Basic array operations

In this example, member functions of the arrays were used. Alternatively, standalone functions in the NumPy module can be accessed:

```
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

For most of the routines described below, both standalone and member functions are available.

A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation:

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

It's also possible to find the minimum and maximum element values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

In [32]:
```
a = np.array([2, 4, 3], float)
print(a.sum())

print(a.prod())
```

```
9.0
24.0
```

In [33]:
```
np.array([2, 1, 9], float)
print(a.mean())
print(a.var())

print(a.std())
```

```
3.0
0.6666666666666666
0.816496580927726
```

In [34]:
```
a = np.array([2, 1, 9], float)
a.min()
```

Out[34]: 1.0

In [35]:
```
a.max()
```

Out[35]: 9.0

The `argmin` and `argmax` functions return the array indices of the minimum and maximum values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

In [37]:
```
a = np.array([200, 100, 900], float)
print(a.argmin())
print(a.argmax())
```

```
1
2
```

For multidimensional arrays, each of the functions thus far described can take an optional argument `axis` that will perform an operation along only the specified axis, placing the results in a return array:

In [40]:
```
a = np.array([[0, 2], [3, -1], [3, 5]], float)
print(a)
print(a.mean(axis=0))
print(a.mean(axis=1))
print(a.min(axis=1))
print(a.max(axis=0))
```

```
[[ 0.  2.]
 [ 3. -1.]
 [ 3.  5.]]
[2. 2.]
[1. 1. 4.]
[ 0. -1.  3.]
[3. 5.]
```

In [42]:
```
#Unique elements can be extracted from an array:
a = np.array([1, 1, 4, 5, 5, 5, 7], float)
np.unique(a)
```

Out[42]:
```
array([1., 4., 5., 7.])
```

Comparison operators and value testing

Boolean comparisons can be used to compare members elementwise on arrays of equal size. The return value is an array of Boolean `True` / `False` values:

In [43]:
```
a = np.array([1, 3, 0], float)
b = np.array([0, 3, 2], float)
a > b
```

Out[43]:
```
array([ True, False, False])
```

In [44]:
```
a==b
```

Out[44]:
```
array([False,  True, False])
```

In [45]:
```
a<=b
```

Out[45]:
```
array([False,  True,  True])
```

Statistics

In [47]:
```
 a = np.array([1, 4, 3, 8, 9, 2, 3], float)
np.median(a)
```

Out[47]:
```
3.0
```

In [48]:
```
a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
c = np.corrcoef(a)
c
```

Out[48]:
```
array([[1.        , 0.72870505],
       [0.72870505, 1.        ]])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

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

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

1-D Arrays: An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

2-D Arrays: An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

3-D arrays: An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor.

In [10]:
```
array1 = np.array ( [ 1, 2, 3, 4, 5, 6 ] )
print(array1)
print(type(array1))
```

```
[1 2 3 4 5 6]
<class 'numpy.ndarray'>
```

Shape of an array

Shape of an Array The shape of an array is the number of elements in each dimension.

Get the Shape of an Array NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

In [66]:
```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a)
print(a.ndim)
print(a.shape)
print("____________")

print(b)
print(b.ndim)
print(b.shape)
print("____________")

print(c)
print(c.ndim)
print(c.shape)
print("____________")

print(d)
print(d.ndim)
print(d.shape)
```

```
42
0
()
____________
[1 2 3 4 5]
1
(5,)
____________
[[1 2 3]
 [4 5 6]]
2
(2, 3)
____________
[[[1 2 3]
  [4 5 6]]

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

In [65]:
```
import numpy as np
```

```
a = np.array([42])
print(a)
print(a.ndim)
print(a.shape)
print("_____________")
```

```
[42]
1
(1,)
_____________
```

In [ ]:

In [9]:
```
#Creating higher dimensional array:
```

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

Create an array with 5 dimensions using ndmin using a vector with values 11,12,13,14 and verify that last dimension has value 4:

In [39]:
```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=5)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[[[[11 12 13 14]]]]]
Dimension of array : 5
shape of array : (1, 1, 1, 1, 4)
```

In [40]:
```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=4)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[[[11 12 13 14]]]]
Dimension of array : 4
shape of array : (1, 1, 1, 4)
```

In [41]:
```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=3)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[[11 12 13 14]]]
Dimension of array : 3
shape of array : (1, 1, 4)
```

In [42]:
```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=2)
```

```
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[11 12 13 14]]
Dimension of array : 2
shape of array : (1, 4)
```

In [43]:

```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=1)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[11 12 13 14]
Dimension of array : 1
shape of array : (4,)
```

In [44]:

```
import numpy as np

arr = np.array([11, 12, 13, 14], ndmin=0)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[11 12 13 14]
Dimension of array : 1
shape of array : (4,)
```

In [45]:

```
import numpy as np

arr = np.array([4], ndmin=0)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[4]
Dimension of array : 1
shape of array : (1,)
```

In [46]:

```
import numpy as np

arr = np.array(4, ndmin=0)

print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
4
Dimension of array : 0
shape of array : ()
```

In [47]:

```
import numpy as np

arr = np.array([[11,22,33],[11,22,33]])
print(arr)
```

```
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[11 22 33]
 [11 22 33]]
Dimension of array : 2
shape of array : (2, 3)
```

In [50]:

```
import numpy as np

arr = np.array([[11,22,33],[11,22,33],[11,22,33],[11,22,33]],dtype='int32')
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[11 22 33]
 [11 22 33]
 [11 22 33]
 [11 22 33]]
Dimension of array : 2
shape of array : (4, 3)
```

In [51]:

