NumPy

NumPy, which stands for Numerical Python, is a powerful library in Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these elements. NumPy is a fundamental package for scientific computing with Python and is widely used in various fields such as physics, engineering, machine learning, and data science.

Key features of NumPy include:

1. **Multidimensional Arrays:** NumPy provides a powerful `ndarray` object, which is a flexible, homogeneous array capable of holding elements of any data type. This array can be one-dimensional, two-dimensional, or even multi-dimensional.
2. **Element-Wise Operations:** NumPy enables efficient element-wise operations on arrays. Mathematical, logical, and statistical operations can be performed on entire arrays without the need for explicit loops, making it computationally efficient.
3. **Broadcasting:** NumPy allows operations between arrays of different shapes and sizes through a mechanism known as broadcasting. This simplifies operations on arrays and makes the code more concise.
4. **Linear Algebra Operations:** NumPy provides a variety of linear algebra functions, including matrix multiplication, eigenvalue decomposition, and singular value decomposition. These functions are crucial for numerical simulations and data analysis.
5. **Random Number Generation:** NumPy includes a powerful random module for generating random numbers. This is often used in simulations, statistical modeling, and various machine learning applications.
6. **Integration with Other Libraries:** NumPy seamlessly integrates with other scientific computing libraries in Python, such as SciPy (Scientific Python) and Matplotlib (plotting library).

NumPy Performance

NumPy is a fundamental library in the Python ecosystem, and its efficient array operations make it an essential tool for numerical computing and scientific applications.

NumPy is well-known for its performance, and several factors contribute to its efficiency in numerical and scientific computing:

1. **Array Operations:** NumPy performs array operations in compiled C and Fortran code, making these operations significantly faster than equivalent operations in pure Python. This is particularly important when dealing with large datasets or matrices.
2. **Vectorization:** NumPy encourages vectorized operations, where operations are applied to entire arrays rather than individual elements. These vectorized operations are implemented in highly optimized C or Fortran code, leading to faster execution compared to equivalent Python code with loops.
3. **Memory Layout:** NumPy uses a contiguous block of memory to store arrays. This memory layout allows for efficient access and manipulation of array elements, especially during operations involving contiguous memory access patterns.
4. **Data Types and Memory Efficiency:** NumPy arrays are homogeneous, meaning all elements in an array must have the same data type. This allows for efficient storage and retrieval of elements. Additionally, NumPy arrays have a smaller memory footprint compared to Python lists, reducing memory overhead.
5. **Broadcasting:** NumPy's broadcasting capabilities enable efficient element-wise operations on arrays of different shapes and sizes without the need for explicit loops. This reduces the amount of code and enhances performance.
6. **Caching:** NumPy operations can take advantage of CPU caches more effectively than equivalent Python code, leading to improved performance.
7. **Optimized Algorithms:** NumPy implements highly optimized algorithms for common operations such as sorting, searching, and linear algebra. These algorithms are often written in lower-level languages like C, ensuring fast execution.
8. **Parallelization:** Some NumPy operations, especially those involving large datasets, can take advantage of multiple processor cores, leading to parallelized computations and improved performance.
9. **Integration with Low-Level Libraries:** NumPy integrates seamlessly with low-level libraries and functions, such as BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage). These libraries are highly optimized and provide efficient implementations of linear algebra operations.

It's important to note that while NumPy provides excellent performance for many scientific and numerical computing tasks, there might be specific use cases where other libraries or specialized tools are more suitable. For example, deep learning tasks are often better handled by

libraries like TensorFlow or PyTorch, which are optimized for neural network computations.

In summary, NumPy's performance stems from its efficient array operations, vectorization, memory layout, data types, and integration with optimized algorithms and low-level libraries. This makes it a powerful tool for numerical and scientific computing in Python.

When to use Python / NumPy

NumPy arrays and Python lists are both used to store collections of elements, but they have key differences in terms of functionality, performance, and use cases.

Numpy Arrays:

1. **Homogeneous Data Type:** NumPy arrays are homogeneous, meaning all elements must have the same data type. This allows for more efficient storage and optimized operations.
2. **Multidimensional Arrays:** NumPy supports multidimensional arrays, making it well-suited for tasks involving matrices, numerical computations, and data manipulations.
3. **Vectorized Operations:** NumPy encourages vectorized operations, where operations are performed on entire arrays at once. This leads to concise and efficient code.
4. **Efficiency and Performance:** NumPy operations are implemented in compiled C and Fortran code, resulting in faster execution compared to equivalent operations in pure Python.
5. **Broadcasting:** NumPy supports broadcasting, a feature that allows for operations on arrays of different shapes and sizes without the need for explicit loops.
6. **Memory Efficiency:** NumPy arrays have a smaller memory footprint compared to Python lists, as they store elements in contiguous memory locations.

Python Lists:

1. **Heterogeneous Data Type:** Python lists can store elements of different data types, providing more flexibility but potentially leading to less efficient operations.

2. **Dynamic Resizing:** Lists in Python can dynamically resize, allowing for easy addition and removal of elements. This flexibility is beneficial for general-purpose tasks but may come at a performance cost.
3. **General-Purpose Use:** Python lists are versatile and suitable for general-purpose programming. They are not optimized for numerical computations or specialized tasks.

When to Use Each:

- **Use NumPy Arrays When:**
  - Dealing with numerical computations, linear algebra, and scientific computing.
  - Working with large datasets or multidimensional arrays.
  - Performance is crucial, and vectorized operations are desired.
  - Broadcasting is needed for efficient operations on arrays of different shapes.
- **Use Python Lists When:**
  - Dealing with general-purpose tasks and small datasets.
  - Elements of different data types need to be stored in the same container.
  - Dynamic resizing and flexibility in adding/removing elements are required.
  - Performance is not a critical factor, and simplicity is more important.

