.det(q)
```

```
-2.00
```

numpy.linalg.inv(array): Returns the inverse of an array. The product of original array and inverse array returns an identity array.

```
q = np.array([[5, 6], [7, 8]])
print(q)
```

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

```
print(np.linalg.inv(q))
```

```
[[-4.   3. ]
 [ 3.5 -2.5]]
```

```
print(np.dot(q, np.linalg.inv(q)))
```

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

Linear Algebra

numpy.linalg.solve(array): Returns the solution of linear equations.

$2x + 3y - 6z = 24$

$x - 17y + 11z = 37$

$7x - y - 9z = 54$

```
p = np.array([[2, 3, -6], [1, -17, 11], [7, -1, -9]])
q = np.array([24, 37, 54])
```

→ Creates 1st array containing coefficients of x, y and z from all three equations and 2nd array containing equivalent values of each equations.

```
print(np.linalg.solve(p, q))
```

```
[-11.7295082  -11.80327869
-13.81147541]
```

→ Returns:
x = -11.72
y = -11.80
z = -13.81

Sequence of Dates

Using numpy.arange and numpy.datetime64, we can create the sequence of dates.

```
dates = np.arange(np.datetime64('2018-04-01'), np.datetime64('2018-04-10'))
print(dates)
```

```
['2018-04-01' '2018-04-02' '2018-04-03' '2018-04-04' '2018-04-05'
 '2018-04-06' '2018-04-07' '2018-04-08' '2018-04-09']
```

```
#Check if its a business day: For Saturday & Sunday it will return False
print(np.is_busday(dates))
```

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

Reverse the rows and the whole array

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]); print(a)
```

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

```
print(a[: : -1, ]) #Reverse only rows
```

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

```
print(a[::-1,::-1]) #Reverse rows & columns
```

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

Most Used Numpy Functions

- numpy.arange
- numpy.linspace
- numpy.reshape
- numpy.random.randint, numpy.random.uniform, numpy.random.rand, numpy.random.random
- numpy.flatten
- numpy.mean, numpy.median, numpy.argmin, numpy.argmax, numpy.std, numpy.var, numpy.percentile, numpy.corrcoef
- numpy.power, numpy.around, numpy.round, numpy.sqrt, numpy.square

End

Thanks!

Signitive Technologies

Sneh Nagar, Behind ICICI Bank,
Chhatrapati Square, Nagpur 15

Landmark:
Bharat Petrol Pump
Chhatrapati Square

Contact: 9011033776
www.signitivetech.com

“Keep Learning, Happy Learning”

Best Luck!

Have a Happy Future