```
import numpy as np

arr = np.array([[[11,22,33],[11,22,33]],[11,22,33],[11,22,33]],dtype='int32')
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-51-9b4ff0e09fda> in <module>
      1 import numpy as np
      2
----> 3 arr = np.array([[[11,22,33],[11,22,33]],[11,22,33],[11,22,33]],dtype='int32'
)
      4 print(arr)
      5 print('Dimension of array :', arr.ndim)

ValueError: setting an array element with a sequence. The requested array has an inh
omogeneous shape after 1 dimensions. The detected shape was (3,) + inhomogeneous par
t.
```

In [68]:

```
import numpy as np

arr = np.array([[1,2,3],[4,5,6]])
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)

print(arr[])
```

```
[[1 2 3]
 [4 5 6]]
Dimension of array : 2
shape of array : (2, 3)
```

In [ ]:

In [52]:

```
import numpy as np

arr = np.array([[11],[12], [13], [14], [15], [16]])
```

```
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[11]
 [12]
 [13]
 [14]
 [15]
 [16]]
Dimension of array : 2
shape of array : (6, 1)
```

In [56]:

```
import numpy as np

arr = np.array([11,12,13,14,15,16])
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[11 12 13 14 15 16]
Dimension of array : 1
shape of array : (6,)
```

In [57]:

```
# arr is given to you
import numpy as np

arr = np.array([[11,22,33],[21,56,45],[67,78,69],[17,18,19]],dtype='int32')
print(arr)
print('Dimension of array :', arr.ndim)
print('shape of array :', arr.shape)
```

```
[[11 22 33]
 [21 56 45]
 [67 78 69]
 [17 18 19]]
Dimension of array : 2
shape of array : (4, 3)
```

change element 78 to 88 in array arr = np.array([[11,22,33],[21,56,45],[67,78,69],[17,18,19]],dtype='int32')

In [62]:

```
print(arr[2,1])
```

```
78
```

In [63]:

```
arr[2,1]=88
```

In [64]:

```
print(arr)
```

```
[[11 22 33]
 [21 56 45]
 [67 88 69]
 [17 18 19]]
```

NumPy functions

1. np.arange()

numpy.arange() is an inbuilt numpy function that returns an ndarray object containing evenly spaced values within a defined interval. For instance, you want to create values from 1 to 10; you can use np.arange() in Python function.

Syntax:

numpy.arange(start, stop, step, dtype) Python NumPy arange Parameters:

Start: Start of interval for np.arange in Python function. Stop: End of interval. Step: Spacing between values. Default step is 1. Dtype: Is a type of array output for NumPy arange in Python.

In [4]:
```
import numpy as np
np.arange(1, 11)
```

Out[4]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

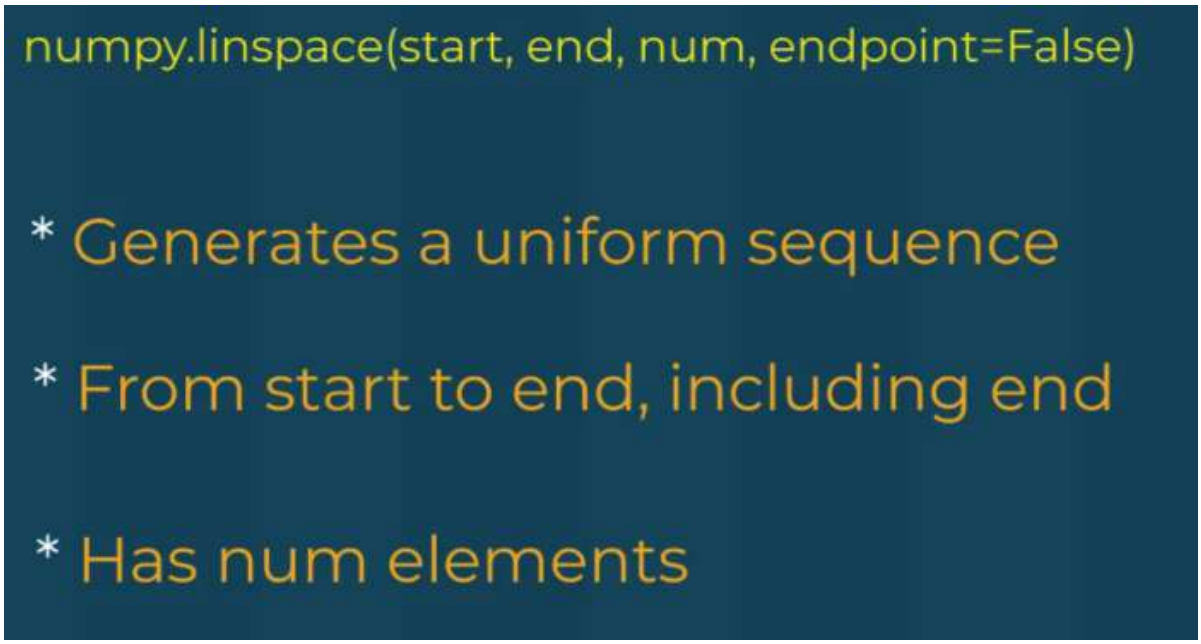
If you want to change the step in this NumPy arange function in Python example, you can add a third number in the parenthesis. It will change the step.

In [7]:
```
import numpy as np
np.arange(1, 14, 4)
```

Out[7]:
```
array([ 1,  5,  9, 13])
```

1. numpy.linspace() Linear Sequence Generator

Gives evenly spaced samples.