In summary, NumPy arrays are specifically designed for numerical and scientific computing tasks, providing better performance and efficiency for such operations. Python lists are more versatile and suitable for general-purpose programming tasks where numerical computations are not the primary focus. The choice between NumPy arrays and Python lists depends on the nature of the task and the specific requirements of the application.

Double-click (or enter) to edit

```
import numpy as np
from google.colab import drive
drive.mount('/content/drive')
# /content/drive/My Drive/Colab Notebooks/Numpy/planets_small.txt
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
```

```
%%time
N=100000000
list_=list(range(N))
for i in range(N):
  list_[i] = list_[i]+list_[i]
```

```
CPU times: user 26.1 s, sys: 2.19 s, total: 28.3 s
Wall time: 29.8 s
```

```
%%time
N=100000000
list_=list(range(N))
list_= [item*item for item in list_]
```

```
CPU times: user 9.98 s, sys: 5.42 s, total: 15.4 s
Wall time: 15.4 s
```

```
%%time
N=100000000
list_=list(range(N))
list_= map(lambda x: x*x, list_)
```

```
CPU times: user 1.36 s, sys: 1.43 s, total: 2.79 s
Wall time: 2.78 s
```

```
%%time
N=100000000
list_=list(range(N))
list_sum=0
for item in list_:
  list_sum += item
```

```
CPU times: user 14.9 s, sys: 4.86 s, total: 19.7 s
Wall time: 19.8 s
```

```
%%time
N=100000000
list_=list(range(N))
list_sum = sum(list_)
```

```
CPU times: user 3.27 s, sys: 1.73 s, total: 5.01 s
Wall time: 5 s
```

numpy

```
%%time
import numpy as np
N=100000000
arr=np.arange(N)
arr=arr * arr
```

```
CPU times: user 407 ms, sys: 25.3 ms, total: 432 ms
Wall time: 417 ms
```

```
%%time
import numpy as np
N=100000000
arr=np.arange(N)
arr_sum = np.sum(arr)
```

```
CPU times: user 268 ms, sys: 0 ns, total: 268 ms
Wall time: 267 ms
```

High Dimensional Array & Creating NumPy Array

```
import numpy as np
arr=np.arange(5)
print(arr, type(arr))
```

```
[0 1 2 3 4] <class 'numpy.ndarray'>
```

```
arr = np.array([0,2,4,6,8])
print(arr, "Type of :: ", type(arr), "Data Type :: ", arr.dtype)
print("Dimension of Array  : ", arr.ndim)
print("Shape of Array  : ", arr.shape)
print("Size of Array  : ", arr.size)
print("ItemSize of Array  in bytes : ", arr.itemsize)
```

```
[0 2 4 6 8] Type of ::  <class 'numpy.ndarray'> Data Type ::  int64
Dimension of Array  :  1
Shape of Array  :  (5,)
Size of Array  :  5
ItemSize of Array  :  8
```

```
import numpy as np
arr2d = np.array([
      [1,2,3],
      [4,5,6]

    ])
print(arr2d)
print("Type of :: ", type(arr2d))
print("Data Type :: ", arr2d.dtype)
print("Dimension of Array  : ", arr2d.ndim)
print("Shape of Array  : ", arr2d.shape)
print("Size of Array  : ", arr2d.size)
print("ItemSize of Array  in bytes : ", arr2d.itemsize)
```

```
[[1 2 3]
 [4 5 6]]
Type of ::  <class 'numpy.ndarray'>
Data Type ::  int64
Dimension of Array  :  2
Shape of Array  :  (2, 3)
Size of Array  :  6
ItemSize of Array  in bytes :  8
```

```
import numpy as np
arr3d = np.array([
      [
        [1,2,3],
        [4,5,6],
        [7,8,9]
      ],

      [
        [10,11,12],
        [13,14,15],
        [16,17,18]
      ]
    ])
print(arr3d)
print("Type of :: ", type(arr3d))
print("Data Type :: ", arr3d.dtype)
print("Dimension of Array  : ", arr3d.ndim)
print("Shape of Array  : ", arr3d.shape)
print("Size of Array  : ", arr3d.size)
print("ItemSize of Array  in bytes : ", arr3d.itemsize)
```

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

      [[10 11 12]
       [13 14 15]
       [16 17 18]]]
     Type of ::  <class 'numpy.ndarray'>
     Data Type ::  int64
     Dimension of Array  :  3
     Shape of Array  :  (2, 3, 3)
     Size of Array  :  18
     ItemSize of Array  in bytes :  8
```

create diffren dimeson arrays with only 1's

[ ] ↳ *4 cells hidden*

Random

```
np.random.rand(3,3)
```

```
array([[0.08425101, 0.75683133, 0.7072109 ],
       [0.68354249, 0.25276973, 0.90353977],
       [0.42741492, 0.46523142, 0.36816719]])
```

```
np.random.rand(2,3)
```

```
array([[0.35886785, 0.36904446, 0.97823166],
       [0.67234975, 0.58634752, 0.69668721]])
```

```
np.random.randint(0,100,(2,3))
```

```
array([[21, 92, 20],
       [35, 40, 48]])
```

```
np.arange(0,50, 5)
```

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

```
np.linspace(7,70,10)
```

```
array([ 7., 14., 21., 28., 35., 42., 49., 56., 63., 70.])
```

```
x=np.array([True, False, False, True])
print(x)
```

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

