Numpy Documentation

Installation

```
pip install numpy
conda install numpy
```

Import

Numpy Python library can be imported using the following command

```
In [1]: import numpy as np
```

Converting Lists into Numpy Arrays

Use the np.array cmd which can take in a parameter of a list, to convert a regular python list into a Numpy Array

```
In [2]: list1 = [0,1,2,3,4]
        list1
Out[2]: [0, 1, 2, 3, 4]

In [3]: arr1d = np.array(list1)
        arr1d
Out[3]: array([0, 1, 2, 3, 4])
```

NOTE: Elements cannot be appended into Numpy Arrays since memory has already been allocated

The append cmd can be used to append elements into lists. However, this command doesn't work with numpy arrays

```
In [4]: list1.append(5)

In [5]: # arr1d.append(5) # Memory already allocated can't reallocate
```

Creating Multi-dimensional Numpy Arrays

Similarly, when nested lists are passed into the np.array() cmd, the list is converted into a 2d array

```
In [7]: list2 = [[1,1,1],[2,2,2],[3,3,3]]
        arr2d = np.array(list2)

In [8]: type(arr2d)
Out[8]: numpy.ndarray
```

The Data-type can also be checked using the dtype cmd

```
In [9]: arr2d.dtype
Out[9]: dtype('int32')
```

When converting the list into a numpy array, the data-type of the elements can be modified using the dtype parameter

```
In [10]: arr2d = np.array(list2,dtype='float')

In [11]: arr2d
Out[11]: array([[1., 1., 1.],
                [2., 2., 2.],
                [3., 3., 3.]])
```

The astype () cmd can be used to cast a particular object into the desired dtype

Using the inplace=False parameter doesn't modify the original data

```
In [12]: arr2d.astype('int')
Out[12]: array([[1, 1, 1],
                [2, 2, 2],
                [3, 3, 3]])

In [13]: arr2d.astype('str')
Out[13]: array([['1.0', '1.0', '1.0'],
                ['2.0', '2.0', '2.0'],
                ['3.0', '3.0', '3.0']], dtype='<U32')

In [14]: arr2d  # because inplace was False
Out[14]: array([[1., 1., 1.],
                [2., 2., 2.],
                [3., 3., 3.]])
```

Converting Numpy array back to list

The original list can retrieved by using the tolist() cmd

```
In [16]: np2list = arr2d.tolist()
         np2list
Out[16]: [[1.0, 1.0, 1.0], [2.0, 2.0, 2.0], [3.0, 3.0, 3.0]]
```

Gathering Useful information regarding the Numpy array

The statistics regarding unidimensional or multidimensional data can be found using the shape, dtype, size and dimension keywords

```
In [22]: print('shape',arr1d.shape)
         print('dtype',arr1d.dtype)
         print('size',arr1d.size)
         print('dimension',arr1d.ndim)

         shape (5,)
         dtype int32
         size 5
         dimension 1
```

```
In [23]: print('shape',arr2d.shape)
         print('dtype',arr2d.dtype)
         print('size',arr2d.size)
         print('dimension',arr2d.ndim)

         shape (3, 3)
         dtype float64
         size 9
         dimension 2
```

Printing the contents of the Numpy Array

The contents of the numpy array can be revealed simply using the print () cmd.

```
In [24]: print(arr1d)
         print()
         print(arr2d)

         [0 1 2 3 4]

         [[1. 1. 1.]
          [2. 2. 2.]
          [3. 3. 3.]]
```

Accessing elements of the Array

The elements in the array can be accessed simply using the indices in square brackets succeeding the numpy array name as illustrated below

```
In [26]: arr1d
Out[26]: array([ 0,  1,  4,  9, 16])

In [27]: arr1d[1]
Out[27]: 1

In [28]: arr2d[1]
Out[28]: array([2., 2., 2.])

In [29]: arr2d[0][0]
Out[29]: 1.0
```

Further desired conditions can be applied according to the user and the same can be passed into the indices of the numpy array to retrieve the elements matching the particular condition.

```
In [30]: boolarr = arr2d < 3

In [31]: boolarr

Out[31]: array([[ True,  True,  True],
                [ True,  True,  True],
                [False, False, False]])

In [32]: arr2d[boolarr]  # makes an array using the true-condition items

Out[32]: array([1., 1., 1., 2., 2., 2.])
```

Reversing rows, columns or both

The Numpy rows, columns or both can reversed by passing in -1 while slicing the numpy array as illustrated below.

```
In [33]: # Reversing Rows
         arr2d[::-1,]

Out[33]: array([[3., 3., 3.],
                [2., 2., 2.],
                [1., 1., 1.]])

In [34]: # Reversing Columns
         arr2d[:,::-1]

Out[34]: array([[1., 1., 1.],
                [2., 2., 2.],
                [3., 3., 3.]])

In [35]: # Reversing both
         arr2d[::-1,::-1]

Out[35]: array([[3., 3., 3.],
                [2., 2., 2.],
                [1., 1., 1.]])
```

Numpy also supports NULL and INFINITE values

This can be implemented using np.nan and np.inf respectively

```
In [36]: np.nan

Out[36]: nan

In [37]: np.inf

Out[37]: inf
```

Checking for NULL and INFINITE values in the Numpy Arrays

The null and infinite values in the numpy arrays can be checked using the isnan() and isinf () cmd.

```
In [39]: np.isnan(arr2d)
Out[39]: array([[ True, False, False],
                [False, False, False],
                [False, False, False]])

In [40]: np.isinf(arr2d)
Out[40]: array([[False,  True, False],
                [False, False, False],
                [False, False, False]])

In [41]: missing_flag = np.isnan(arr2d) | np.isinf(arr2d)
         missing_flag
Out[41]: array([[ True,  True, False],
                [False, False, False],
                [False, False, False]])
```

The missing nan and inf values can replaced by accessing the numpy as illustrated before

```
In [41]: missing_flag = np.isnan(arr2d) | np.isinf(arr2d)
         missing_flag
Out[41]: array([[ True,  True, False],
                [False, False, False],
                [False, False, False]])

In [42]: # Replace inf and nan with 0

In [43]: arr2d[missing_flag]
Out[43]: array([nan, inf])

In [44]: arr2d[missing_flag] = 0

In [45]: arr2d
Out[45]: array([[0., 0., 1.],
                [2., 2., 2.],
                [3., 3., 3.]])
```

Statistical Operations

Statistical Operations can be performed on the numpy array using the mean (), std (), max (), min (), etc. cmds.

```
In [46]: # Mean , std, var

In [47]: print(arr2d.mean())
         # print(arr2d.median())
         # print(arr2d.mode())
         print(arr2d.std())
         print(arr2d.max())
         print(arr2d.min())

         1.7777777777777777
         1.1331154474650633
         3.0
         0.0
```

The array can be squeezed using the squeeze command

```
In [48]: arr2d.squeeze()
Out[48]: array([[0., 0., 1.],
                [2., 2., 2.],
                [3., 3., 3.]])
```

The cumulative sum of the array can be evaluated using the cumsum () command

```
In [49]: arr2d.cumsum()

Out[49]: array([ 0.,  0.,  1.,  3.,  5.,  7., 10., 13., 16.])
```

Reshape, Flatten and Ravel

The array can be reshaped, flattened (converted into a 1d array) and raveled (also converted into a 1d array and in-place)

```
In [54]: arr2d.reshape(9,1)

Out[54]: array([[0.],
               [0.],
               [1.],
               [2.],
               [2.],
               [2.],
               [3.],
               [3.],
               [3.]])

In [55]: a = arr2d.flatten()  # to convert into single dimensional array
         a  # copy

Out[55]: array([0., 0., 1., 2., 2., 2., 3., 3., 3.])

In [56]: b = arr2d.ravel()  # to convert into single dimensional array
         b # reference, thus memory-efficient

Out[56]: array([0., 0., 1., 2., 2., 2., 3., 3., 3.])
```

Sequences, Repetitions and Random Numbers

A sequence of real numbers can be generated within a range using the arrange command and also passing the desired dtype parameter.

```
In [59]: np.arange(1,5,dtype='int')
         # np.arange(1,5,dtype='float')
         # np.arange(1,5,dtype='str')
         # np.arange(1,5,dtype='object')

Out[59]: array([1, 2, 3, 4])

In [60]: np.arange(1,50,2)  # Odd nos
         # np.arange(2,50,2)  # Even nos

Out[60]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,
               35, 37, 39, 41, 43, 45, 47, 49])
```

A sequence of linearly or logarithmically spaced real numbers can be inserted between 2 limits using the linspace cmd () or logspace() cmd

```
In [61]: # Inserting linearly spaecd numbers between numbers
         np.linspace(1,50,10)

Out[61]: array([ 1.        ,  6.44444444, 11.88888889, 17.33333333, 22.77777778,
                28.22222222, 33.66666667, 39.11111111, 44.55555556, 50.        ])

In [62]: # Inserting logarithmically spaecd numbers between numbers
         np.logspace(1,50,10)

Out[62]: array([1.00000000e+01, 2.78255940e+06, 7.74263683e+11, 2.15443469e+17,
                5.99484250e+22, 1.66810054e+28, 4.64158883e+33, 1.29154967e+39,
                3.59381366e+44, 1.00000000e+50])
```

Constructing Matrices with all zeroes and ones as elements

These types of matrices can be constructed using the zeroes () and ones () cmd

```
In [63]: np.zeros([2,2])

Out[63]: array([[0., 0.],
                [0., 0.]])

In [64]: np.ones([2,2])

Out[64]: array([[1., 1.],
                [1., 1.]])
```

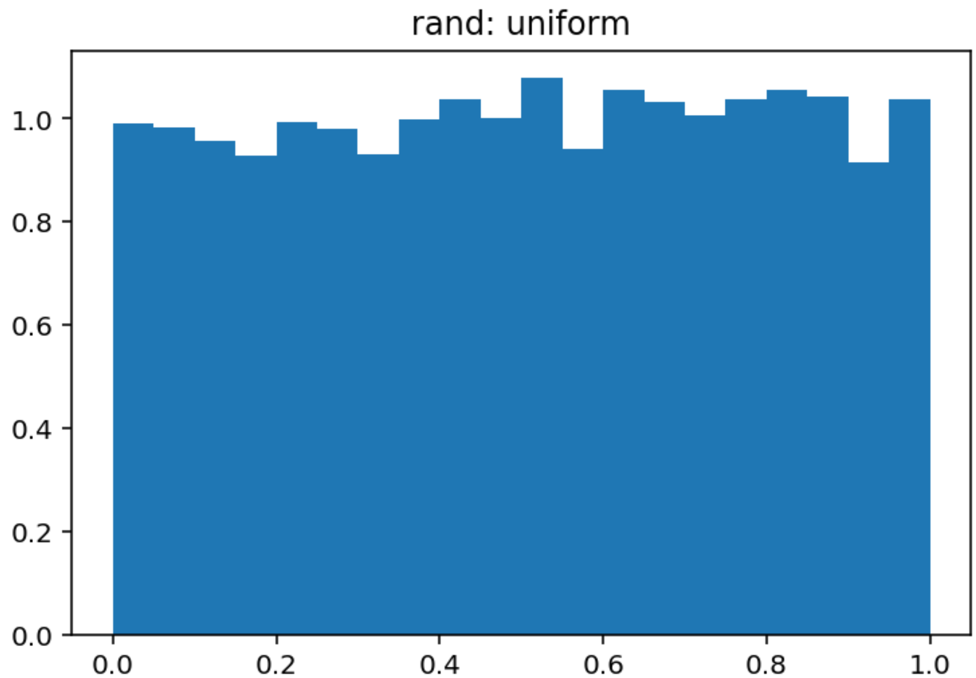
Random Numbers

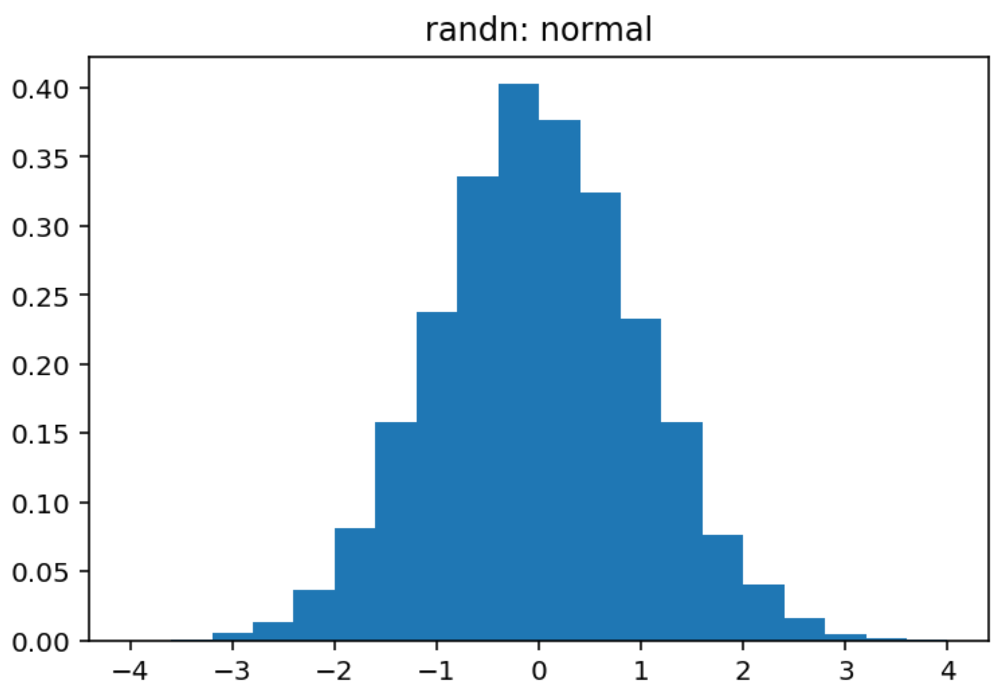
Numpy has a random module which can be used to generate random required number of floating numbers or integers within a given range.

Random.seed() is a starting point in generating random numbers

Random.randint() is used for generating random integers

Random.rand() and Random.randn() are used for generating random real numbers between 0 and 1. Additionally while rand generates numbers randomly where as randn makes sure that the random makes sure that the numbers generated follow a Bell-shaped Normal Gaussian Distribution curve.





The following codes illustrate randomly generated real numbers using the random package module of Python:

```
In [70]: np.random.rand(3,3)  # uniformly distributed (0,1)

Out[70]: array([[0.22008009, 0.64153313, 0.03585856],
                [0.34092363, 0.23560165, 0.85592572],
                [0.84215434, 0.73063413, 0.77861593]])

In [71]: np.random.randn(3,3)  # normally distributed (-1,1)

Out[71]: array([[ 0.20085905,  1.04299235,  0.8197348 ],
                [ 1.45136194, -0.00313677, -0.46253146],
                [-0.07440822, -0.88127775,  0.72837506]])

In [72]: np.random.randint(0,10,[3,3])

Out[72]: array([[5, 5, 3],
                [0, 8, 8],
                [2, 8, 7]])

In [73]: np.random.seed(0)
         np.random.randint(0,10,[3,3])

Out[73]: array([[5, 0, 3],
                [3, 7, 9],
                [3, 5, 2]])

In [74]: np.random.seed(0)
         np.random.randint(0,10,[3,3])

Out[74]: array([[5, 0, 3],
                [3, 7, 9],
                [3, 5, 2]])

In [75]: np.random.seed(1)
         np.random.randint(0,10,[3,3])

Out[75]: array([[5, 8, 9],
                [5, 0, 0],
                [1, 7, 6]])
```

Sorting a Numpy Array

The Numpy arrays can be sorted using the sort () cmd.

In Numpy

Axis =0 denotes Columns

Axis =1denotes Numpy

In Pandas

Axis =0 denotes Rows

Axis =1denotes columns

```
In [39]: arr = np.random.randint(1,10,size=[10,5])
         arr

Out[39]: array([[5, 6, 2, 9, 1],
                [3, 7, 1, 9, 6],
                [5, 8, 9, 6, 3],
                [3, 2, 8, 4, 6],
                [7, 5, 8, 4, 1],
                [1, 2, 6, 1, 2],
                [6, 8, 9, 7, 8],
                [1, 7, 1, 5, 3],
                [8, 9, 6, 2, 7],
                [4, 4, 3, 6, 3]])

In [40]: np.sort(arr,axis=0) # Columnwise

Out[40]: array([[1, 2, 1, 1, 1],
                [1, 2, 1, 2, 1],
                [3, 4, 2, 4, 2],
                [3, 5, 3, 4, 3],
                [4, 6, 6, 5, 3],
                [5, 7, 6, 6, 3],
                [5, 7, 8, 6, 6],
                [6, 8, 8, 7, 6],
                [7, 8, 9, 9, 7],
                [8, 9, 9, 9, 8]])

In [41]: np.sort(arr,axis=1) #Rowwise

Out[41]: array([[1, 2, 5, 6, 9],
                [1, 3, 6, 7, 9],
                [3, 5, 6, 8, 9],
                [2, 3, 4, 6, 8],
                [1, 4, 5, 7, 8],
                [1, 1, 2, 2, 6],
                [6, 7, 8, 8, 9],
                [1, 1, 3, 5, 7],
                [2, 6, 7, 8, 9],
                [3, 3, 4, 4, 6]])

In [43]: # Keeping a Row intact
         sorted_index = arr[:,0].argsort()

In [44]: arr[sorted_index]

Out[44]: array([[1, 2, 6, 1, 2],
                [1, 7, 1, 5, 3],
                [3, 7, 1, 9, 6],
                [3, 2, 8, 4, 6],
                [4, 4, 3, 6, 3],
                [5, 6, 2, 9, 1],
                [5, 8, 9, 6, 3],
                [6, 8, 9, 7, 8],
                [7, 5, 8, 4, 1],
                [8, 9, 6, 2, 7]])
```

Working with Numpy datetime64 submodule

Numpy has module called datetime64 to handle with dates

```
In [45]: d = np.datetime64('2019-06-02 23:10:00')
         d

Out[45]: numpy.datetime64('2019-06-02T23:10:00')

In [46]: d + 1000

Out[46]: numpy.datetime64('2019-06-02T23:26:40')

In [47]: 16*60 + 40

Out[47]: 1000

In [48]: oneday = np.timedelta64(1,'D')
         oneday

Out[48]: numpy.timedelta64(1,'D')

In [49]: d + oneday

Out[49]: numpy.datetime64('2019-06-03T23:10:00')

In [50]: oneminute = np.timedelta64(1,'m')
         oneminute

Out[50]: numpy.timedelta64(1,'m')

In [51]: d + oneminute

Out[51]: numpy.datetime64('2019-06-02T23:11:00')

In [57]: dates = np.arange(np.datetime64('2019-06-02'),np.datetime64('2020-06-02'),5)
         dates

Out[57]: array(['2019-06-02', '2019-06-07', '2019-06-12', '2019-06-17',
                '2019-06-22', '2019-06-27', '2019-07-02', '2019-07-07',
                '2019-07-12', '2019-07-17', '2019-07-22', '2019-07-27',
                '2019-08-01', '2019-08-06', '2019-08-11', '2019-08-16',
                '2019-08-21', '2019-08-26', '2019-08-31', '2019-09-05',
                '2019-09-10', '2019-09-15', '2019-09-20', '2019-09-25',
                '2019-09-30', '2019-10-05', '2019-10-10', '2019-10-15',
                '2019-10-20', '2019-10-25', '2019-10-30', '2019-11-04',
                '2019-11-09', '2019-11-14', '2019-11-19', '2019-11-24',
                '2019-11-29', '2019-12-04', '2019-12-09', '2019-12-14',
                '2019-12-19', '2019-12-24', '2019-12-29', '2020-01-03',
                '2020-01-08', '2020-01-13', '2020-01-18', '2020-01-23',
                '2020-01-28', '2020-02-02', '2020-02-07', '2020-02-12',
                '2020-02-17', '2020-02-22', '2020-02-27', '2020-03-03',
                '2020-03-08', '2020-03-13', '2020-03-18', '2020-03-23',
                '2020-03-28', '2020-04-02', '2020-04-07', '2020-04-12',
                '2020-04-17', '2020-04-22', '2020-04-27', '2020-05-02',
                '2020-05-07', '2020-05-12', '2020-05-17', '2020-05-22',
                '2020-05-27', '2020-06-01'], dtype='datetime64[D]')
```

Numpy Advanced Functions

There are several in-built advanced Numpy functions like vectorize () which makes handling multi-dimensional arrays easier.

```
In [58]: # Vectorize()
```

```
In [59]: def foo(x):
             if x % 2 == 0:
                 return x ** 2
             else:
                 return x / 2
         foo(10)
```

```
Out[59]: 100
```

```
In [60]: foo(11)
```

```
Out[60]: 5.5
```

```
In [61]: foo_v = np.vectorize(foo,otypes=[float])
         print(arr)
         print()
         foo_v(arr)
```

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

```
Out[61]: array([[ 2.5, 36. ,  4. ,  4.5,  0.5],
               [ 1.5,  3.5,  0.5,  4.5, 36. ],
               [ 2.5, 64. ,  4.5, 36. ,  1.5],
               [ 1.5,  4. , 64. , 16. , 36. ],
               [ 3.5,  2.5, 64. , 16. ,  0.5],
               [ 0.5,  4. , 36. ,  0.5,  4. ],
               [36. , 64. ,  4.5,  3.5, 64. ],
               [ 0.5,  3.5,  0.5,  2.5,  1.5],
               [64. ,  4.5, 36. ,  4. ,  3.5],
               [16. , 16. ,  1.5, 36. ,  1.5]])
```

```
In [ ]:
```

Bibliography

1. TakenMind Course [Udemy]
2. Google
3. Stack Overflow
4. Wikipedia