Creating an array

In [2]:
```
import numpy as np
```

In [6]:
```
list1 = [1,2,3,4,5]

array_from_list = np.array(list1)  # or np.array([1,2,3,4,5]). This is 1D array
array_from_list
```

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

In [34]:
```
array_from_tuple = np.array((1,2,3,4,5))
array_from_tuple
```

Out[34]:
```
array([1, 2, 3, 4, 5])
```

Numpy arrays are homogenous. That is all the elements in the array have to be of same datatype

In [9]:
```
array_2d = np.array([[1,2,3], [4,5,6]])  #Pass a list of lists if you have to create 2D Arra
ys
array_2d
```

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

Other methods of creating an array are as follows:

np.arange(arg1, arg2, step) - To create an evenly spaced array. Note that arg2 is excluded from the result.

In [13]:
```
np.arange(1 , 20, 2)
```

Out[13]:
```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

In [12]:
```
np.arange(20)  #if just one arg is passed it takes that arg as 'arg2' and step by default as
1. Also note that it starts from 0
```

Out[12]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

np.zeros(arg) - To create an array of 'arg' number of zeroes

In [16]:
```
np.zeros(5)   # By default it will return float values.
```

Out[16]:
```
array([0., 0., 0., 0., 0.])
```

In [18]:
```
np.zeros(5, dtype= int)   # we can change dtype by passing arg as 'dtype'
```

Out[18]:
```
array([0, 0, 0, 0, 0])
```

In [19]:
```
np.ones(5, dtype= int)   # similarly ones
```

Out[19]:
```
array([1, 1, 1, 1, 1])
```

In [37]:
```
np.ones((5,4))    # Similarly we can creat 2D arrays for zeros.
```

Out[37]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

np.linspace(arg1, arg2, arg3) - To create an array of specified evenly spaced numbers between given intervals. Please note the 'arg1' and 'arg2' both are included. 'arg3' is number of elements you want in the result.

In [20]:
```
np.linspace(1, 20, 10)
```

Out[20]:
```
array([ 1.        ,  3.11111111,  5.22222222,  7.33333333,  9.44444444,
       11.55555556, 13.66666667, 15.77777778, 17.88888889, 20.        ])
```

np.eye(arg) - To create (arg*arg) identity matrix

In [21]:
```
np.eye(2)
```

Out[21]:
```
array([[1., 0.],
       [0., 1.]])
```

To create an array of random variables:

1 - From Uniform Distribution i.e between 0 and 1 \ 2 - From standard Normal Distribution \ 3 - Random integers in given interval

In [22]:
```
np.random.rand(5)  # 'arg' is the number of values needed from UNIFORM DISTRIBUTION
```

Out[22]:
```
array([0.49677541, 0.90801866, 0.24105841, 0.31583176, 0.70368364])
```

In [24]:
```
np.random.rand(5,4)   # To get 2D array
```

Out[24]:
```
array([[0.40060315, 0.69951369, 0.76731604, 0.99228396],
       [0.7105502 , 0.11386213, 0.50807106, 0.02357548],
       [0.64256844, 0.82927103, 0.67255387, 0.54773636],
       [0.62309893, 0.16661983, 0.23692307, 0.54939187],
       [0.23615707, 0.37658835, 0.7509972 , 0.68237739]])
```

In [25]:
```
np.random.randn(5)  #'arg' is number of values needed from STANDARD NORMAL DISTRIBUTION
```

Out[25]:
```
array([ 0.34920857,  0.96214072, -0.09344336, -0.33045297,  0.61370805])
```

In [26]:
```
np.random.randn(5,4)
```

Out[26]:
```
array([[ 0.57274397, -0.3662771 ,  0.43532685,  0.31563745],
       [-0.36493581, -0.12328378,  2.06373906, -1.18583175],
       [-1.94399592, -0.19296823,  0.13993823, -0.56171989],
       [-0.12216412, -0.63250391,  1.20629521, -0.27698503],
       [-0.30765962, -0.00966756, -0.69163603, -0.32919575]])
```

In [27]:
```
np.random.randint(0,20,5)   # To get 5 random integers from 0 to 19(arg2 exculded). If arg3
 not specified then 1 value is returned
```

Out[27]:
```
array([ 1, 18,  3, 18,  6])
```

In [12]:
```
np.random.random((3,3))   # Pass a tuple if you want multi dimensional array
```

Out[12]:
```
array([[0.55462432, 0.35920371, 0.67345534],
       [0.82870826, 0.21924166, 0.01798282],
       [0.49697061, 0.38994154, 0.41810924]])
```

np.full(arg1 , arg2) - To create an array of 'arg2' int/float 'arg1' times

In [39]:
```
np.full (5 , 10)
```

Out[39]:
```
array([10, 10, 10, 10, 10])
```

In [40]:
```
np.full((3,4), 10)
```

Out[40]:
```
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

np.tile(array, arg2) - This function is used to create a new array by repeating an existing array for 'arg2' number of times

In [42]:
```
np.tile(np.array([1,2,3,4]), 2)
```

Out[42]:
```
array([1, 2, 3, 4, 1, 2, 3, 4])
```

In [45]:
```
np.tile(np.array([1,2,3,4]), (2,2))  # Note:array multipled by 2 in row and in column also
```

Out[45]:
```
array([[1, 2, 3, 4, 1, 2, 3, 4],
       [1, 2, 3, 4, 1, 2, 3, 4]])
```

Operations that we can do on Numpy Arrays:

Calculate element-wise product of 2 arrays:

In [30]:
```
arr_1 = np.array([1,2,3,4])   #For lists, we will have to use lambda with map function
arr_2 = np.array([5,6,7,8])
arr_1*arr_2
```

Out[30]:
```
array([ 5, 12, 21, 32])
```

Rounding off the values in the array:

np.round(array_name, decimals)

In [127]:
```
random=np.random.rand(5,4)
random
```

Out[127]:
```
array([[0.89257429, 0.7485368 , 0.71142753, 0.79757671],
       [0.62494539, 0.92301408, 0.1958532 , 0.37862384],
       [0.19746691, 0.11181457, 0.5906272 , 0.16165135],
       [0.9805932 , 0.71722276, 0.9996579 , 0.50816858],
       [0.98857773, 0.33417449, 0.45141936, 0.97898859]])
```

In [128]:
```
np.round(random,2)
```

Out[128]:
```
array([[0.89, 0.75, 0.71, 0.8 ],
       [0.62, 0.92, 0.2 , 0.38],
       [0.2 , 0.11, 0.59, 0.16],
       [0.98, 0.72, 1.  , 0.51],
       [0.99, 0.33, 0.45, 0.98]])
```

Squared array:

In [32]:
```
arr_1**2   # For lists we can do by list comprehension
```

Out[32]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

Reshaping Arrays : array_name.reshape(rows,cols)

In [75]:
```
np.arange(20).reshape(4,5)
```

Out[75]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Stacking and splitting arrays :

1 - np.hstack() - For horizontal stacking number of rows must be same

2 - np.vstack() - For vertical stacking number of cols must be same

Alternative syntax : np.stack((arr1,arr2) , axis=1/0)

In [77]:
```
arr_1
```

Out[77]:
```
array([1, 2, 3, 4])
```

In [78]:
```
arr_2
```

Out[78]:
```
array([5, 6, 7, 8])
```

In [81]:
```
np.hstack((arr_1,arr_2))    # Pass arrays as a tuple or it will throw an error
```

Out[81]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In [82]:
```
np.vstack((arr_1,arr_2))
```

Out[82]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Return indices of non-zero elements

In [6]:
```
a=np.array([1,3,0,0,5])
nz=np.nonzero(a)
nz                              # Returns array of indices
```

Out[6]:
```
(array([0, 1, 4], dtype=int64),)
```

Extract a diagonal of matrix

In [15]:
```
np.diag(np.arange(4),k=0)  #  It will put values of array in the diagonal
```

Out[15]:
```
array([[0, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3]])
```

In [18]:
```
d=np.diag(np.arange(4),k=-1)
d#If k<0, the diagonal below original diagonal will take values of array
```

Out[18]:
```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0]])
```

Changing dtype of array

In [20]:
```
d.dtype
```

Out[20]:
```
dtype('int32')
```

In [24]:
```
f=d.astype(float)
f.dtype
f
```

Out[24]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 2., 0., 0.],
       [0., 0., 0., 3., 0.]])
```

Structure and content of arrays

In [ ]:

array_name.shape : It is an attribute to determine number of rows and columns(rows,columns)

In [47]:
```
array_2d.shape
```

Out[47]:
```
(2, 3)
```

array_name.dtype : It determines the datatype

In [48]:
```
array_2d.dtype
```

Out[48]:
```
dtype('int32')
```

In [55]:
```
heter=np.array([1,2,3,'4'])
heter.dtype
```

Out[55]:
```
dtype('<U11')
```

array_name.ndim : It gives the dimension or the axes of the array

In [50]:
```
array_from_list.ndim
```

Out[50]:
```
1
```

In [51]:
```
array_2d.ndim
```

Out[51]:
```
2
```

array_name.itemsize : It determines the memory used by each element of an array in bytes.

In [52]:
```
array_from_list.itemsize
```

Out[52]:
```
4
```

In [53]:
```
array_2d.itemsize
```

Out[53]:
```
4
```

In [56]:
```
heter.itemsize
```

Out[56]:
```
44
```

Transpose of a multidimensional array

In [57]:
```
array_2d.T
```

Out[57]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Slicing and Dicing through Arrays

In [58]:
```
array_from_list
```

Out[58]:
```
array([1, 2, 3, 4, 5])
```

In [60]:
```
array_from_list[2]  # Same as in lists
```

Out[60]:
```
3
```

In [62]:
```
array_from_list[[1,2,3]]    # To fetch multiple elements pass list of indices as an arg
```

Out[62]:
```
array([2, 3, 4])
```

In [63]:
```
array_from_list[2:]        # 3rd element onwards
```

Out[63]:
```
array([3, 4, 5])
```

In [64]:
```
array_from_list[:2]      #Upto 2nd element
```

Out[64]:
```
array([1, 2])
```

In [65]:
```
array_from_list[2:4]
```

Out[65]:
```
array([3, 4])
```

In [66]:
```
array_2d
```

Out[66]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [67]:
```
array_2d[1,2]
```

Out[67]:
```
6
```

In [68]:
```
array_2d[1, :]           # to fetch 2nd row and all columns
```

Out[68]:
```
array([4, 5, 6])
```

In [69]:
```
array_2d[:,1]           #Similarly fetching all rows and 2nd column
```

Out[69]:
```
array([2, 5])
```

In [70]:
```
array_2d[:, 0:2]        # all rows and columns in range o to 2
```

Out[70]:
```
array([[1, 2],
       [4, 5]])
```

In [72]:
```
array_2d[: , (0,2)]     # all rows and 1st and 3rd column..If you want separate rows/cols, p
ass indices in a tuple
```

Out[72]:
```
array([[1, 3],
       [4, 6]])
```

In [73]:
```
for row in array_2d:        # We can also iterate using for loop, but numpy arrays are not m
eant to be iterated using for loops
    print(row)
```

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

Basic Mathematical operations on Arrays

In [83]:
```
arr_1
```

Out[83]:
```
array([1, 2, 3, 4])
```

In [84]:
```
arr_1*2
```

Out[84]:
```
array([2, 4, 6, 8])
```

In [85]:
```
arr_1/2
```

Out[85]:
```
array([0.5, 1. , 1.5, 2. ])
```

In [86]:
```
arr_1+2
```

Out[86]:
```
array([3, 4, 5, 6])
```

In [87]:
```
arr_1-2
```

Out[87]:
```
array([-1,  0,  1,  2])
```

In [88]:
```
arr_1=arr_1**2
arr_1
```

Out[88]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

In [91]:
```
np.sqrt(arr_1)
```

Out[91]:
```
array([1., 2., 3., 4.])
```

In [92]:
```
np.exp(arr_1)
```

Out[92]:
```
array([2.71828183e+00, 5.45981500e+01, 8.10308393e+03, 8.88611052e+06])
```

In [93]:
```
np.sin(arr_1)
```

Out[93]:
```
array([ 0.84147098, -0.7568025 ,  0.41211849, -0.28790332])
```

In [94]:
```
np.cos(arr_1)
```

Out[94]:
```
array([ 0.54030231, -0.65364362, -0.91113026, -0.95765948])
```

In [96]:
```
np.arcsin(arr_1)      #Opposite of sine
```

```
C:\Users\Pratik\anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: invalid
value encountered in arcsin
  """Entry point for launching an IPython kernel.
```

Out[96]:
```
array([1.57079633,        nan,        nan,        nan])
```

In [98]:
```
np.degrees(1)          # to convert radians in degrees
```

Out[98]:
```
57.29577951308232
```

In [100]:
```
np.max(arr_1)          # gives max value in array
```

Out[100]:
```
16
```

In [101]:
```
np.argmax(arr_1)        #Gives index position of max value. Note: this method doesn't work o
n multidimensional arrays
```

Out[101]:
```
3
```

In [102]:
```
np.min(arr_1)
```

Out[102]:
```
1
```

In [103]:
```
np.argmin(arr_1)      # Note: this method doesn't work on multidimensional arrays
```

Out[103]:
```
0
```

In [105]:
```
np.square(arr_1)
```

Out[105]:
```
array([  1,  16,  81, 256], dtype=int32)
```

In [106]:
```
np.max(array_2d)
```

Out[106]:
```
6
```

In [108]:
```
np.log(arr_1)
```

Out[108]:
```
array([0.        , 1.38629436, 2.19722458, 2.77258872])
```

User-defined functions on arrays

If you want to apply a specific function on numpy array, you can use np.vectorize() method

np.vectorize(function)

In [109]:
```
f = np.vectorize(lambda x: x/(x+1))   # 'f' 's datatype is an object.
f(arr_1)
```

Out[109]:
```
array([0.5       , 0.8       , 0.9       , 0.94117647])
```

In [112]:
```
f(array_2d)                # You can apply this vectorised function multiple times ahead.
```

Out[112]:
```
array([[0.5       , 0.66666667, 0.75      ],
       [0.8       , 0.83333333, 0.85714286]])
```

Basic Linear Algebra Operations

In [120]:
```
array_inv = np.array([[1,2,3],[4,5,6],[7,8,9]])
array_inv
```

Out[120]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [121]:
```
np.linalg.inv(array_inv)               # Inverse of matrix
```

Out[121]:
```
array([[ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15],
       [-6.30503948e+15,  1.26100790e+16, -6.30503948e+15],
       [ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15]])
```

In [122]:
```
np.linalg.det(array_inv)             # Determinant of matrix
```

Out[122]:
```
-9.51619735392994e-16
```

In [123]:
```
np.linalg.eig(array_inv)         # Eigen values and eigen vectors of matrix
```

Out[123]:
```
(array([ 1.61168440e+01, -1.11684397e+00, -9.75918483e-16]),
 array([[-0.23197069, -0.78583024,  0.40824829],
        [-0.52532209, -0.08675134, -0.81649658],
        [-0.8186735 ,  0.61232756,  0.40824829]]))
```

In [124]:
```
np.dot(array_inv[(0,1),(0,1)],array_inv[(1,2),(1,2)])      # dot product of a matrix
```

Out[124]:
```
50
```

Broadcasting

when we subset an array using '[:]', we just see the sub-view of that array and note that data is not copied in the subset

In [131]:
```
arr_1[0:2]
```

Out[131]:
```
array([1, 4], dtype=int32)
```

In [133]:
```
arr_1[0:2] = 100      #Broadcasting. While broadcasting, original array gets updated
arr_1
```

Out[133]:
```
array([100, 100,   9,  16], dtype=int32)
```

Saving and Loading an Array

1 - np.save('file_name.npy', array_name) : This stores a single array.

2 - np.load('filename.npy') : This loads the saved array

3 - np.savez('filename.npz', 'a'=arr1, 'b'=arr2) : This saves the 2 arrays in a zip file.

4 - abc = np.load('filename.npz') abc['a'] : loads arr1 abc['b'] : loads arr2

5 - np.savetxt('filename.txt', arrname, delimiter=',')

5 - np.loadtxt ('filename.txt', delimiter =',')

np.where() in detail

Syntax : np.where (condition, x, y) : If condition is true, it returns 'x' else returns 'y'. It return a new matrix

In [136]:
```
matrix = np.linspace(0,20,9).reshape(3,3)
matrix
```

Out[136]:
```
array([[ 0. ,  2.5,  5. ],
       [ 7.5, 10. , 12.5],
       [15. , 17.5, 20. ]])
```

In [144]:
```
matrix2=np.where (matrix<10 , 1, 100)
matrix2
```

Out[144]:
```
array([[  1,   1,   1],
       [  1, 100, 100],
       [100, 100, 100]])
```

In [141]:
```
np.where (matrix==1,100,matrix)
```

Out[141]:
```
array([[ 0. ,  2.5,  5. ],
       [ 7.5, 10. , 12.5],
       [15. , 17.5, 20. ]])
```

In [145]:
```
np.where((matrix2==1) & (matrix2==100), -1,10)  #Multple filters for a single element
```

Out[145]:
```
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]])
```

In [146]:
```
np.where(matrix2==1,matrix2*100,matrix2)     # To process elements that satisy condition
```

Out[146]:
```
array([[100, 100, 100],
       [100, 100, 100],
       [100, 100, 100]])
```

Print Numpy Version and config

In [3]:
```
np.__version__
```

Out[3]:
```
'1.18.1'
```

In [4]:
```
np.show_config()
```

```
blas_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/Pratik/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.11
7\\windows\\mkl', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117
\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_201
9.0.117\\windows\\mkl\\lib', 'C:/Users/Pratik/anaconda3\\Library\\include']
blas_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/Pratik/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.11
7\\windows\\mkl', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117
\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_201
9.0.117\\windows\\mkl\\lib', 'C:/Users/Pratik/anaconda3\\Library\\include']
lapack_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/Pratik/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.11
7\\windows\\mkl', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117
\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_201
9.0.117\\windows\\mkl\\lib', 'C:/Users/Pratik/anaconda3\\Library\\include']
lapack_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/Pratik/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.11
7\\windows\\mkl', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117
\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_201
9.0.117\\windows\\mkl\\lib', 'C:/Users/Pratik/anaconda3\\Library\\include']
```

In [ ]: