Lec 1,2: Numpy

- Vectorization
- Boardcast
- Identify Matrix
- Indexing and Slicing
- Fancy Array
- Reduction Operation

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

Numpy (Review)

Main object type is `np.array`

Many ways to create it,

One way is to convert a python list

In [2]:
```
python_list = [1, 2, 3]
arr = np.array(python_list)
arr
```

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

In [3]:
```
arr**2
```

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

In [4]:
```
python_list**2 #__pow__
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 python_list**2 #__pow__

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Many times a list comprehension is used to create a list and then converted to a array

In [5]:
```
python_list_pow = [i**2 for i in python_list]  # list comprehension
python_list_pow
```

Out[5]:
```
[1, 4, 9]
```

Exercise (Pre-Lec)

Create a numpy array that contain intergers i such that 0<i<100 and 2^i has the last digit 6

In [6]:
```
l = [i for i in range(1, 100) if 2**i // 10 == 6]
np.array(l)
```

Out[6]:
```
array([6])
```

Create a 2D numpy array A (5,10) such that $A_{ij} = i \times j$

In [7]:
```
# Init an empty array and assign values
A = np.zeros(shape=(5, 10))
for i in range(5):
    for j in range(10):
        A[i, j] = i * j
A
```

Out[7]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.],
       [ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27.],
       [ 0.,  4.,  8., 12., 16., 20., 24., 28., 32., 36.]])
```

In [8]:
```
# Or Init an nested list and transform it to an array
l = [[i * j for j in range(10)] for i in range(5)]
np.array(l)
```

Out[8]:
```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
       [ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27],
       [ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36]])
```

Another way to create a numpy array is with initializing functions

- np.zeros
- np.ones
- np.arange

These functions along with `reshape` can be used to create initial matrix without any for loops

In [9]:
```
np.zeros(shape = (10, 10))
```

Out[9]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [10]:
```
np.ones((10, 10)) * 2
```

Out[10]:
```
array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.]])
```

In [11]:
```
np.arange(2, 10, 2)  # equivalent to range(2,10,2)
```

Out[11]:
```
array([2, 4, 6, 8])
```

Exercise

Create an array of first 10 powers of 2

In [18]:
```
np.array([2**i for i in range(10)])
```

Out[18]:
```
array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512])
```

1D vs 2D array

In [19]:
```
array1D = np.arange(10) * np.arange(10)
array1D = array1D.reshape(1,-1)
print(array1D.shape)
array1D
```

```
(1, 10)
```

Out[19]:
```
array([[ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81]])
```

In [20]:
```
array2D = np.arange(10).reshape(10, 1)
print(array2D.shape)
array2D
```

```
(10, 1)
```

Out[20]:
```
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8],
       [9]])
```

In [21]:
```
array2D + array1D #boardcasting
```

Out[21]:
```
array([[ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81],
       [ 1,  2,  5, 10, 17, 26, 37, 50, 65, 82],
       [ 2,  3,  6, 11, 18, 27, 38, 51, 66, 83],
       [ 3,  4,  7, 12, 19, 28, 39, 52, 67, 84],
       [ 4,  5,  8, 13, 20, 29, 40, 53, 68, 85],
       [ 5,  6,  9, 14, 21, 30, 41, 54, 69, 86],
       [ 6,  7, 10, 15, 22, 31, 42, 55, 70, 87],
       [ 7,  8, 11, 16, 23, 32, 43, 56, 71, 88],
       [ 8,  9, 12, 17, 24, 33, 44, 57, 72, 89],
       [ 9, 10, 13, 18, 25, 34, 45, 58, 73, 90]])
```

Distinction between numpy 1D arrays and numpy 2D arrays

This tends to cause a lot of confusion for new numpy users. Follow the below examples carefully to understand the distinction.

In [22]:
```
Z = np.zeros(shape=10)
print(Z)
Z.shape
```

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

Out[22]:
```
(10,)
```

In [23]:
```
# Create 2D array by reshape
Z = np.zeros(10).reshape(10, 1)
print(Z)
Z.shape
```

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

Out[23]:
```
(10, 1)
```

In [24]:
```
Z.squeeze()  # remove axis with length = 1
```

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

In [25]:
```
# squeeze Remove axes of length one
Z = np.zeros(6).reshape(1, 1, 2, 3)
print(Z)
print(Z.shape, "\n")

Z_squeeze = Z.squeeze()
print(Z_squeeze)
print(Z_squeeze.shape)
```

```
[[[[0. 0. 0.]
   [0. 0. 0.]]]]
(1, 1, 2, 3)

[[0. 0. 0.]
 [0. 0. 0.]]
(2, 3)
```

In [26]:
```
# Matrix Multiplication
Mat = np.random.randn(10, 10)
Mat.shape
```

Out[26]:
```
(10, 10)
```

In [27]:
```
Z = np.arange(10).reshape(10, 1)
print(Z)
print(Z.shape)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
(10, 1)
```

In [28]:
```
# (N, M) @ (M, K) = (N, K)
Mat @ Z
```

Out[28]:
```
array([[  6.16902411],
       [ 14.75584471],
       [  0.45801183],
       [-23.08428445],
       [ -4.79697344],
       [ 19.01273892],
       [ 20.00785746],
       [ -6.88310327],
       [  4.85980299],
       [  9.62414849]])
```

In [29]:
```
Z = np.arange(10).reshape(1, 10)

print(Mat.shape)
print(Z.shape)

# (N, M) @ (M, K) = (N, K)
Mat @ Z  # (10, 10) @ (1, 10) NOT WORKING
```

```
(10, 10)
(1, 10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[29], line 7
      4 print(Z.shape)
      6 # (N, M) @ (M, K) = (N, K)
----> 7 Mat @ Z  # (10, 10) @ (1, 10) NOT WORKING

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with guf
unc signature (n?,k),(k,m?)->(n?,m?) (size 1 is different from 10)
```

In [30]:
```
(Z @ Mat).shape
```

Out[30]:
```
(1, 10)
```

In [31]:
```
# array variable is also a pointer
x = np.zeros((5, 5))
y = x.copy()
x[1, 1] = 2

y
```

Out[31]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Array Broadcasting

Normally you only do arithmetic operations between arrays of the same dimension

The smaller array of at least 1 dimension of size 1 is “broadcast” across the larger array so that they have compatible shapes by dimension.

In [32]:
```
a = np.arange(3).reshape(1, 3)
b = np.arange(6).reshape(6, 1)
c = np.ones((3, 3))
d = np.zeros((6, 3))
print(a, "\n")
print(b, "\n")
print(c, "\n")
print(d, "\n")
```

```
[[0 1 2]]

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

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

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

In [33]:
```
# c + 2
# c + c
# d + a
# c + d
b + d
```

Out[33]:
```
array([[0., 0., 0.],
       [1., 1., 1.],
       [2., 2., 2.],
       [3., 3., 3.],
       [4., 4., 4.],
       [5., 5., 5.]])
```

In [34]:
```
a + b
```

Out[34]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6],
       [5, 6, 7]])
```

In [ ]:
```
X = np.arange(4).reshape(-1, 1) * 10
Y = np.arange(3).reshape(1, -1)
print(X.shape)
```

```
print(Y.shape, "\n")
print(X, "\n")
print(Y, "\n")
X + Y
```

```
(4, 1)
(1, 3)

[[ 0]
 [10]
 [20]
 [30]]

[[0 1 2]]
```