```
arrStr=np.array(["1.4","20.1","11.2","6.2"])
print(arrStr)
```

```
['1.4' '20.1' '11.2' '6.2']
```

```
arrStr=np.array(["1.4","20.1","11.2","6.2"], dtype="float")
print(arrStr)
```

```
[ 1.4 20.1 11.2  6.2]
```

```
arrStr=np.array([1.4,20.1,11.2,6.2], dtype="int")
print(arrStr)
```

```
[ 1 20 11  6]
```

Indexing Arrays

```
arr3d = np.array([
              [
                  [1,2,3],
                  [4,5,6]
              ],
              [
                  [7,8,9],
                  [10,11,12]
              ]

                  ])
print(arr3d[0,0,0])
```

```
1
```

print first dimension

```
arr3d.shape
```

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

```
arr3d[0,0,0] #inexing 0,0,0 of the array
```

```
1
```

```
arr3d[0,1,2]
```

```
6
```

```
arr3d[1,0,0]
```

```
7
```

```
arr3d[1,1,2]
```

```
12
```

slicing of arrays

```
arr3d[0,::]  # shows only one dimesion array
```

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

```
arr3d[1,::]  # shows only one dimesion array
```

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

```
arr3d[: ,1, :]  # shows only 2nd row of the all matrix
```

```
array([[ 4,  5,  6],
       [10, 11, 12]])
```

```
arr3d[: ,0, :]  # shows only 2nd row of the all matrix
```

```
array([[1, 2, 3],
       [7, 8, 9]])
```

```
arr3d[  :,  :,  0:1  ] #both rows but 0 a,d 1 column
```

```
array([[[ 1],
        [ 4]],

       [[ 7],
        [10]]])
```

get indices of even numbers in array

```
print(arr3d % 2 ==0 ) # eah element will be divided and boolen is assigned
```

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

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

Find all even numbers in a Array

```
arr3d[arr3d % 2 == 0]
```

```
array([ 2,  4,  6,  8, 10, 12])
```

find all odd numbers with array GT 3

```
arr3d[(arr3d % 2 == 1) & (arr3d > 3)]
```

```
array([ 5,  7,  9, 11])
```

```
arr_slice = arr3d[  :,  :,  0:2  ]
print(arr_slice)
print(type(arr_slice))
print(arr_slice.size)
print(arr_slice.shape)
print(arr_slice[1,0,0])
```

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

 [[ 7  8]
```

```
  [10 11]]]
<class 'numpy.ndarray'>
8
(2, 2, 2)
7
```

```
arr_slice[1,0,0]=999
print(arr_slice)
```

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

 [[999   8]
  [ 10  11]]]
```

it also changes the original array

```
arr3d
```

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

       [[999,   8,   9],
        [ 10,  11,  12]]])
```

if we change in the original it will reflect in the sub array

```
arr3d[1,0,0] =7
arr3d
print(arr_slice)
```

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

```
       [[ 7  8]
        [10 11]]]
```

if we do a deep copy then the original is not affected

```
arr_slice = np.copy(arr3d[  :,  :,  0:2  ])
print(arr_slice)
```

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

      [[ 7  8]
       [10 11]]]
```

you can also use indexing on arrays

```
 # This is formatted as code
```

```
larr=np.random.randint(0,10,5)
print(larr)
```

```
     [6 0 9 0 4]
```

```
my_indices = [1,3,4]
larr[my_indices]
```

```
     array([0, 0, 4])
```

Operations on numpy array

```
arr1= np.random.rand(3,4)
arr2= np.random.rand(3,4)
```

```
arr1+arr2
```

```
array([[1.11927253, 0.99634418, 0.9578525 , 1.4878038 ],
       [0.46511732, 1.5945864 , 1.08276699, 0.78156265],
       [1.51148878, 1.00903695, 1.48133051, 1.48579658]])
```

```
arr1-arr2
```

```
array([[ 0.7008203 ,  0.01471901, -0.52917736, -0.43707588],
       [-0.04888785,  0.01300825, -0.25782561,  0.01854153],
       [ 0.05116904, -0.85519308,  0.38508121,  0.41692173]])
```

```
arr1*arr2
```

```
array([[0.19040548, 0.24812127, 0.15936318, 0.5056312 ],
       [0.05348602, 0.63563414, 0.27647758, 0.1526241 ],
       [0.57049502, 0.07170009, 0.51151314, 0.50844193]])
```

```
arr1/arr2
```

```
array([[4.34958323, 1.02998907, 0.28827608, 0.54586681],
       [0.80977682, 1.01644971, 0.61535576, 1.0486003 ],
       [1.07007923, 0.08252408, 1.70254313, 1.78011329]])
```

```
np.exp(arr1)
```

```
array([[2.48443784, 1.6578666 , 1.23904085, 1.69107422],
       [1.23135444, 2.2340081 , 1.51054527, 1.4919024 ],
       [2.18437318, 1.07995776, 2.5426475 , 2.58922643]])
```

```
np.log(arr1)
```

```
array([[-0.09425968, -0.68214474, -1.54020308, -0.643664  ],
       [-1.56966574, -0.21840812, -0.88559013, -0.91616052],
       [-0.24675908, -2.56496425, -0.06912946, -0.04986363]])
```

Log and exponentation is inverse function

```
np.log(np.exp(arr1))
```

```
array([[0.91004641, 0.5055316 , 0.21433757, 0.52536396],
       [0.20811474, 0.80379733, 0.41247069, 0.40005209],
       [0.78132891, 0.07692193, 0.93320586, 0.95135916]])
```

```
np.sin(arr1)
```

```
array([[0.78953222, 0.48427261, 0.2127002 , 0.50152791],
       [0.20661568, 0.71999653, 0.40087403, 0.38946632],
       [0.70422354, 0.0768461 , 0.80353239, 0.81420535]])
```