```
numpy.linspace(start, stop, num, endpoint)
```

Here,

- Start: Starting value of the sequence
- Stop: End value of the sequence
- Num: Number of samples to generate. Default is 50
- Endpoint: If True (default), stop is the last value. If False, stop value is not included.

In [8]:
```
# For instance, it can be used to create 10 values from 1 to 5 evenly spaced.
import numpy as np
np.linspace(1.0, 5.0, num=10)
```

Out[8]:
```
array([1.        , 1.44444444, 1.88888889, 2.33333333, 2.77777778,
       3.22222222, 3.66666667, 4.11111111, 4.55555556, 5.        ])
```

In [10]:
```
# If you do not want to include the last digit in the interval, you can set endpoint

np.linspace(1.0, 5.0, num=5, endpoint=False)
```

Out[10]:
```
array([1. , 1.8, 2.6, 3.4, 4.2])
```

numpy.digitize

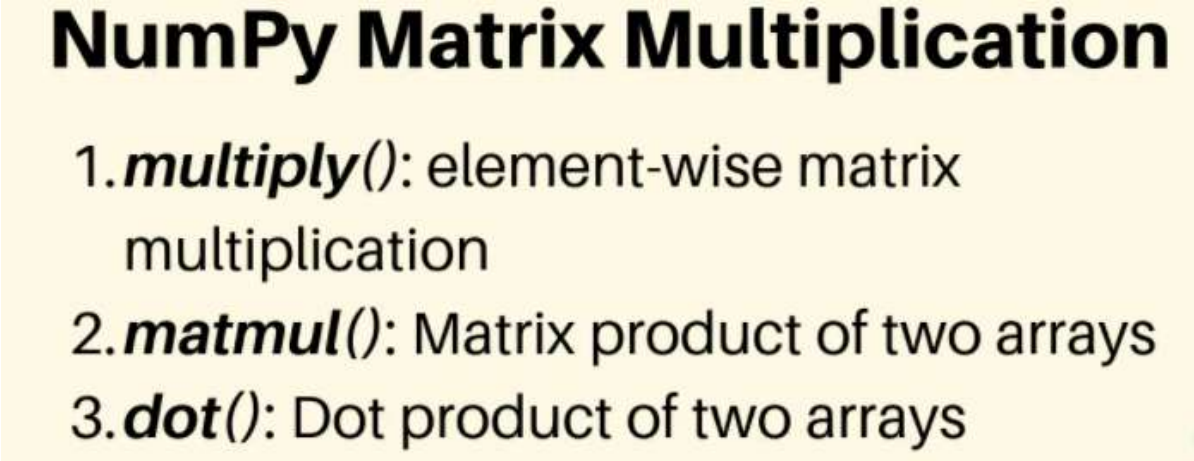
The numpy.digitize(x, bins, right=False) function has two arguments: (1) an input array x, and (2) an array of bins, returning the indices of the bins to which each value in input array belongs.

In [50]:
```
# Input array
x = np.array([0.5])
print(x)

# Bins - 5 bins in total
bins = np.array([0,1,2,3])
print(bins)

# Digitize function - 0.5 belong to the bin 0<= 0.5 <1 - therefore returned index 1
np.digitize(x,bins)
# array([1], dtype=int64)

# The input array can contain several inputs
x = np.array([-0.5,1,3.5])
print(x)
# Digitize function
np.digitize(x,bins)
# array([0, 2, 4], dtype=int64)
```

```
[0.5]
[0 1 2 3]
[-0.5  1.   3.5]
```

Out[50]:
```
array([0, 2, 4], dtype=int64)
```

Vector and matrix mathematics



NumPy provides many functions for performing standard vector and matrix multiplication routines. To perform a dot product,

In [3]:
```
import numpy as np
a = np.array([1, 2, 3], float)
b = np.array([1, 2, 3], float)
np.dot(a,b)
```

Out[3]: 14.0

In [13]:
```
a = np.array([[0, 1], [2, 3]], float)
b = np.array([2, 3], float)

print(a)
print("______")
print(b)
print("______")


#The dot function also generalizes to matrix multiplication:


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

```
[[0. 1.]
 [2. 3.]]
______
[2. 3.]
______
[ 6. 11.]
```

numpy.inner()

In [15]:
```
# Multi-dimensional array example
import numpy as np
a = np.array([[1,2], [3,4]])

print("Array a:")
print(a)
b = np.array([[11, 12], [13, 14]])

print("Array b:")
print(b)
```

```
print("Inner product")
print(np.inner(a,b))
```