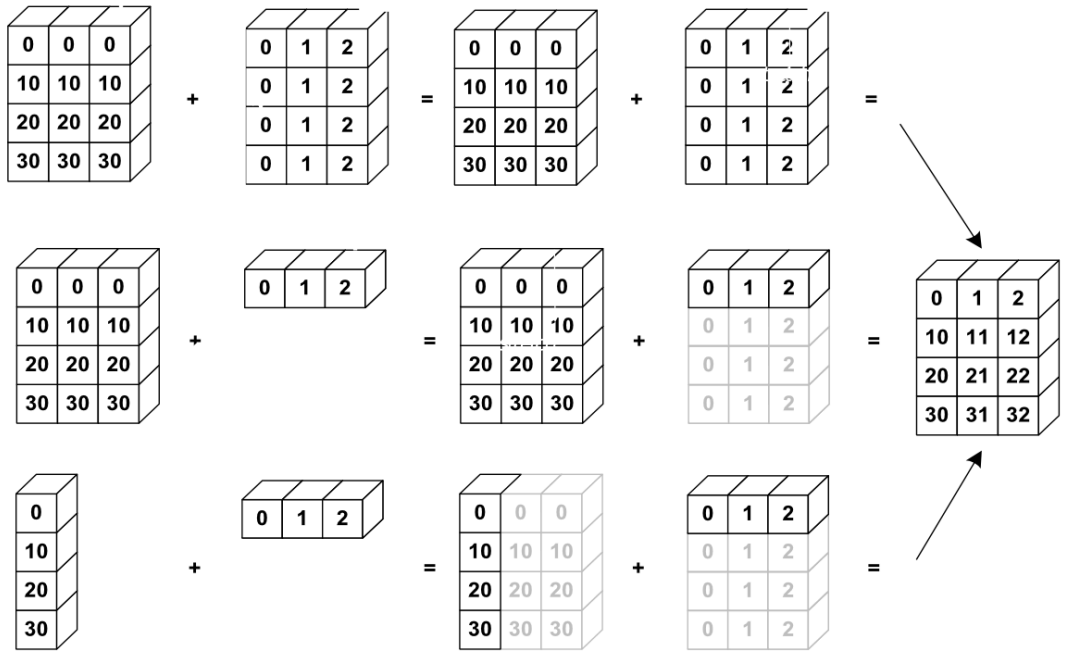
Out[ ]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

it tries to duplicates every x's col and y's row (dim size = 1) to match the other arrays

1. Make the two arrays have the same number of dimensions.

- If the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.

2. If there is a dimension whose size is not 1 in either of the two arrays, it cannot be broadcasted, and an error is raised.



Exercise

create a 2D numpy array A (shape = (5,10)) such that $A_{ij} = i \times j$, but without using list comprehensions. Use broadcasting instead

In [41]:
```
# hint: check how it looks for
#       np.arange(5).reshape(-1,1)
#       np.arange(10).reshape(1,-1)
```

In [42]:
```
# Solution:
a=np.arange(5).reshape(-1,1)
b=np.arange(10).reshape(1,-1)
a*b
```

Out[42]:
```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
       [ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27],
       [ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36]])
```

Use array broadcasting to create a (10,10) numpy array with values

$$A_{ij} = 2^i + j$$

In [ ]:
```
# Hint: Check the values of
#       2**(np.arange(10).reshape(-1,1))
#       np.arange(10).reshape(1,-1)
```

In [44]:
```
# Solutions
c=2**(np.arange(10).reshape(-1,1))
d=np.arange(10).reshape(1,-1)
c*d
```

Out[44]:
```
array([[   0,    1,    2,    3,    4,    5,    6,    7,    8,    9],
       [   0,    2,    4,    6,    8,   10,   12,   14,   16,   18],
       [   0,    4,    8,   12,   16,   20,   24,   28,   32,   36],
       [   0,    8,   16,   24,   32,   40,   48,   56,   64,   72],
       [   0,   16,   32,   48,   64,   80,   96,  112,  128,  144],
       [   0,   32,   64,   96,  128,  160,  192,  224,  256,  288],
       [   0,   64,  128,  192,  256,  320,  384,  448,  512,  576],
       [   0,  128,  256,  384,  512,  640,  768,  896, 1024, 1152],
       [   0,  256,  512,  768, 1024, 1280, 1536, 1792, 2048, 2304],
       [   0,  512, 1024, 1536, 2048, 2560, 3072, 3584, 4096, 4608]])
```