```
np.sqrt(arr1)
```

```
array([[0.95396353, 0.71100745, 0.46296606, 0.72481995],
       [0.45619594, 0.89654745, 0.64223881, 0.63249671],
       [0.88392811, 0.27734803, 0.96602581, 0.97537642]])
```

```
np.sum(arr1)
```

```
6.7225302379181695
```

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

print(np.mean(a, axis=0))

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

```
2.5
[2. 3.]
[1.5 3.5]
```

```
arr_inverse = 1/arr1
arr_inverse
```

```
array([[ 1.09884505,  1.97811572,  4.66553765,  1.90344233],
       [ 4.80504179,  1.24409471,  2.42441468,  2.4996745 ],
       [ 1.27987072, 13.00019356,  1.07157492,  1.05112774]])
```

```
arr0=np.zeros((3,3))
arr_inverse = 1/arr0
arr_inverse
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in true_divide

array([[inf, inf, inf],
       [inf, inf, inf],
       [inf, inf, inf]])
```

```
np.isinf(arr_inverse[0,1])
```

```
True
```

```
np.isinf(arr_inverse)
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Double-click (or enter) to edit

Broadcasting

When to do broadcasting, go through your dimensions backwards and see if it matches then if missing add the dimenson to it

```
arr1 = np.arange(6)
arr1 = arr1.reshape(3,2)
arr2 = np.arange(6).reshape((3,2))
arr1+arr2 # add 2 array of same size
```

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

```
arr2[0].reshape((1,2))
```

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

```
print(arr1, " + ")
print(arr2[0].reshape((1,2)))
print(arr1+arr2[0].reshape((1,2))) # a [3,2] added with [1,2]
```

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

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

```
print(arr2)
print(arr2[: ,0].reshape((3,1)))
print(arr2 + arr2[: ,0].reshape((3,1))) #adding [3,2] and [3,1]
```

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

```
# 2 planes, 3 roows and 4 columns
arr234 = np.arange(1,25).reshape(2,3,4)
arr111 = np.ones((1,4))

print(arr111)
print(arr234)
print(arr234+arr111) #adding [2,3,4] to [1,4]
# missing plane is added and 1 row is replicated to 3,
#so the second array will be [2,3,4]
```

```
     [[1. 1. 1. 1.]]
     [[[ 1  2  3  4]
       [ 5  6  7  8]
       [ 9 10 11 12]]

      [[13 14 15 16]
       [17 18 19 20]
       [21 22 23 24]]]
     [[[ 2.  3.  4.  5.]
```

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

       [[14. 15. 16. 17.]
        [18. 19. 20. 21.]
        [22. 23. 24. 25.]]]
```

```
_4ele = np.arange(4)
_5ele = np.arange(5)
print(_4ele.shape , _5ele.shape)
print(_4ele+_5ele) # operands could not be broadcast together with shapes (4,) (5,)
# error is due to incompatability of arrays
```

```
(4,) (5,)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-567f5d3f9807> in <module>
      2 _5ele = np.arange(5)
      3 print(_4ele.shape , _5ele.shape)
----> 4 print(_4ele+_5ele)

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

SEARCH STACK OVERFLOW

```
_4ele = np.arange(4)
_5ele = np.arange(5)
print(_4ele.reshape(4,1), _5ele)
print(_4ele.reshape(4,1) +  _5ele) # this works because [4,1] and [1,5]
```

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

Transpose array i.e Row to Column

```
arr43 = np.random.randint(5, size=(4,3))
x8= arr43.T
print(x8)
```