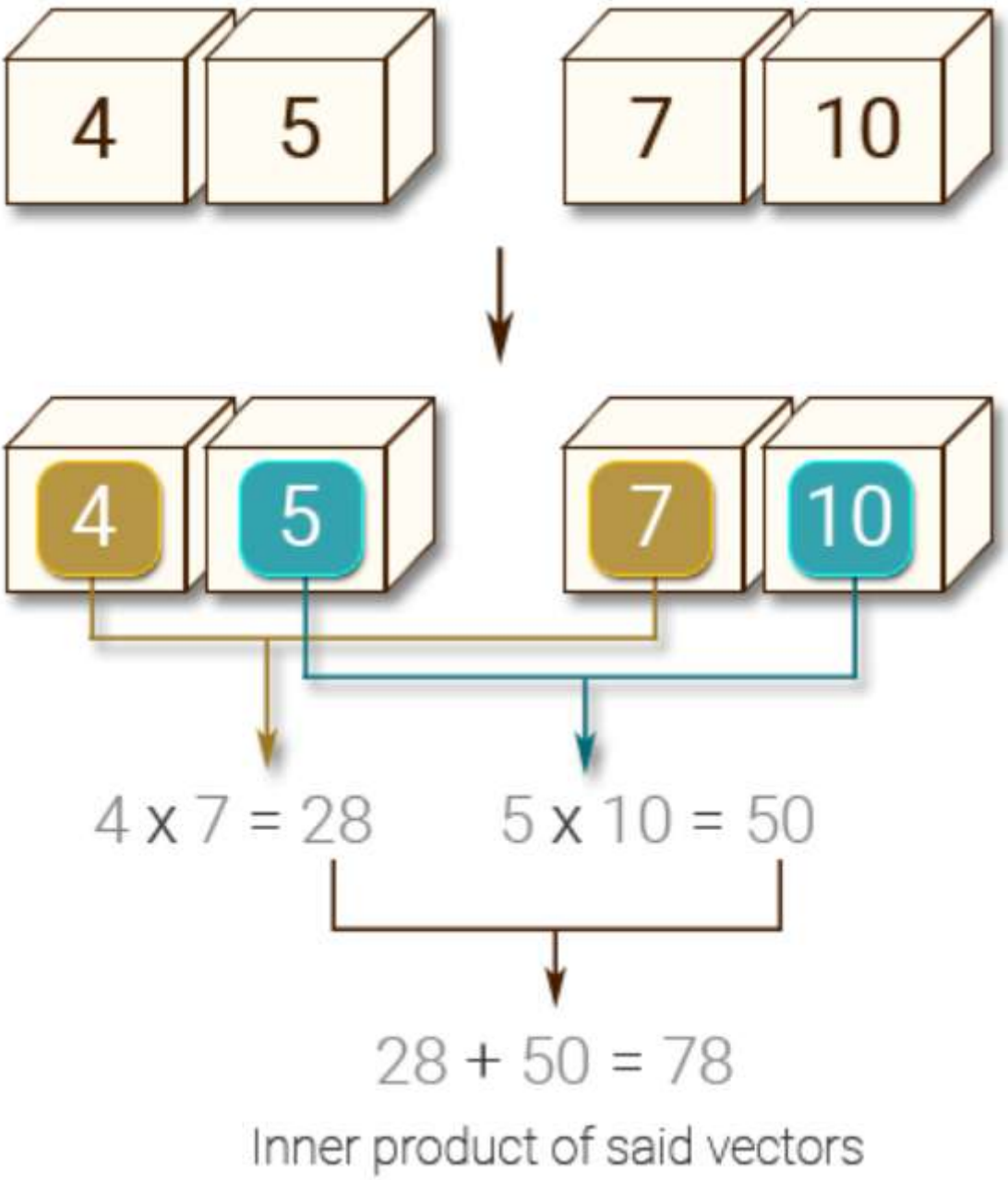
```
Array a:
[[1 2]
 [3 4]]
Array b:
[[11 12]
 [13 14]]
Inner product
[[35 41]
 [81 95]]
```

```
1*11+2*12, 1*13+2*14
3*11+4*12, 3*13+4*14
```

In [16]:

```
print("Inner product")
print(np.inner(a,b))
```

```
Inner product
[[35 41]
 [81 95]]
```



In [17]:

```
import numpy as np
x = np.array([4, 5])
y = np.array([7, 10])
print("Original vectors:")
print(x)
print(y)
print("Inner product of said vectors:")
print(np.dot(x, y))
```

```
Original vectors:
[4 5]
[ 7 10]
Inner product of said vectors:
78
```

In [22]:

```
import numpy as np
a = np.ones(4)
print(a)
print("\n")
b = np.linspace(-1, 2, 4)
print(b)
```

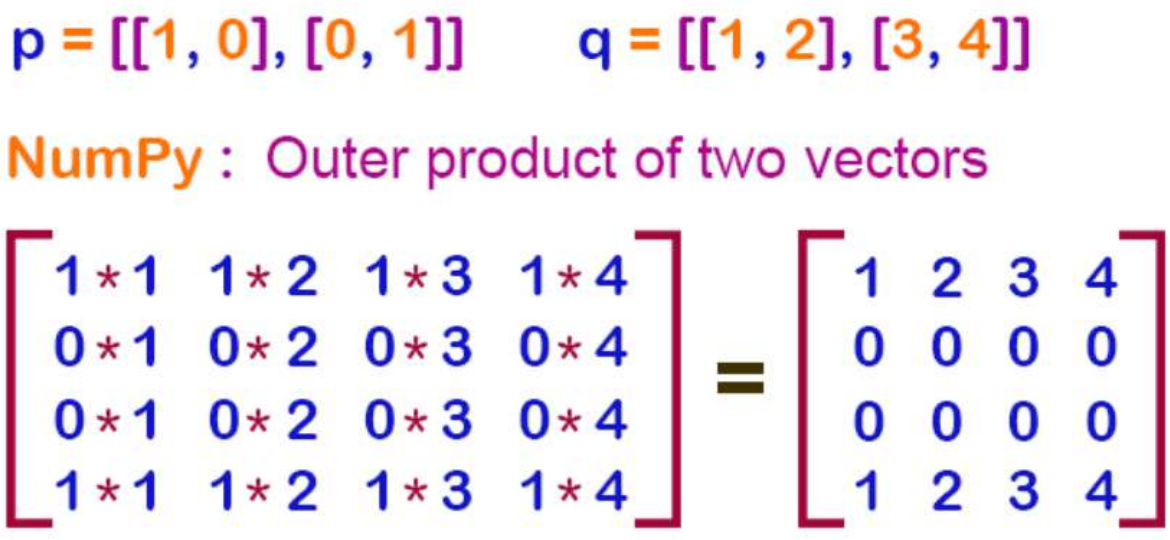
```
print("\n")

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

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


[-1.  0.  1.  2.]


[[-1.  0.  1.  2.]
 [-1.  0.  1.  2.]
 [-1.  0.  1.  2.]
 [-1.  0.  1.  2.]]
```

In [23]:

```
import numpy as np
p = [[1, 0], [0, 1]]
q = [[1, 2], [3, 4]]
print("original matrix:")
print(p)
print(q)
result = np.outer(p, q)
```

```
original matrix:
[[1, 0], [0, 1]]
[[1, 2], [3, 4]]
Outer product of the said two vectors:
[[1 2 3 4]
 [0 0 0 0]
 [0 0 0 0]
 [1 2 3 4]]
```



In [24]:

```
print("Outer product of the said two vectors:")
print(result)
```

```
Outer product of the said two vectors:
[[1 2 3 4]
 [0 0 0 0]
 [0 0 0 0]
 [1 2 3 4]]
```

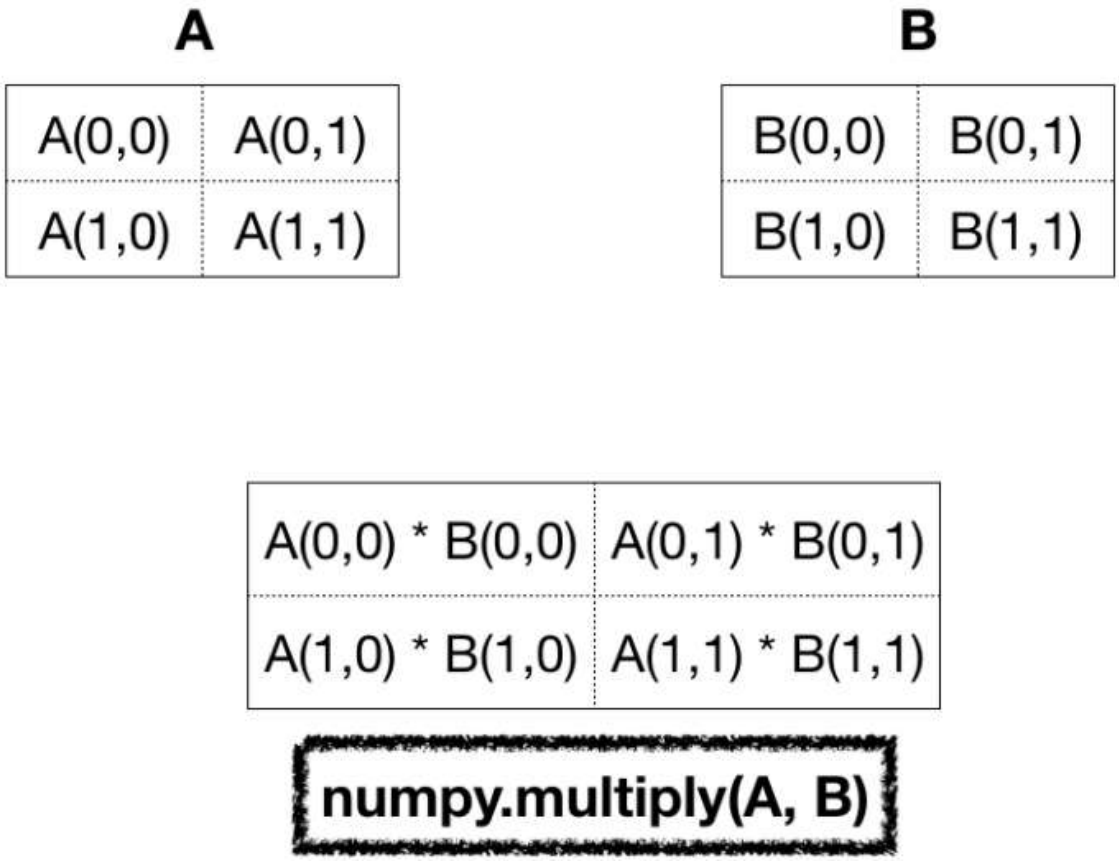
In [25]:

```
import numpy as np
p = [[1, 0], [0, 1]]
q = [[1, 2], [3, 4]]
```

```
print("original matrix:")
print(p)
print(q)
result = np.inner(p, q)
```

```
original matrix:
[[1, 0], [0, 1]]
[[1, 2], [3, 4]]
```

multiply()