Matrix creation

There are some functions to create standard matrices

In [45]:
```
np.eye(5)
```

Out[45]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [46]:
```
np.arange(10)
```

Out[46]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [47]:
```
# np.diag: Extract a diagonal or construct a diagonal array.
M = np.diag(np.arange(10))  # .reshape(5,20)
M
```

Out[47]:
```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 3, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 4, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 5, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 6, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 7, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 8, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 9]])
```

In [48]:
```
np.diag(M)
```

Out[48]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [49]:
```
# Transpose of the matrix
A = np.arange(15).reshape(5, 3)
print(A.shape)
A
```

```
(5, 3)
```

Out[49]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [50]:
```
A.T
```

Out[50]:
```
array([[ 0,  3,  6,  9, 12],
       [ 1,  4,  7, 10, 13],
       [ 2,  5,  8, 11, 14]])
```

In [51]:
```
A.transpose()
```

Out[51]:
```
array([[ 0,  3,  6,  9, 12],
       [ 1,  4,  7, 10, 13],
       [ 2,  5,  8, 11, 14]])
```

random seed

In [52]:
```
np.random.rand(5, 5)
```

Out[52]:
```
array([[0.29546074, 0.02840564, 0.21385255, 0.33095107, 0.40069313],
       [0.51055577, 0.17778541, 0.94320653, 0.42215463, 0.61007559],
       [0.75276374, 0.93780342, 0.14248202, 0.30166784, 0.49830364],
       [0.84982435, 0.16620083, 0.14147031, 0.7805677 , 0.00463861],
       [0.41315036, 0.50815103, 0.26585049, 0.22805464, 0.41626867]])
```

In [53]:
```
np.random.seed(0)  # control the random state
print(np.random.rand(5, 5))
print(np.random.rand(5, 5))
print(np.random.rand(5, 5))
```

```
[[0.5488135  0.71518937 0.60276338 0.54488318 0.4236548 ]
 [0.64589411 0.43758721 0.891773   0.96366276 0.38344152]
 [0.79172504 0.52889492 0.56804456 0.92559664 0.07103606]
 [0.0871293  0.0202184  0.83261985 0.77815675 0.87001215]
 [0.97861834 0.79915856 0.46147936 0.78052918 0.11827443]]
[[0.63992102 0.14335329 0.94466892 0.52184832 0.41466194]
 [0.26455561 0.77423369 0.45615033 0.56843395 0.0187898 ]
 [0.6176355  0.61209572 0.616934   0.94374808 0.6818203 ]
 [0.3595079  0.43703195 0.6976312  0.06022547 0.66676672]
 [0.67063787 0.21038256 0.1289263  0.31542835 0.36371077]]
[[0.57019677 0.43860151 0.98837384 0.10204481 0.20887676]
 [0.16130952 0.65310833 0.2532916  0.46631077 0.24442559]
 [0.15896958 0.11037514 0.65632959 0.13818295 0.19658236]
 [0.36872517 0.82099323 0.09710128 0.83794491 0.09609841]
 [0.97645947 0.4686512  0.97676109 0.60484552 0.73926358]]
```

Exercise

Create this matrix

```
array([[5., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 4., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 3., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 2., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 2., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 3., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 4., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 5.]])
```

In [63]:
```
# Solution
e = np.ones((11,11))

np.fill_diagonal(e,[abs(i) for i in range(-5,5)])
e
```

Out[63]:
```
array([[5., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 4., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 3., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 2., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 2., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 3., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 4., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 5.]])
```

Array Indexing and Slicing

In [64]:
```
import numpy as np

arr = np.arange(10)
arr
```

Out[64]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [65]:
```
arr[5]
```

Out[65]:
```
np.int64(5)
```

In [66]:
```
arr[-3]
```

Out[66]:
```
np.int64(7)
```

In [67]:
```
arr[3:7]
```

Out[67]:
```
array([3, 4, 5, 6])
```

In [68]:
```
arr[2:]
```

Out[68]:
```
array([2, 3, 4, 5, 6, 7, 8, 9])
```

In [69]:
```
arr[0:-3]
```

Out[69]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

In [70]:
```
arr[0:6:2]  # similar as range(0,6,2)
```

Out[70]:
```
array([0, 2, 4])
```

In [71]:
```
arr[5:0:-2]
```

Out[71]:
```
array([5, 3, 1])
```

In [72]:
```
arr[::-1]
```

Out[72]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

In [73]:
```
arr[:]
```

Out[73]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [74]:
```
arr
```

Out[74]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [75]:
```
a = 10 * np.arange(6).reshape(-1, 1) + np.arange(6)
print(a)
# a[4:, 4:]
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
```

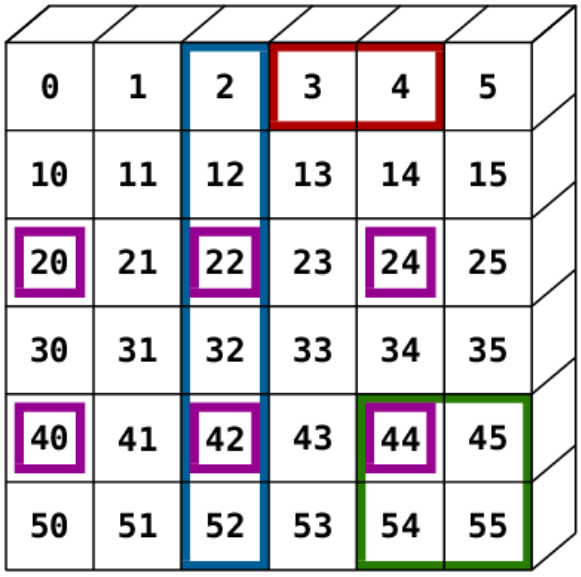
Can use all the above slicing methods for each dimension of a multidemnsional array

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



try it yourself

Exercise

Create the following matrix

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

In [87]:
```
# Solution
f = np.ones((10,10))
f[2:8,2:8] = 0

f
# for i in range(len(f)):
#     k = f[i]
#     for j in range(len(k)):
#         if j<=1 or j>=8:
#             k[j] = 1
#         elif i <= 1 or i >= 8:
```

```
#             k[j] = 1
```

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

Create the following matrix

```
array([[-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.],
       [-1.,  0.,  1.,  2.,  3.,  4., -1., -1., -1., -1.],
       [-1.,  5.,  6.,  7.,  8.,  9., -1., -1., -1., -1.],
       [-1., 10., 11., 12., 13., 14., -1., -1., -1., -1.],
       [-1., 15., 16., 17., 18., 19., -1., -1., -1., -1.],
       [-1., 20., 21., 22., 23., 24., -1., -1., -1., -1.],
       [-1., 25., 26., 27., 28., 29., -1., -1., -1., -1.],
       [-1., 30., 31., 32., 33., 34., -1., -1., -1., -1.],
       [-1., 35., 36., 37., 38., 39., -1., -1., -1., -1.],
       [-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.]])
```

In [99]:
```
# Solution
g = np.ones((10,10))
g = g*-1
h = np.arange(40).reshape(8,5)
g[1:9,1:6] = h
g
```

Out[99]:
```
array([[-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.],
       [-1.,  0.,  1.,  2.,  3.,  4., -1., -1., -1., -1.],
       [-1.,  5.,  6.,  7.,  8.,  9., -1., -1., -1., -1.],
       [-1., 10., 11., 12., 13., 14., -1., -1., -1., -1.],
       [-1., 15., 16., 17., 18., 19., -1., -1., -1., -1.],
       [-1., 20., 21., 22., 23., 24., -1., -1., -1., -1.],
       [-1., 25., 26., 27., 28., 29., -1., -1., -1., -1.],
       [-1., 30., 31., 32., 33., 34., -1., -1., -1., -1.],
       [-1., 35., 36., 37., 38., 39., -1., -1., -1., -1.],
       [-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.]])
```