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

```
arr43
```

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

File Handling

```
planets_small = np.loadtxt("/content/drive/My Drive/Colab Notebooks/Numpy/planets_small.txt")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
planets_small = np.loadtxt("/content/drive/My Drive/Colab Notebooks/Numpy/planets_small.txt",
 skiprows =1,
 usecols = (1,2,3,4,5,6,7,8,9) )
print(planets_small)
print(planets_small.shape)
```

```
[[3.3000e-01 4.8700e+00 5.9700e+00 6.4200e-01 1.8980e+03 5.6800e+02
  8.6800e+01 1.0200e+02 1.4600e-02]
 [5.7900e+01 1.0820e+02 1.4960e+02 2.2790e+02 7.7860e+02 1.4335e+03
  2.8725e+03 4.4951e+03 5.9064e+03]
 [4.2226e+03 2.8020e+03 2.4000e+01 2.4700e+01 9.9000e+00 1.0700e+01
  1.7200e+01 1.6100e+01 1.5330e+02]]
(3, 9)
```

```
planets = np.genfromtxt(
    "/content/drive/My Drive/Colab Notebooks/Numpy/planets.txt",
    skip_header =1,
    usecols = (1,2,3,4,5,6,7,8,9)
                            )
print(planets)
print(planets.shape)
```

```
[[ 3.30000e-01  4.87000e+00  5.97000e+00  7.30000e-02  6.42000e-01
   1.89800e+03  5.68000e+02  8.68000e+01  1.02000e+02]
 [ 4.87900e+03  1.21040e+04  1.27560e+04  3.47500e+03  6.79200e+03
   1.42984e+05  1.20536e+05  5.11180e+04  4.95280e+04]
 [ 5.42700e+03  5.24300e+03  5.51400e+03  3.34000e+03  3.93300e+03
   1.32600e+03  6.87000e+02  1.27100e+03  1.63800e+03]
 [ 3.70000e+00  8.90000e+00  9.80000e+00  1.60000e+00  3.70000e+00
   2.31000e+01  9.00000e+00  8.70000e+00  1.10000e+01]
 [ 4.30000e+00  1.04000e+01  1.12000e+01  2.40000e+00  5.00000e+00
   5.95000e+01  3.55000e+01  2.13000e+01  2.35000e+01]
 [ 1.40760e+03 -5.83250e+03  2.39000e+01  6.55700e+02  2.46000e+01
   9.90000e+00  1.07000e+01 -1.72000e+01  1.61000e+01]
 [ 4.22260e+03  2.80200e+03  2.40000e+01  7.08700e+02  2.47000e+01
   9.90000e+00  1.07000e+01  1.72000e+01  1.61000e+01]
```

```
       [ 5.79000e+01  1.08200e+02  1.49600e+02  3.84000e-01  2.27900e+02
         7.78600e+02  1.43350e+03  2.87250e+03  4.49510e+03]
       [ 4.60000e+01  1.07500e+02  1.47100e+02  3.63000e-01  2.06600e+02
         7.40500e+02  1.35260e+03  2.74130e+03  4.44450e+03]
       [ 6.98000e+01  1.08900e+02  1.52100e+02  4.06000e-01  2.49200e+02
         8.16600e+02  1.51450e+03  3.00360e+03  4.54570e+03]
       [ 8.80000e+01  2.24700e+02  3.65200e+02  2.73000e+01  6.87000e+02
         4.33100e+03  1.07470e+04  3.05890e+04  5.98000e+04]
       [ 4.74000e+01  3.50000e+01  2.98000e+01  1.00000e+00  2.41000e+01
         1.31000e+01  9.70000e+00  6.80000e+00  5.40000e+00]
       [ 7.00000e+00  3.40000e+00  0.00000e+00  5.10000e+00  1.90000e+00
         1.30000e+00  2.50000e+00  8.00000e-01  1.80000e+00]
       [ 2.05000e-01  7.00000e-03  1.70000e-02  5.50000e-02  9.40000e-02
         4.90000e-02  5.70000e-02  4.60000e-02  1.10000e-02]
       [ 3.40000e-02  1.77400e+02  2.34000e+01  6.70000e+00  2.52000e+01
         3.10000e+00  2.67000e+01  9.78000e+01  2.83000e+01]
       [ 1.67000e+02  4.64000e+02  1.50000e+01 -2.00000e+01 -6.50000e+01
        -1.10000e+02 -1.40000e+02 -1.95000e+02 -2.00000e+02]
       [ 0.00000e+00  9.20000e+01  1.00000e+00  0.00000e+00  1.00000e-02
                 nan          nan          nan          nan]
       [ 0.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  2.00000e+00
         7.90000e+01  8.20000e+01  2.70000e+01  1.40000e+01]
       [ 0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
         1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]
       [ 1.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00
         1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]]
      (20, 9)
```

```
np.isnan(planets)
```

```
     array([[False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
            [False, False, False, False, False, False, False, False, False],
```

```
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False,  True,  True,  True,  True],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False]])
```

```
planets_new  = np.nan_to_num(planets, nan=-1 )
print(planets_new)
```

```
[[ 3.30000e-01  4.87000e+00  5.97000e+00  7.30000e-02  6.42000e-01
   1.89800e+03  5.68000e+02  8.68000e+01  1.02000e+02]
 [ 4.87900e+03  1.21040e+04  1.27560e+04  3.47500e+03  6.79200e+03
   1.42984e+05  1.20536e+05  5.11180e+04  4.95280e+04]
 [ 5.42700e+03  5.24300e+03  5.51400e+03  3.34000e+03  3.93300e+03
   1.32600e+03  6.87000e+02  1.27100e+03  1.63800e+03]
 [ 3.70000e+00  8.90000e+00  9.80000e+00  1.60000e+00  3.70000e+00
   2.31000e+01  9.00000e+00  8.70000e+00  1.10000e+01]
 [ 4.30000e+00  1.04000e+01  1.12000e+01  2.40000e+00  5.00000e+00
   5.95000e+01  3.55000e+01  2.13000e+01  2.35000e+01]
 [ 1.40760e+03 -5.83250e+03  2.39000e+01  6.55700e+02  2.46000e+01
   9.90000e+00  1.07000e+01 -1.72000e+01  1.61000e+01]
 [ 4.22260e+03  2.80200e+03  2.40000e+01  7.08700e+02  2.47000e+01
   9.90000e+00  1.07000e+01  1.72000e+01  1.61000e+01]
 [ 5.79000e+01  1.08200e+02  1.49600e+02  3.84000e-01  2.27900e+02
   7.78600e+02  1.43350e+03  2.87250e+03  4.49510e+03]
 [ 4.60000e+01  1.07500e+02  1.47100e+02  3.63000e-01  2.06600e+02
   7.40500e+02  1.35260e+03  2.74130e+03  4.44450e+03]
 [ 6.98000e+01  1.08900e+02  1.52100e+02  4.06000e-01  2.49200e+02
   8.16600e+02  1.51450e+03  3.00360e+03  4.54570e+03]
 [ 8.80000e+01  2.24700e+02  3.65200e+02  2.73000e+01  6.87000e+02
   4.33100e+03  1.07470e+04  3.05890e+04  5.98000e+04]
 [ 4.74000e+01  3.50000e+01  2.98000e+01  1.00000e+00  2.41000e+01
   1.31000e+01  9.70000e+00  6.80000e+00  5.40000e+00]
 [ 7.00000e+00  3.40000e+00  0.00000e+00  5.10000e+00  1.90000e+00
   1.30000e+00  2.50000e+00  8.00000e-01  1.80000e+00]
```

```
       [ 2.05000e-01  7.00000e-03  1.70000e-02  5.50000e-02  9.40000e-02
         4.90000e-02  5.70000e-02  4.60000e-02  1.10000e-02]
       [ 3.40000e-02  1.77400e+02  2.34000e+01  6.70000e+00  2.52000e+01
         3.10000e+00  2.67000e+01  9.78000e+01  2.83000e+01]
       [ 1.67000e+02  4.64000e+02  1.50000e+01 -2.00000e+01 -6.50000e+01
        -1.10000e+02 -1.40000e+02 -1.95000e+02 -2.00000e+02]
       [ 0.00000e+00  9.20000e+01  1.00000e+00  0.00000e+00  1.00000e-02
        -1.00000e+00 -1.00000e+00 -1.00000e+00 -1.00000e+00]
       [ 0.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  2.00000e+00
         7.90000e+01  8.20000e+01  2.70000e+01  1.40000e+01]
       [ 0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
         1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]
       [ 1.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00
         1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]]
```