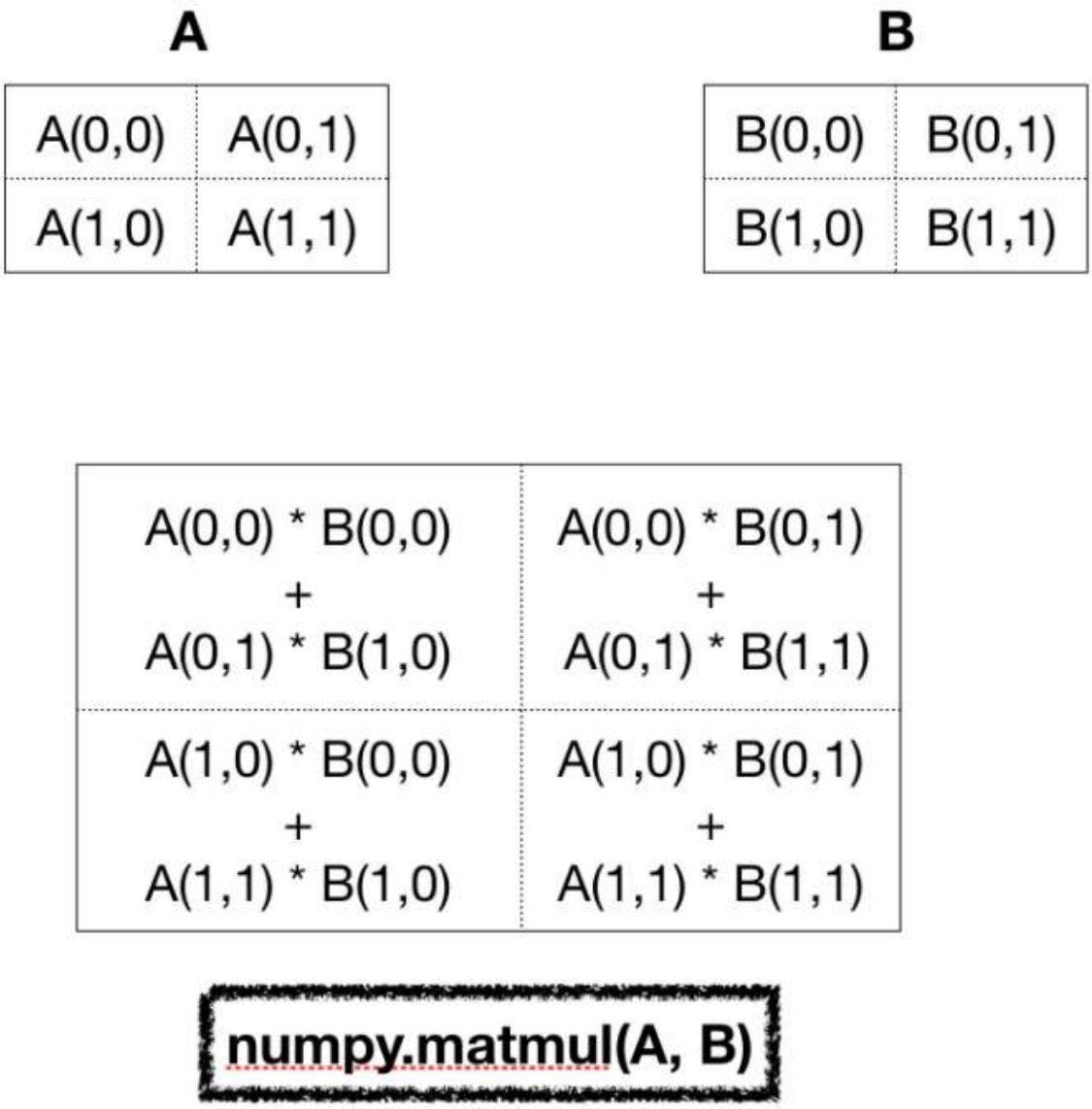


In [12]:

```
import numpy as np

arr1 = np.array([[1, 2],
                 [3, 4]])
arr2 = np.array([[5, 6],
                 [7, 8]])

arr_result = np.multiply(arr1, arr2)

print(arr_result)
```

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

matmul(A,B)



In [13]:

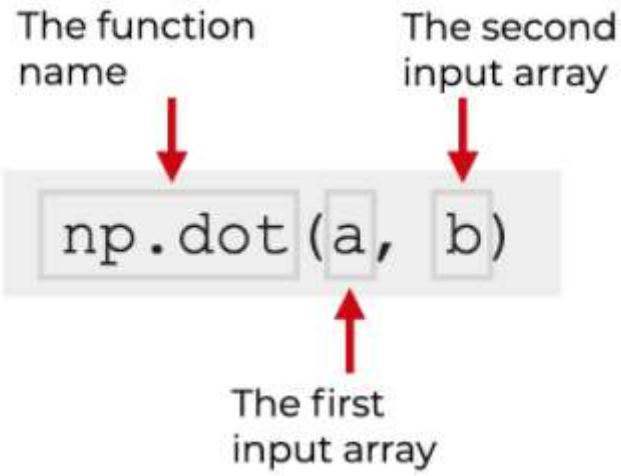
```
import numpy as np

arr1 = np.array([[1, 2],
                 [3, 4]])
arr2 = np.array([[5, 6],
                 [7, 8]])

arr_result = np.matmul(arr1, arr2)

print(f'Matrix Product of arr1 and arr2 is:\n{arr_result}')

arr_result = np.matmul(arr2, arr1)

print(f'Matrix Product of arr2 and arr1 is:\n{arr_result}')
```

```
Matrix Product of arr1 and arr2 is:
[[19 22]
 [43 50]]
Matrix Product of arr2 and arr1 is:
[[23 34]
 [31 46]]
```

np.dot()

There are three broad cases that we'll consider with np.dot:

both inputs are 1D arrays both are 2D arrays one inputs is a scalar and one input is an array



Case 1 IF THE INPUT ARRAYS ARE BOTH 1-DIMENSIONAL ARRAYS, NP.DOT COMPUTES THE VECTOR DOT PRODUCT

Let's say we have two Numpy arrays, **a** and **b**, and each array has 3 values.

$$\mathbf{a} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$

$$\mathbf{a} \cdot \mathbf{b} = \boxed{3 \times 7} + \boxed{4 \times 8} + \boxed{5 \times 9}$$

$$= \boxed{21} + \boxed{32} + \boxed{45}$$

$$= 98$$

In [ ]:

Case 2 IF ONE OF THE INPUTS IS A SCALAR, NP.DOT PERFORMS SCALAR MULTIPLICATION

The second case is when one input is a scalar value r, and one input is a Numpy array, which here we'll call $\mathbf{b}$.

$$r = 2$$

$$\mathbf{b} = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$

If we use Numpy dot on these inputs with the code `np.dot(r,b)` Numpy will perform *scalar multiplication* on the array:

$$r\mathbf{b} = \begin{bmatrix} 2 \times 7 & 2 \times 8 & 2 \times 9 \end{bmatrix}$$

$$= \begin{bmatrix} 14 & 16 & 18 \end{bmatrix}$$

In [ ]:

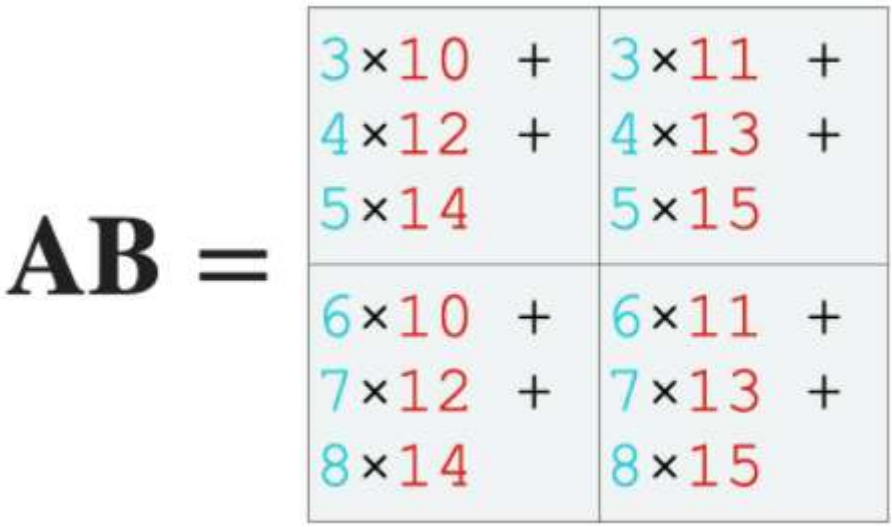
In [ ]:

Case 3 IF BOTH INPUTS ARE 2D ARRAYS, NP.DOT PERFORMS MATRIX MULTIPLICATION

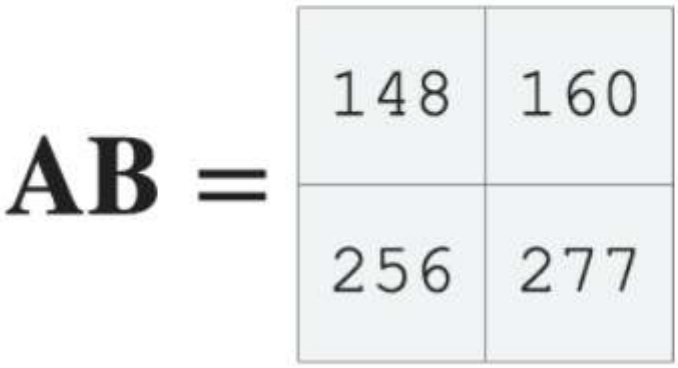
Let's say that you have two 2D arrays, **A** and **B**.

$$\mathbf{A} = \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{bmatrix}$$



And here's the final computed output:



- If both inputs are scalars, `np.dot()` will multiply the scalars together and output a scalar.
- If one input is a scalar and one is an array, `np.dot()` will multiply every value of the array by the scalar (i.e., scalar multiplication).
- If both inputs are 1-dimensional arrays, `np.dot()` will compute the dot product of the inputs
- If both inputs are 2-dimensional arrays, then `np.dot()` will perform matrix multiplication.

In [23]:

```
# EXAMPLE 1: MULTIPLY TWO NUMBERS
import numpy as np
np.dot(2,3)
```

```
#When we call np.dot() with two scalars, it simply multiplies them together.
#Obviously 2 * 3 = 6.
```

Out[23]: 6

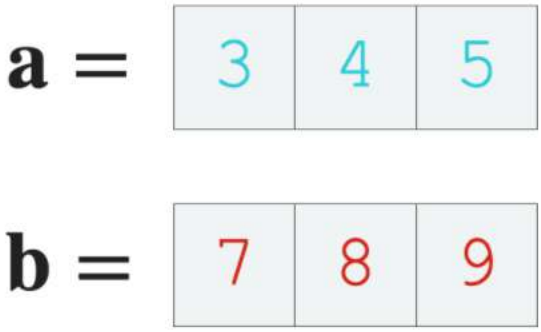
In [26]:
```
#EXAMPLE 2: MULTIPLY A NUMBER AND AN ARRAY
import numpy as np
np.dot(2,[5,6])
#The first argument to the function is the scalar value 2.

#The second argument to the function is the Python list [5,6].

#When we provide a scalar as one input and a list (or Numpy array) as the other inpu
```

Out[26]: array([10, 12])

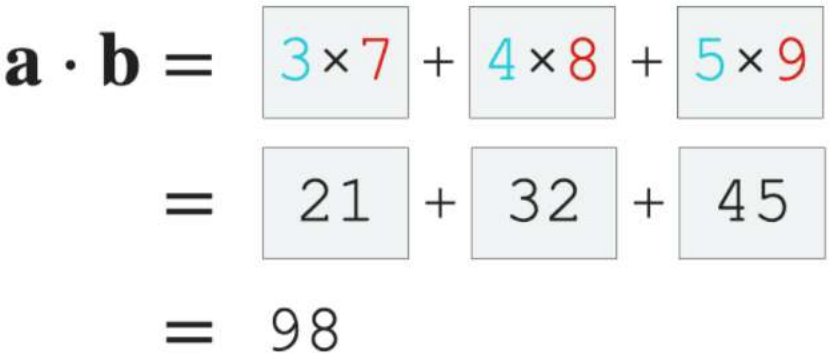
In [27]:
```
#EXAMPLE 3: COMPUTE THE DOT PRODUCT OF TWO 1D ARRAYS
import numpy as np
np.dot([3,4,5],[7,8,9])
```

Out[27]: 98

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$



... it takes the product of the pairwise elements, and then adds them together:



In [30]:
```
#EXAMPLE 4: PERFORM MATRIX MULTIPLICATION ON TWO 2D ARRAYS
import numpy as np
A_array_2d = np.arange(start = 3, stop = 9).reshape((2,3))
B_array_2d = np.arange(start = 10, stop = 16).reshape((3,2))
print(A_array_2d)
print(B_array_2d)
np.dot(A_array_2d, B_array_2d)
```

```
[[3 4 5]
 [6 7 8]]
[[10 11]
 [12 13]
 [14 15]]
```

Out[30]:
```
array([[148, 160],
       [256, 277]])
```

$$\mathbf{A} = \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{bmatrix}$$

$$\mathbf{AB} = \begin{bmatrix} 3\times10 + 4\times12 + 5\times14 & 3\times11 + 4\times13 + 5\times15 \\ 6\times10 + 7\times12 + 8\times14 & 6\times11 + 7\times13 + 8\times15 \end{bmatrix}$$

$$AB = \begin{bmatrix} 148 & 160 \\ 256 & 277 \end{bmatrix}$$

reference https://www.sharpsightlabs.com/blog/numpy-dot/

QUESTION 1: WHAT'S THE DIFFERENCE BETWEEN NP.DOT() AND NP.MATMUL() ?