Fancy Array Indexing

We can use numpy arrays as an index for other numpy arrays

In [100…
```
arr = np.arange(10)
arr
```

Out[100…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [101…
```
# use array/list/tuple as indexs
idx = np.array([2, 7, -1])
idx
```

Out[101…
```
array([ 2,  7, -1])
```

In [102…
```
print(arr[idx])
arr[idx] = -1
arr
```

```
[2 7 9]
```

Out[102…
```
array([ 0,  1, -1,  3,  4,  5,  6, -1,  8, -1])
```

In [103…
```
# use bool array
arr < 0
```

Out[103…
```
array([False, False,  True, False, False, False, False,  True, False,
        True])
```

In [104…
```
arr[arr < 0] = 100
arr
```

Out[104…
```
array([  0,   1, 100,   3,   4,   5,   6, 100,   8, 100])
```

For multidimensional array, array indexing works different from slicing

In [105…
```
X = np.zeros((6, 6))
X[2:5, 0:3] = 1
X
```

Out[105…
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

In [106…
```
np.arange(2, 5), np.arange(0, 3)
```

Out[106…
```
(array([2, 3, 4]), array([0, 1, 2]))
```

In [107…
```
X = np.zeros((6, 6))
X[np.arange(2, 5), np.arange(0, 3)] = 1
X
```

Out[107…
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

In [108…
```
# Here is our array, what should we return?
a = 10 * np.arange(6).reshape(-1, 1) + np.arange(6)
a
```

Out[108…
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

In [109…
```
a[(1, 2, 3, 4, 5), (0, 1, 2, 3, 4)]
```

Out[109…
```
array([10, 21, 32, 43, 54])
```

In [110…
```
a[3:, [0, 2, 5]]
```

Out[110…
```
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])
```

In [111…
```
mask = np.array([1, 0, 1, 0, 0, 1], dtype=bool)
print(mask)
a[mask, 2]
```

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

Out[111…
```
array([ 2, 22, 52])
```

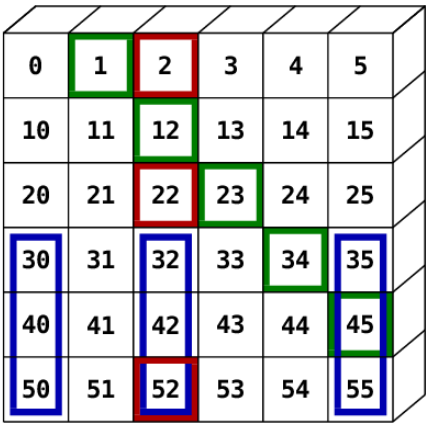
In [112…
```
mask = np.array([1,0,1,0,0,1])
a[mask,2]
```

Out[112…
```
array([12,  2, 12,  2,  2, 12])
```

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



Exercise

Create the following matrix

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

```
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

In [121…
```
# Solution
i = np.zeros((10,10))
i[(5,4,3,6,2,7,1,8,9,0),(1,1,3,3,4,4,7,7,9,9)] = 1
i
```

Out[121…
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

Exercise

Write a function to compute the trace) of a square numpy array using fancy array indexing. Compare your implementation to numpy's built-in function `np.trace`.

In [128…
```
# Solution
l = np.random.rand(3,3)
l[tuple([i for i in range(len(l))]),tuple([i for i in range(len(l))])].sum()
```

Out[128…
```
np.float64(1.7743613320120613)
```

We can use `np.where`, to get indices of the `True` values in a boolean array

In [129…
```
Y = np.arange(25).reshape(5, 5)
print(Y)
Y > 14
print(np.where(Y > 14))
# Y[np.where(Y>14)]
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
(array([3, 3, 3, 3, 3, 4, 4, 4, 4, 4]), array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4]))
```

Reduction operations

Many reduction functions are available

- np.sum, np.prod
- np.min, np.max
- np.any, np.all

Partial reductions

- np.cumsum, np.cumprod

In [130…
```
X = np.arange(50).reshape(10,5)
X
```

Out[130…
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44],
       [45, 46, 47, 48, 49]])
```

In [131…
```
np.sum(X), np.prod(X)
# class.method(self)
```

Out[131…
```
(np.int64(1225), np.int64(0))
```

The way to understand the “axis” of numpy sum is it collapses the specified axis. So when it collapses the axis 0 (row), it becomes just one row and column-wise sum.

In [132…
```
# sum of the rows
np.sum(X, axis=1)
```

Out[132…
```
array([ 10,  35,  60,  85, 110, 135, 160, 185, 210, 235])
```

In [133…
```
X.sum(axis=1)
```

Out[133…
```
array([ 10,  35,  60,  85, 110, 135, 160, 185, 210, 235])
```

In [134…
```
np.min(X), np.max(X)
```

Out[134…
```
(np.int64(0), np.int64(49))
```

In [135…
```
X
```

Out[135…
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44],
       [45, 46, 47, 48, 49]])
```

In [136…
```
np.min(X, axis=0)
```

Out[136…
```
array([0, 1, 2, 3, 4])
```

In [137…
```
Y = X < 12
Y
```

Out[137…
```
array([[ True,  True,  True,  True,  True],
       [ True,  True,  True,  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, False],
       [False, False, False, False, False],
       [False, False, False, False, False]])
```

In [138…
```
np.any(Y, axis=1)
```

Out[138…
```
array([ True,  True,  True, False, False, False, False, False, False,
       False])
```

In [139…
```
np.all(Y, axis=1)
```

Out[139…
```
array([ True,  True, False, False, False, False, False, False, False,
       False])
```

All the above functions can be called on the array object directly

In [140…
```
# instance.method(args) = class.method(instance, args)
X.max(axis=0)
```

Out[140…
```
array([45, 46, 47, 48, 49])
```

In [141…
```
# np.cumsum
Y = np.arange(10)
print(Y)
np.cumsum(Y)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Out[141…
```
array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
```

In [142…
```
X = np.arange(16).reshape(4, 4)
print(X)
np.cumsum(X, axis=1)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

Out[142…
```
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38],
       [12, 25, 39, 54]])
```

Cumulative operations don't change the shape of the array

Exercise

- Find the column with maximum column sum
- For which rows of the matrix, the sum of the first three elements of the row is greater than the sum of the last two elements of the row

In [143…
```
import numpy as np
```

In [160…
```
np.random.seed(10)
X = np.random.rand(5, 5)
# Solution
print(X)
a = np.cumsum(X,axis=0)
max_col = 0
max_val = 0
for i in range(len(a[-1])):
    if a[-1][i] > max_val:
        max_col = i
        max_val = a[-1][i]
print(f'Column with the maximum column sum is: {max_col}')

b = X[:,:3]
c = X[:,3:]
b1 = np.cumsum(b,axis=1)[:,-1:]
c1 = np.cumsum(c,axis=1)[:,-1:]
d = b1-c1
gt0 = []
for i in range(len(d)):
    if d[i][0] > 0:
        gt0.append(i)
print(f'Rows where the sum of the first two columns are greater than the last two c
```

```
[[0.77132064 0.02075195 0.63364823 0.74880388 0.49850701]
 [0.22479665 0.19806286 0.76053071 0.16911084 0.08833981]
 [0.68535982 0.95339335 0.00394827 0.51219226 0.81262096]
 [0.61252607 0.72175532 0.29187607 0.91777412 0.71457578]
 [0.54254437 0.14217005 0.37334076 0.67413362 0.44183317]]
Column with the maximum column sum is: 3
Rows where the sum of the first two columns are greater than the last two columns ar
e: [0, 1, 2]
```