saving the data back

```
np.savetxt(
    "/content/drive/My Drive/Colab Notebooks/Numpy/planets_new.txt",
    planets_new,
    delimiter = ","
                           )
```

To save as binary file so that the read and write are fast

No extension and also no delimiter

```
np.save(
    "/content/drive/My Drive/Colab Notebooks/Numpy/planets_numpy_File",
    planets_new)
```

```
!ls -lh
```

```
total 8.0K
drwx------ 5 root root 4.0K Oct  2 12:17 drive
drwxr-xr-x 1 root root 4.0K Sep 26 13:45 sample_data
```

```
arrf1 = np.random.rand(1000, 10)
arrf2 = np.random.rand(2000, 5)
arrf3 = np.random.rand(10, 10000)
np.savez("/content/drive/My Drive/Colab Notebooks/Numpy/many_files",
         arrf1,arrf2,arrf3)
```

```
arrs=np.load(
  "/content/drive/My Drive/Colab Notebooks/Numpy/many_files.npz"
)
```

```
arrs.files  # get the files
```

```
arrs["arr_0"]  #print just one file
```

```
array([[0.30995393, 0.95100393, 0.39448441, ..., 0.41796277, 0.35132871,
        0.17324549],
       [0.72898005, 0.91882492, 0.75787393, ..., 0.20481267, 0.48282973,
        0.96301701],
       [0.63662371, 0.04193139, 0.63077853, ..., 0.90470819, 0.0797945 ,
        0.39671762],
       ...,
       [0.9277067 , 0.99261307, 0.57069027, ..., 0.51386695, 0.93152637,
        0.29293758],
       [0.48299345, 0.88399119, 0.21092462, ..., 0.23232852, 0.11704555,
        0.8680223 ],
       [0.81068364, 0.77605621, 0.96151385, ..., 0.85581209, 0.99281605,
        0.13030047]])
```

```
np.savez_compressed("/content/drive/My Drive/Colab Notebooks/Numpy/many_files_compressed",
         arrf1,arrf2,arrf3)
```

Load arrays from native numpy format file

```
from_numpy_fmt = np.load(
"/content/drive/My Drive/Colab Notebooks/Numpy/planets_numpy_File.npy"
                         )
print(from_numpy_fmt)
```

```
[[ 3.30000e-01  4.87000e+00  5.97000e+00  7.30000e-02  6.42000e-01
   1.89800e+03  5.68000e+02  8.68000e+01  1.02000e+02]
 [ 4.87900e+03  1.21040e+04  1.27560e+04  3.47500e+03  6.79200e+03
   1.42984e+05  1.20536e+05  5.11180e+04  4.95280e+04]
 [ 5.42700e+03  5.24300e+03  5.51400e+03  3.34000e+03  3.93300e+03
   1.32600e+03  6.87000e+02  1.27100e+03  1.63800e+03]
 [ 3.70000e+00  8.90000e+00  9.80000e+00  1.60000e+00  3.70000e+00
   2.31000e+01  9.00000e+00  8.70000e+00  1.10000e+01]
 [ 4.30000e+00  1.04000e+01  1.12000e+01  2.40000e+00  5.00000e+00
   5.95000e+01  3.55000e+01  2.13000e+01  2.35000e+01]
 [ 1.40760e+03 -5.83250e+03  2.39000e+01  6.55700e+02  2.46000e+01
   9.90000e+00  1.07000e+01 -1.72000e+01  1.61000e+01]
 [ 4.22260e+03  2.80200e+03  2.40000e+01  7.08700e+02  2.47000e+01
   9.90000e+00  1.07000e+01  1.72000e+01  1.61000e+01]
 [ 5.79000e+01  1.08200e+02  1.49600e+02  3.84000e-01  2.27900e+02
   7.78600e+02  1.43350e+03  2.87250e+03  4.49510e+03]
 [ 4.60000e+01  1.07500e+02  1.47100e+02  3.63000e-01  2.06600e+02
   7.40500e+02  1.35260e+03  2.74130e+03  4.44450e+03]
 [ 6.98000e+01  1.08900e+02  1.52100e+02  4.06000e-01  2.49200e+02
   8.16600e+02  1.51450e+03  3.00360e+03  4.54570e+03]
 [ 8.80000e+01  2.24700e+02  3.65200e+02  2.73000e+01  6.87000e+02
   4.33100e+03  1.07470e+04  3.05890e+04  5.98000e+04]
 [ 4.74000e+01  3.50000e+01  2.98000e+01  1.00000e+00  2.41000e+01
   1.31000e+01  9.70000e+00  6.80000e+00  5.40000e+00]
 [ 7.00000e+00  3.40000e+00  0.00000e+00  5.10000e+00  1.90000e+00
   1.30000e+00  2.50000e+00  8.00000e-01  1.80000e+00]
 [ 2.05000e-01  7.00000e-03  1.70000e-02  5.50000e-02  9.40000e-02
   4.90000e-02  5.70000e-02  4.60000e-02  1.10000e-02]
 [ 3.40000e-02  1.77400e+02  2.34000e+01  6.70000e+00  2.52000e+01
   3.10000e+00  2.67000e+01  9.78000e+01  2.83000e+01]
```

```
[ 1.67000e+02  4.64000e+02  1.50000e+01 -2.00000e+01 -6.50000e+01
 -1.10000e+02 -1.40000e+02 -1.95000e+02 -2.00000e+02]
[ 0.00000e+00  9.20000e+01  1.00000e+00  0.00000e+00  1.00000e-02
 -1.00000e+00 -1.00000e+00 -1.00000e+00 -1.00000e+00]
[ 0.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  2.00000e+00
  7.90000e+01  8.20000e+01  2.70000e+01  1.40000e+01]
[ 0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]
[ 1.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00
  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00]]
```

Stats with numpy

```
arrst= np.random.rand(100000,)
print(np.amin(arrst))
print(np.amax(arrst))
print("Mean = ", np.mean(arrst))
print("Median = ", np.median(arrst))
print("Varience = ", np.var(arrst))
print("Standard Deveation = ", np.std(arrst))
print("25 Percentile = ", np.percentile(arrst, 25))
print("25 Percentile = ", np.percentile(arrst, 75))
print("Inter Quartile Percentile = ", np.percentile(arrst, 75) - np.percentile(arrst, 25))

quartiles=np.percentile(arrst, [25,75])
print("Inter Quartile Percentile = ", quartiles[1]-quartiles[0])
#instead of calling np. many times -- above two lines

print("Z Score -How far particular point is from Mean = \n",
      arrst - np.mean(arrst)/np.std(arrst)
      )
```

```
1.778806587160453e-05
0.9999919070499059
```

```
Mean =  0.49897219592599334
Median =  0.4984176199154281
Varience =  0.083446431733198
Standard Deveation =  0.28887096034942317
25 Percentile =  0.24830921183985133
25 Percentile =  0.7486664442963318
Inter Quartile Percentile =  0.5003572324564804
Inter Quartile Percentile =  0.5003572324564804
Z Score -How far particular point is from Mean =
 [-1.20396951 -0.97358098 -1.28210918 ... -0.87802972 -1.0486598
 -1.6271281 ]
```

Histogram

```
np.histogram(arrst)
#returns 2 array: 2nd array gives the BINS for histogram
# array 1 gives the points in the range
```

```
(array([10076, 10066, 10082,  9875, 10055, 10039,  9937,  9956,  9926,
         9988]),
 array([1.77880659e-05, 1.00015200e-01, 2.00012612e-01, 3.00010024e-01,
        4.00007436e-01, 5.00004848e-01, 6.00002259e-01, 6.99999671e-01,
        7.99997083e-01, 8.99994495e-01, 9.99991907e-01]))
```

```
np.histogram(arrst, bins=5)
```

```
(array([20142, 19957, 20094, 19893, 19914]),
 array([1.77880659e-05, 2.00012612e-01, 4.00007436e-01, 6.00002259e-01,
        7.99997083e-01, 9.99991907e-01]))
```

```
np.histogram(arrst, bins=[0, .25, .5, .75, 1])
```

```
(array([25181, 24971, 24983, 24865]), array([0.  , 0.25, 0.5 , 0.75, 1.  ]))
```

points to bin number

```
np.histogram(arrst, bins=[0, .25, .5, .75, 1])
np.digitize(arrst, bins=[0, .25, .5, .75, 1])
```

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

will tell in which bins the data point is

```
arrint = np.random.randint(0,10, (10))
print(arrint)
bins=[0,4,8,10]
np.digitize(arrint, bins)  # right=True will add the right number
```

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

MultiDimension Array

Also work with randn - which will give you nomalized array

```
weight = np.random.randint(50,80,100)
height = np.random.randint(150,185,100)
age    = np.random.randint(17,22,100) # bring them togeater
#np.concatenate((weight, height, age)).shape # We need 3 columns with 100 rows
#the above fn prints 300 data, .e one after the other
combArr = np.vstack((weight, height, age))
print("Shape  = ", np.vstack((weight, height, age)).shape)
print("Minimum in each Cols", np.amin(combArr, axis =1))
print("Max in each Cols", np.amax(combArr, axis =1))

print("Mean = ", np.mean(combArr, axis=1))
print("Median = ", np.median(combArr, axis=1))
print("Varience = ", np.var(combArr, axis=1))
print("Standard Deveation = ", np.std(combArr, axis=1))
print("25 Percentile = ", np.percentile(combArr,  25, axis=1))
print("25 Percentile = ", np.percentile(combArr,  75, axis=1))
print("Inter Quartile Percentile = ",
      np.percentile(combArr,  75, axis=1) -
      np.percentile(combArr,  25, axis=1))

quartiles=np.percentile(combArr, [25,75], axis=1)
print("Inter Quartile Percentile = ", quartiles[1]-quartiles[0])
```

```
Shape  =  (3, 100)
Minimum in each Cols [ 50 150  17]
Max in each Cols [ 79 184  21]
Mean =  [ 64.47 167.83  18.9 ]
Median =  [ 64. 169.  19.]
Varience =  [ 75.0091 113.9611   1.73  ]
Standard Deveation =  [ 8.66077941 10.67525644  1.31529464]
25 Percentile =  [ 57.75 158.    18.  ]
25 Percentile =  [ 72.   177.25  20.  ]
Inter Quartile Percentile =  [14.25 19.25  2.  ]
Inter Quartile Percentile =  [14.25 19.25  2.  ]
```

```
arrint = np.random.randint(0,10, (10))

print(np.cumsum(arrint))
print("Means of No.s =", np.cumsum(arrint)/np.arange(1,11))
```

```
[ 6 12 18 27 29 36 40 47 48 48]
Means of No.s = [6.         6.         6.         6.75       5.8        6.
 5.71428571 5.875      5.33333333 4.8       ]
```

Double-click (or enter) to edit

Checking Stats rules with numpy

Mean subracted array has zero mean

```
arr = np.random.rand(1000)
mean = np.mean(arr)
print(mean)
arr1= arr-mean
np.mean(arr1) # Should be Zero
```

```
0.4987605077007912
1.909583602355269e-17
```

computing mean with small values

```
arrm= np.random.rand(1000)
for k in range(1,100):
  arrm1=arr[0:k]
  print(k ,"mean = ", np.mean(arrm1))
```

```
np.where?
```

```
import numpy as np
a = np.arange(10)
b = np.where(a % 2 == 1, 1, a)
print(b)
```

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

```
arrh= np.random.rand(300)
np.cumsum(arrh) / np.arange(1,301)
```

Effect of outliers on mean and median

```
arro = np.random.randint(1,100,100)
print("Mean  =" , np.mean(arro))
print("Median " ,np.median(arro))
# add outliers
arro=np.append(arro, [2,3, 2000, 1000])
print("Mean  =" , np.mean(arro))
print("Median " ,np.median(arro))
```

```
Mean  = 50.2
Median  51.0
Mean  = 77.16346153846153
Median  51.0
```

Effect of scaling arrays on mean and median

```
arrs = np.random.randint(1,100,100)
print("Mean  =" , np.mean(arrs))
print("Median " ,np.median(arrs))

print("Mean  =" , np.mean(2.5 * arrs + 0.50))
print("Mean  =" , 2.5*np.mean(arrs)+ 0.50)
print("Median " , np.median(2.5 * arrs + 0.50))
print("Median " , 2.5*np.median(arrs)+ 0.50)

# Varience -- shift will not affect varience
print(np.var(2.5 * arrs +0.5), 2.5*2.5*np.var(arrs))
# Standard Deviation - shift Will not affect std
print(np.std(2.5 * arrs +0.5), 2.5*np.std(arrs))
```

```
Mean  = 52.34
Median  52.0
Mean  = 131.35
Mean  = 131.35000000000002
Median  130.5
Median  130.5
4335.027499999999 4335.027499999999
65.84092572253218 65.84092572253218
```

```
import numpy as np
a = np.array([1,2,3,4])
a[0] =0.9
print(a[0], a.dtype)
```

```
0 int64
```

Find Mean, Median, Mode, IQR for Sachin, Rahul and India

```
#!head  #unix command to view text file
valArr = np.loadtxt(
    "/content/drive/My Drive/Colab Notebooks/Numpy/cric_data.tsv",
    skiprows =1 )
#print(valArr)

valArr = valArr[   : ,[1,2,3]]
sachin = valArr[   :  ,0]
rahul = valArr[   :  ,1]
India = valArr[   :  ,2]
#print(sachin)

def stats(name,cols):
  print(name, "Statistics")
  print("Mean", np.mean(cols))
  print("median", np.median(cols))
  print("Iqr", np.percentile(cols, 75)-
        np.percentile(cols, 25))

stats("Sachin", sachin)
stats("Rahul", rahul)
stats("India", India)
```

```
Sachin Statistics
Mean 39.87555555555556
median 27.0
Iqr 57.0
Rahul Statistics
Mean 32.062222222222225
median 22.0
Iqr 46.0
India Statistics
Mean 220.79555555555555
median 216.0
Iqr 98.0
```

Other way of doing this

```
means= np.mean(valArr, axis = 0)
means[0]
percent = np.percentile(valArr, 75, axis =0)
percent
```

```
array([ 63.,  53., 273.])
```

Histogram of Sachin with 10 bins

```
np.histogram(sachin ,bins=10)
```

```
(array([99, 36, 28, 16, 11, 17,  8,  8,  1,  1]),
 array([  0. ,  18.6,  37.2,  55.8,  74.4,  93. , 111.6, 130.2, 148.8,
        167.4, 186. ]))
```

Sachin mean grouped by

```
#sachin.reshape(9,25)
np.mean(sachin.reshape(9,25), axis = 1)
```

```
array([33.96, 49.4 , 38.48, 40.16, 39.36, 38.2 , 44.6 , 39.52, 35.2 ])
```

Find mean of sachin's scores where he has scored a century

```
newsac = sachin[sachin >= 100]
np.mean(newsac)
```

```
125.0
```

Find mean of sachin's scores where rhul score is less than 10

```
np.mean(sachin[rahul <= 10])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-160-e1fad5aa11aa> in <module>
----> 1 np.mean(sachin[rahul <= 10])

IndexError: too many indices for array: array is 1-dimensional, but 2 were indexed
```

SEARCH STACK OVERFLOW

sachins average when IQR =25,50,75,100

```
iqrIndia = np.percentile(India, [25,50,75,100])
print(iqrIndia)
iqrIndia= iqrIndia.reshape(4,1)
India.shape
indices = India <= iqrIndia
indices.shape
```

```
[175. 216. 273. 499.]
(4, 225)
```

```
sachin[indices[0,:]]
for i in range(4):
  print(i, np.mean(sachin[indices[i]]))
```

```
0 20.912280701754387
1 27.92920353982301
2 32.0887573964497
```