In [33]:
```
import numpy as np
np.dot(2,[5,6])
#np.matmul(2,[5,6])
```

Out[33]:
```
array([10, 12])
```

In [34]:
```
import numpy as np
np.dot(2,[5,6])
np.matmul(2,[5,6])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-34-ffa12f982770> in <module>
      1 import numpy as np
      2 np.dot(2,[5,6])
----> 3 np.matmul(2,[5,6])

ValueError: matmul: Input operand 0 does not have enough dimensions (has 0, gufunc c
ore with signature (n?,k),(k,m?)->(n?,m?) requires 1)
```

QUESTION 2: WHAT'S THE DIFFERENCE BETWEEN NP.DOT() AND NDARRAY.DOT() ?

`np.dot()` and `ndarray.dot()` are very similar, and effectively perform the same operations.

The difference is that `np.dot()` is a Python *function* and `ndarray.dot()` is a *Numpy array method*.

So they effectively do the same thing, but you call them in a slightly different way.

Let's say we have two 2-dimensional arrays.

In [3]:
```
import numpy as np
A_array_2d = np.arange(start = 3, stop = 9).reshape((2,3))
B_array_2d = np.arange(start = 10, stop = 16).reshape((3,2))
print(A_array_2d)
print(B_array_2d)
```

```
[[3 4 5]
 [6 7 8]]
[[10 11]
 [12 13]
 [14 15]]
```

We can call the `np.dot()` function as follows:

```
np.dot(A_array_2d, B_array_2d)
```

But we use so-called "dot syntax" to call the `.dot()` method:

```
A_array_2d.dot(B_array_2d)
```

The output is the same, but the syntax is slightly different.

In [4]:
```
np.dot(A_array_2d, B_array_2d)
```

Out[4]:
```
array([[148, 160],
       [256, 277]])
```

In [5]:
```
A_array_2d.dot(B_array_2d)
```

Out[5]:
```
array([[148, 160],
       [256, 277]])
```

determinant

Determinant is a very useful value in linear algebra. It calculated from the diagonal elements of a square matrix. For a 2x2 matrix, it is simply the subtraction of the product of the top left and bottom right element from the product of other two.

In other words, for a matrix [[a,b], [c,d]], the determinant is computed as 'ad-bc'. The larger square matrices are considered to be a combination of 2x2 matrices.

The **numpy.linalg.det()** function calculates the determinant of the input matrix.

In [28]:
```
import numpy as np
a = np.array([[1,2], [3,4]])
print(a)
det = np.linalg.det(a)
print(det)
```

```
[[1 2]
 [3 4]]
-2.0000000000000004
```

Slicing arrays

In [ ]:
```
```

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

Example

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

please note that in slicing 'end' index is exclusive

In [15]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

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

In [7]:
```
a = np.random.random((4,5))
print(a)
```

```
[[0.82736324 0.75903456 0.99446762 0.2779022  0.84271935]
 [0.47983959 0.44274919 0.34087435 0.35063902 0.78228237]
 [0.11754346 0.92758593 0.53224428 0.94153447 0.13046144]
 [0.70896064 0.6650253  0.20305246 0.6688146  0.2399899 ]]
```

```
a = np.random.random((4,5))

a[2, :]
# third row, all columns
a[1:3]
# 2nd, 3rd row, all columns
a[:, 2:4]
# all rows, columns 3 and 4
```

In [10]:
```
a = np.random.random((4,5))
print(a)
a[2, :]
# Third row (index 2) all columns
```

```
[[0.46507549 0.21085376 0.09112913 0.31171613 0.62426762]
 [0.13893839 0.26031315 0.54402119 0.32322626 0.51787708]
 [0.48858137 0.74832798 0.1839837  0.8894496  0.22671746]
 [0.87100714 0.05137552 0.92225222 0.55669954 0.34320581]]
```

Out[10]:
```
array([0.48858137, 0.74832798, 0.1839837 , 0.8894496 , 0.22671746])
```

In [11]:
```
a[1:3]
```

Out[11]:
```
array([[0.13893839, 0.26031315, 0.54402119, 0.32322626, 0.51787708],
       [0.48858137, 0.74832798, 0.1839837 , 0.8894496 , 0.22671746]])
```

In [13]:
```
a[:, 2:4]
# all rows, columns 3  and 4
```

Out[13]:
```
array([[0.09112913, 0.31171613],
       [0.54402119, 0.32322626],
       [0.1839837 , 0.8894496 ],
       [0.92225222, 0.55669954]])
```

Example

Slice elements from index 4 to the end of the array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

In [16]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

```
[5 6 7]
```

Example

Slice elements from the beginning to index 4 (not included):

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

In [17]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

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

Negative Slicing

Use the minus operator to refer to an index from the end:

Example

Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

In [18]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

```
[5 6]
```

STEP

Use the `step` value to determine the step of the slicing:

Example

Return every other element from index 1 to index 5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

In [19]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

```
[2 4]
```

Example

Return every other element from the entire array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

In [20]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

```
[1 3 5 7]
```

In [21]:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

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

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

Slicing 2-D Arrays

Example

From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

Try it Yourself »

Note: Remember that *second element* has index 1.

In [22]:
```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

```
[7 8 9]
```

Example

From both elements, return index 2:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

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

In [23]:
```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

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

```
[3 8]
```

In [25]:

```
# only elt 3 extracted from first dimension 1,2,3,4,5
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:1, 2])
```

```
[3]
```

Example

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

Try it Yourself »

In [26]:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

In [ ]: