In [1]:

```
import numpy as np
```

In [86]:

```
a = np.array([1, 2, 3,4, 5]) # Create a rank 1 array
print(a)
print(type(a))
print(a.ndim, a.shape)
print(a.size, a.dtype)
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
1 (5,)
5 int32
```

In [87]:

```
print(a)
print(a[0], a[1], a[2])
print(a[-3], a[-4], a[-1])
```

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

In [90]:

```
a[0] = 5 # Change an element of the array
print(a)
```

```
[5 2 3 4 5]
```

In [6]:

```
b = np.array([[1,2],[3,4]]) # Create a rank 2 array
print(b)
print(type(b))
print(b.ndim, b.shape, b.size, b.dtype)
print(b[0, 0], b[0, 1], b[1, 1])
```

```
[[1 2]
 [3 4]]
<class 'numpy.ndarray'>
2 (2, 2) 4 int32
1 2 4
```

In [94]:

```
a = np.zeros((2,2,2)) # Create an array of all zeros
print(a)
```

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

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

In [100]:

```
b = np.ones((2,1)) # Create an array of all ones
print(b)
```

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

In [103]:

```
# Array with int dtype
a = np.zeros((2, 2), dtype=int)
print(a)
```

```
[[0 0]
 [0 0]]
```

In [105]:

```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

In [108]:

```
d = np.empty((4,2))
print(d)
```

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

In [109]:

```
np.arange(10)
```

Out[109]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [111]:

```
e = np.arange(2, 25, 2)
print(e)
print(e.ndim, e.shape, e.size, e.dtype)
```

```
[ 2  4  6  8 10 12 14 16 18 20 22 24]
1 (12,) 12 int32
```

In [113]:

```
# To reshape the array
f = e.reshape(4, 3)
print(f)
print(f.ndim, f.shape, f.size, f.dtype)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]
 [20 22 24]]
2 (4, 3) 12 int32
```

In [116]:

```
f = e.reshape(4, -1) # -1 means "whatever is needed" or "Unknown dimension"
print(f) # useful when reshaping to multi-dim (3 or more).
print(f.ndim, f.shape, f.size, f.dtype)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]
 [20 22 24]]
2 (4, 3) 12 int32
```

In [120]:

```
# To flatten your array into a 1D array
g=f.flatten()
print(g)
print(g.ndim, g.shape)
```

```
[ 2  4  6  8 10 12 14 16 18 20 22 24]
1 (12,)
```

In [121]:

```
h = np.linspace(1, 3, 6)
print(h)
```

```
[1.  1.4 1.8 2.2 2.6 3. ]
```

In [123]:

```
i = np.eye(3, dtype=int) # Create a 2x2 identity matrix
print(i)
```

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

In [125]:

```
j = np.random.random((2,2)) # Create an array filled with random values
print(j)
```

```
[[0.14210393 0.32228793]
 [0.60083891 0.97784241]]
```

In [129]:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
# Elementwise sum; both produce the array
print(x + y)
print()
print(np.add(x, y))
```

```
[[ 6  8]
 [10 12]]

[[ 6  8]
 [10 12]]
```

In [130]:

```
# Elementwise difference; both produce the array
print(x - y)
print()
print(np.subtract(x, y))
```

```
[[-4 -4]
 [-4 -4]]

[[-4 -4]
 [-4 -4]]
```

In [131]:

```
# Elementwise product; both produce the array
print(x * y)
print()
print(np.multiply(x, y))
```

```
[[ 5 12]
 [21 32]]

[[ 5 12]
 [21 32]]
```

In [132]:

```
# Elementwise division; both produce the array
print(x/y)
print()
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]

[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

In [133]:

```
# Elementwise Square; produces the array
print(x)
print(x**2)
print()
print(np.square(x))
```

```
[[1 2]
 [3 4]]
[[ 1  4]
 [ 9 16]]

[[ 1  4]
 [ 9 16]]
```

In [134]:

```
# Elementwise square root; produces the array
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

In [140]:

```
# Inner product of 2 vectors;
v = np.array([9,10])
w = np.array([11, 12])
print(v.dot(w))
print(np.dot(v, w))
#You can also use the '@' operator which is equivalent to numpy's dot operator.
print(v @ w)
```

```
219
219
219
```

In [142]:

```
x = np.array([[1,2],[3,4]])
v = np.array([9,10])
# Matrix * Vector product; both produce the rank 1 array
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

In [144]:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
# Matrix * Matrix product; both produce the rank 2 array
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

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

In [148]:

```
x = np.array([[1,2],[3,4]])
print(x)
print()
print(np.sum(x)) # Compute sum of all elements;
print(np.sum(x, axis=0)) # Compute sum of each column;
print(np.sum(x, axis=1)) # Compute sum of each row;
```

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

10
[4 6]
[3 7]
```

In [149]:

```
print(np.max(x))
print(np.max(x, axis=0)) # Compute column wise;
print(np.max(x, axis=1)) # Compute row wise;
```

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

In [150]:

```
np.min(x)
```

Out[150]:

```
1
```

In [151]:

```
np.mean(x)
```

Out[151]:

```
2.5
```

In [152]:

```
np.std(x)
```

Out[152]:

```
1.118033988749895
```

In [153]:

```
print(x)
print("Transpose:\n", x.T)
```

```
[[1 2]
 [3 4]]
Transpose:
 [[1 3]
 [2 4]]
```

In [154]:

```
v = np.array([1,2,3])
print(v, v.shape)
print("Transpose:\n", v.T)
```

```
[1 2 3] (3,)
Transpose:
 [1 2 3]
```

In [156]:

```
v = np.array([[1,2,3]])
print(v, v.shape)
print("Transpose:\n", v.T)
```

```
[[1 2 3]] (1, 3)
Transpose:
 [[1]
 [2]
 [3]]
```

In [158]:

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

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

In [159]:

```
np.flip(x)
```

Out[159]:

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

In [160]:

```
np.flip(x, axis=0)
```

Out[160]:

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

In [161]:

```
np.flip(x, axis=1)
```

Out[161]:

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

In [164]:

```
a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
```

In [167]:

```
print(a)
uni_vals, index, occu_cnt = np.unique(a, return_index=True, return_counts=True)
print(uni_vals)
print(index)
print(occu_cnt)
```

```
[11 11 12 13 14 15 16 17 12 13 11 14 18 19 20]
[11 12 13 14 15 16 17 18 19 20]
[ 0  2  3  4  5  6  7 12 13 14]
[3 2 2 2 1 1 1 1 1 1]
```

In [172]:

```
# We will add the vector v to each row of the matrix x and storing the result in the matrix
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # Create an empty matrix with the same shape as x
# Add the vector v to each row of the matrix x with an explicit 'loop';
for i in range(4):
    y[i, :] = x[i, :] + v
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [174]:

```
vv = np.tile(v, (4, 1)) # Stack 4 copies of v on top of each other
print(vv)
```

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

In [175]:

```
y = x + vv # Add x and vv elementwise
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [180]:

```
import numpy as np
# We will add the vector v to each row of the matrix x, storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
v = np.array([1,0,1])
y = x + v # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [182]:

```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
v = np.array([1,2,3])
# x has shape (2, 3) and v has shape (3,)
# So they broadcast to (2, 3), giving the following matrix:
print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

In [190]:

```
# Adding a vector to each column of a matrix where
# x has shape (2, 3) and w has shape (2,).
x = np.array([[1,2,3], [4,5,6]])
w = np.array([4,5])
# If we transpose x then it has shape (3, 2) and
# can be broadcast against w to yield a result of shape (3, 2);
# Transposing this result yields the final result of shape (2, 3),
# which is the matrix x with the vector 'w' added to each column.
# Gives the following matrix:
print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [192]:

```
x = np.array([[1,2,3], [4,5,6]])
w = np.array([4,5])
# Another solution is to reshape w to be a row vector of shape (2, 1);
# We can then broadcast it directly against x to produce the same output:
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [194]:

```
# Multiply a matrix by a constant:
x = np.array([[1,2,3], [4,5,6]])
print(x * 2)
# x has shape (2, 3). Numpy treats Scalars as arrays of shape();
# these can be broadcast together to shape(2,3), producing the following array:
```

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

In [197]:

```
# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
w = np.array([4,5]) # w has shape (2,)
# To compute an outer product, we first reshape v to be a column vector of␣ shape (3, 1);
# We can then broadcast it against w (2,) to yield an output of shape (3, 2),
# Which is the outer product of v and w:
print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

In [199]:

```
a = np.array([[1, 2], [3, 4]])
b = a # no new object is created
b is a # a and b are two names for the same ndarray object.
```

Out[199]:

```
True
```

In [200]:

```
print(id(a)) # id is a unique identifier of an object
print(id(b))
```

```
2052542319552
2052542319552
```

In [202]:

```
a = np.array([[1, 2, 3], [4, 5, 6]]) # Shape of a is (2,3)
c = a.view()
c is a
```

Out[202]:

```
False
```

In [203]:

```
print(id(a)) # id is a unique identifier of an object
print(id(c))
```

```
2052568348672
2052568346832
```

In [204]:

```
c.base is a # c is a view of the data owned by a
```

Out[204]:

```
True
```

In [205]:

```
c = c.reshape(3, 2) # a's shape will not change
c[1, 0] = 789
print(c)
```

```
[[  1   2]
 [789   4]
 [  5   6]]
```

In [206]:

```
print(a) # a's data changes
```

```
[[  1   2 789]
 [  4   5   6]]
```

In [207]:

```
a = np.array([[1, 2, 3], [4, 5, 6]]) # Shape of a is (2,3)
d = a.copy() # New array object with new data is created
d is a
```

Out[207]:

```
False
```

In [208]:

```
d.base is a # d doesn't share anything with a
```

Out[208]:

```
False
```

In [209]:

```
d[1, 0] = 789
print(d)
```

```
[[  1   2   3]
 [789   5   6]]
```

In [210]:

```
print(a) # a's data changes
```

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

In [211]:

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

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

In [212]:

```
# You can select elements that greater than 2:
print(a[a>2])
```

```
[3 4]
```

In [213]:

```
# You can select elements that are divisible by 2:
print(a[a%2==0])
```

```
[2 4]
```

In [214]:

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1 2 3 4]
# [ 5 6 7 8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Use slicing to pull out the subarray consisting of the
# first 2 rows and columns 1 and 2;
# Resultant is the following array of shape (2, 2):
# [[2 3]
# [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

In [216]:

```
print(a[0, 1])
b[0, 0] = 77 # b[0, 0] is the same piece of data as a[0, 1]
print(a)
```

```
77
[[ 1 77  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [217]:

```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a, a.shape, a.ndim)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]] (3, 4) 2
```

In [222]:

```
# To acess 2nd row
row_r1 = a[1, :] # integer indexing with slices; yields Rank 1 view of the␣ second row of a
row_r2 = a[1]
row_r3 = a[1:2, :] # only slice; yelds Rank 2 view of the second row of a
print(row_r1, row_r1.shape, row_r1.ndim)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape, row_r3.ndim)
```

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

In [223]:

```
# We can see the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape, col_r1.ndim)
print()
print(col_r2, col_r2.shape, col_r2.ndim)
```

```
[ 2  6 10] (3,) 1

[[ 2]
 [ 6]
 [10]] (3, 1) 2
```

In [224]:

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

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

In [228]:

```
# integer array indexing:
print(a[[0, 1, 2], [0, 1, 0]])
# The returned array will have shape (3,)
print()
# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
# print([a[0, 0], a[1, 1], a[2, 0]])
```

```
[1 4 5]

[1 4 5]
```

In [229]:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 3) # Find the elements of a that are bigger than 3;
# this returns a numpy array of Booleans of the same
# shape as a, where each slot of bool_idx tells
print(bool_idx) # whether that element of a is > 3.
```

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

In [230]:

```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])
print()
# We can do all of the above in a single statement like:
print(a[a > 3])
```

```
[4 5 6]

[4 5 6]
```

In [231]:

```
help(max)
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

In [81]:

```
max?
```

In [82]:

```
c?
```

In [83]:

```
a?
```

In [84]:

```
np.random?
```

In [85]:

```
help(np.random)
```

```
DESCRIPTION
    ========================
    Random Number Generation
    ========================

    ==================== =========================================================
========
    Utility functions
    ==============================================================================
========
    random_sample        Uniformly distributed floats over ``[0, 1)``.
    random               Alias for `random_sample`.
    bytes                Uniformly distributed random bytes.
    random_integers      Uniformly distributed integers in a given range.
    permutation          Randomly permute a sequence / generate a random s
equence.
    shuffle              Randomly permute a sequence in place.
    seed                 Seed the random number generator.
    choice               Random sample from 1-D array.
```

In [ ]:

```
```