VNUHCM - University of Science
fit@hcmus

Fundamentals of Programming for Artificial Intelligence

Session 09
Numpy

Instructors:
Dr. Lê Thanh Tùng
Dr. Nguyễn Tiến Huy

Numpy

- NumPy is a Python library used for working with arrays
- One of the most important foundational packages for numerical computing in Python.
- Most computational packages providing scientific functionality use NumPy’s array objects as the lingua franca for data exchange.
- Providing fast array-oriented operations and flexible broadcasting capabilities

Array

- NumPy is used to work with arrays. The array object in NumPy is called ndarray
- Create a NumPy ndarray object by using the array() function

```
import numpy as np

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

print(arr) # [1 2 3 4 5]
print(type(arr)) # <class 'numpy.ndarray'>
```

Array

```python
import numpy as np
# from list
print(np.array([1, 2, 3])) # [1 2 3]
# from tuple
print(np.array((1, 2, 3))) # [1 2 3]
print(np.array((1, 2, 3))[2]) # 3
# from set
print(np.array({1, 2, 3})) # {1, 2, 3}
print(np.array({1, 2, 3})[2])
# IndexError: too many indices for array:
# array is 0-dimensional,
# but 1 were indexed
```

Dimensions in Arrays

- A dimension in arrays is one level of array depth (nested arrays)
- 0-D array or Scalars, are the elements in an array
- 1-D array has 0-D arrays as its elements, called uni-dimensional
- 2-D array has 1-D arrays as its elements, called 2-d matrix
- 3-D array has 2-D matrices as its elements

Dimensions in Arrays

```python
import numpy as np


arr2d_1 = np.array([[1, 2, 3], [4, 5, 6]])
arr3d_2 = np.array([1, 2, 3, 4], ndmin=3)
print(arr2d_1) # [[1 2 3] [4 5 6]]
print(arr3d_2) # [[[1 2 3 4]]]
print(arr2d_1.ndim, arr3d_2.ndim) # 2 3
```

Shape & Type of ndarray

- Attribute: ndarray.shape – Tuple of array dimensions

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

- Attribute: ndarray.dtype – Data-type of the array's elements

```
a = np.array([0.5, 1, 2])
print(a.dtype) # float64
```

Create Array

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

```
# Example:
print(np.zeros(5)) # [0. 0. 0. 0. 0.]
print(np.ones((2, 3))) # [[1. 1. 1.]
                       #  [1. 1. 1.]]
```

dtype

Type	Type code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

Type & Type cast

```
[3] #type cast
    arr = np.array([1.5,2,3,4.5,5], dtype=np.float64) #dtype = float64
    int_arr = arr.astype(np.int64) #dtype = int64
    int_arr
```

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

numpy.arange()

- arange([start,] stop[, step,][, dtype])

Parameters :

```
start : [optional] start of interval range. By default start = 0
stop  : end of interval range
step  : [optional] step size of interval. By default step size = 1,
For any output out, this is the distance between two adjacent values, out[i+1] - out[i].
dtype : type of output array
```

Return:

```
Array of evenly spaced values.
Length of array being generated  = Ceil((Stop - Start) / Step)
```

numpy.arange()

```python
a1 = np.arange(10)
print(a1, "dim = %d" % a1.ndim)
# [0 1 2 3 4 5 6 7 8 9] dim = 1
a2 = np.arange(2, 10, 3)
print(a2, "dim = %d" % a2.ndim)
# [2 5 8] dim = 1
```

numpy.random.randn

- random.randn(d0, d1, ..., dn): Return a sample (or samples) from the "standard normal" distribution.

If positive int_like arguments are provided, `randn` generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters: **d0, d1, ..., dn : *int, optional***

The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns: **Z : *ndarray or float***

A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

numpy.random.randn

```
a1 = np.random.randn()
print(type(a1), a1) # <class 'float'> 1.3338705849916304
a2 = np.random.randn(1)
print(type(a2), a2) # <class 'numpy.ndarray'> [1.80212745]
a3 = np.random.randn(2, 3)
print(type(a3), a3.shape) # <class 'numpy.ndarray'> (2, 3)
print(a3) # [[-0.2903103   0.03688071 -0.1302092 ]
          #  [-1.27259517  0.69088078  1.98388177]]
```

Numerical operations on arrays

- Elementwise operations:
  - With scalars:

```
a = np.array([1, 2, 3, 4])
print(a + 1) # [2 3 4 5]
print(3 - a) # [ 2  1  0 -1]
print(a*2) # [2 4 6 8]
print(2**a) # [ 2  4  8 16]
print(a//2) # [0 1 1 2]
```

Numerical operations on arrays

- Elementwise operations: With array (same dimension):

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 1, 7, 4])
print(a + b) # [ 6  3 10  8]
print(a - b) # [-4  1 -4  0]
print(a * b) # [ 5  2 21 16]
print(a % b) # [1 0 3 0]
print(2**a - b) # [-3  3  1 12]
```

Numerical operations on arrays

- Matrix multiplication

```
a = np.array([1, 2, 3, 4])
b = np.array([1, 2, 2, 1])
print(a * b) # element-wise multiplication
# [1 4 6 4]
print(a.dot(b)) # matrix multiplication
# 15
print(a.shape, b.shape) # (4,) (4,)
```

Numerical operations on arrays

Comparisons:

```python
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
print(a == b) # [False  True False  True]
print(a > b) # [False False  True False]
```

Array-wise Comparisons:

```python
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
print(np.array_equal(a, b)) # False
print(np.array_equal(a, a)) # True
```

Numerical operations on arrays

- Logical operations

```
a = np.array([1, 0, 3, 0])
b = np.array([4, 0, 2, 4])
print(np.logical_or(a, b))
# [ True False True True]
print(np.logical_and(a, b))
# [ True False True False]
```

List vs numpy array in Math operation

```
[ ] #Multidimensional array object
    my_np_data = np.random.randn(2,3)
    print(my_np_data)

    [[-0.08221029  0.36287271  0.44168154]
     [ 0.639073    0.52044355 -1.09013561]]
```

```
[ ] my_list = [[-0.14660949,  0.10938535 , 0.73807359],
     [ 0.23479429, -0.18461008 ,-1.22907498]]
```

```
[ ] my_np_data * 10

    array([[ -0.82210292,   3.62872709,   4.41681535],
           [  6.39073001,   5.20443547, -10.90135605]])
```

```
[ ] my_list * 2

    [[-0.14660949, 0.10938535, 0.73807359],
     [0.23479429, -0.18461008, -1.22907498],
     [-0.14660949, 0.10938535, 0.73807359],
     [0.23479429, -0.18461008, -1.22907498]]
```

Indexing & Slicing

```
# One dimensions
arr = np.arange(10) # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
arr[5] # 5
arr[5:8] # array([5, 6, 7])
arr[5:8] = 12 # array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])

# Two dimensions
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2] #array([7, 8, 9])
arr2d[0][2] # 3
arr2d[0,2] # 3
arr2d[:2,1:] # array([[2, 3],[5, 6]]
```

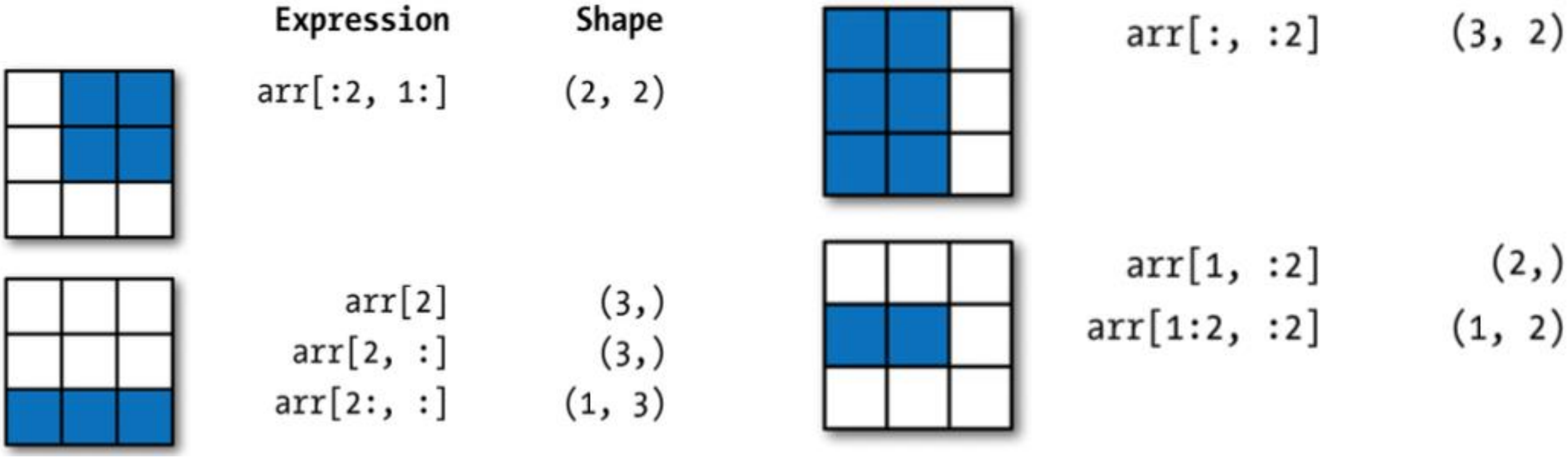
Indexing & Slicing

```
# Three dimensions
a = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(a.shape) # (2, 2, 3)
temp = a[0] # [[1 2 3]
            #  [4 5 6]]
temp[:] = 0
# Explain the result if using the code line  "temp = 0"
print(a) # [[[ 0  0  0]
         #   [ 0  0  0]]
         #  [[ 7  8  9]
         #   [10 11 12]]]
```

Indexing & Slicing

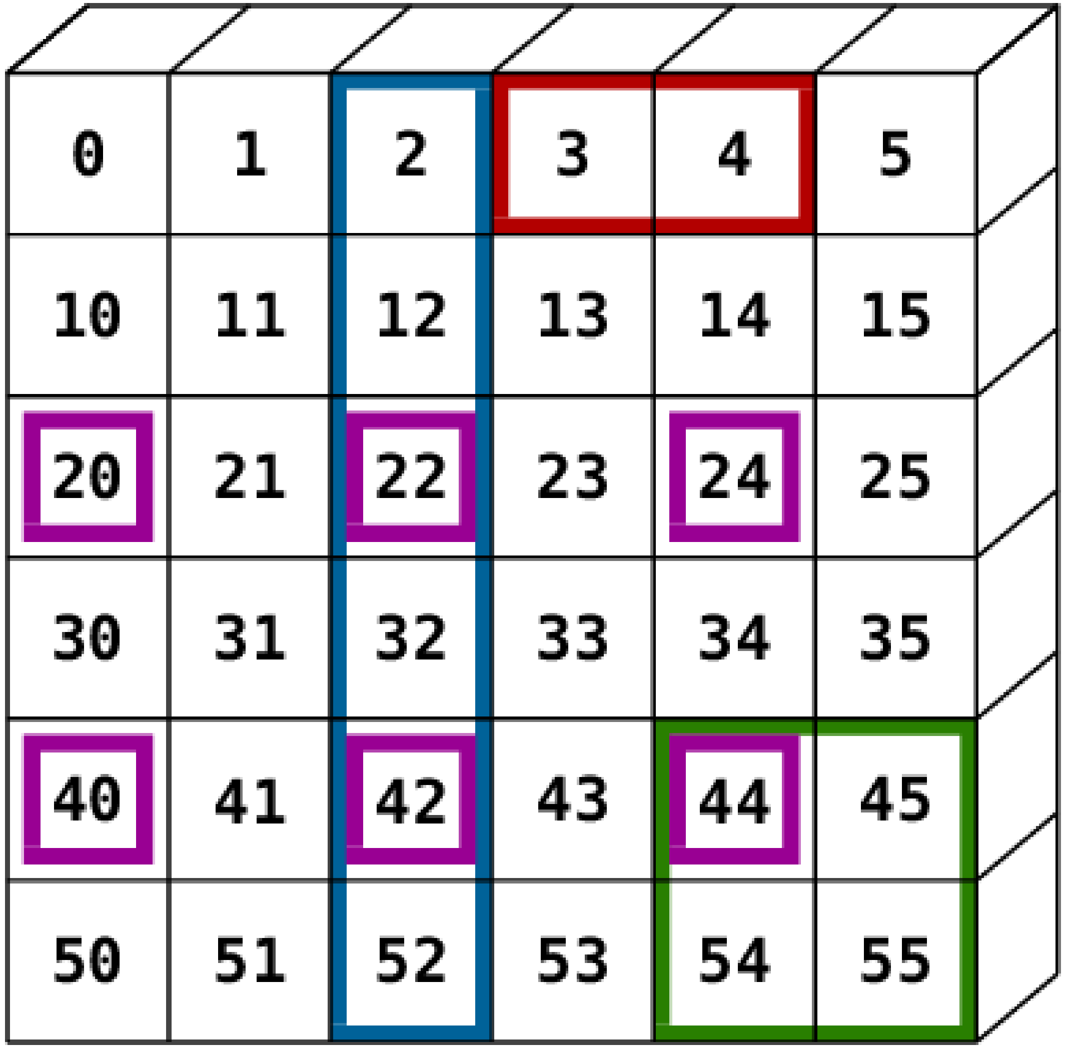
```
temp  = 0
# Explain the result if using the code line  "temp = 0"
print(a) # [[[ 1  2  3]
         #   [ 4  5  6]]
         #  [[ 7  8  9]
         #   [10 11 12]]]
```

Indexing & Slicing



Indexing & Slicing

```
>>> 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]])
```



Copies & Views in ndarray

```python
# View of ndarray via assignment operator
a = np.array([[1, 2, 3], [4, 5, 6]])
b = a[0] # creates a view
print(b) # [1 2 3]
a[0][0] = 0
print(b) # [0 2 3]
```

Copies & Views in ndarray

```
# Copy of ndarray
a = np.array([[1, 2, 3], [4, 5, 6]])
b = a[0].copy() # creates a copy
print(b) # [1 2 3]
a[0][0] = 0
print(a[0]) # [0 2 3]
print(b) # [1 2 3]
```

Boolean indexing

- Numpy arrays support a feature called **conditional selection** to generate a new array of boolean values that state whether each element within the array satisfies a particular if statement.

```
arr = np.array([0.69, 0.94, 0.66, 0.73, 0.83])
index = arr > 0.7 # [False True False True  True]
print(arr[index]) # [0.94 0.73 0.83]
```

Boolean indexing

```
names = np.array(['Bob', 'Will', 'Joe', 'Bob', 'Will'])
data = np.random.randn(5, 3)
print(data)
# [[ 1.64303    -0.91901438 -0.46691434]
#  [ 0.5523564  -1.29891734 -0.76211524]
#  [ 0.3985253   0.60424701  1.68980142]
#  [ 0.82260643  1.74365049 -0.01960414]
#  [ 0.94043782 -1.19145099  0.50533676]]
print(data[names == 'Bob'])
# [[ 1.64303    -0.91901438 -0.46691434]
#  [ 0.82260643  1.74365049 -0.01960414]]
print(data[names == 'Bob', 1:])
#[[-0.91901438 -0.46691434]
# [ 1.74365049 -0.01960414]]
print(data[names == 'Bob'][1:]) # [[ 0.82260643  1.74365049 -0.01960414]]
```

Boolean indexing

```
data = np.random.randn(5, 3)
names = np.array(["Bob", "Will", "Joe", "Bob", "Will"])
print(data)
# [[-1.54658869  0.08718386  0.3361064 ]
#  [-0.76355848  0.41091788  0.36225458]
#  [-0.66877597  1.33625116  1.15840969]
#  [ 0.3875389  -0.18037251  0.34739942]
#  [ 0.21145186  0.45118169  1.20429492]]
print(data[names != "Bob"])
# [[-0.76355848  0.41091788  0.36225458]
#  [-0.66877597  1.33625116  1.15840969]
#  [ 0.21145186  0.45118169  1.20429492]]
print(~(names != "Bob")) # [ True False False  True False]
print(data[~(names != "Bob")])
# [[-1.54658869  0.08718386  0.3361064 ]
#  [ 0.3875389  -0.18037251  0.34739942]]
```

Fancy Index

- Besides using indexing & slicing, NumPy provides you with a convenient way to index an array called fancy indexing.
- Fancy indexing allows you to index a numpy array using the following:
  - Another numpy array
  - A Python list
  - A sequence of integers

Fancy Index

```
a = np.arange(1, 10)
print(a) # [1 2 3 4 5 6 7 8 9]
indices = np.array([2, 3, 4])
print(a[indices]) # [3 4 5]
```

Fancy Index

```
[ ] arr = np.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]])

    arr[[1,4,7]]
```

```
array([[ 4,  5,  6,  7],
       [16, 17, 18, 19],
       [28, 29, 30, 31]])
```

```
[ ] arr[[1,4, 7], [0, 0, 1]]
```

```
array([ 4, 16, 29])
```

```
[ ] arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Reshape

- Reshaping means changing the shape of an array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newArr = arr.reshape(4, 3)
print(newArr)
# [[ 1  2  3]
#  [ 4  5  6]
#  [ 7  8  9]
#  [10 11 12]]
```

Reshape

- Reshaping returns the view of the original array

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newArr = arr.reshape(4, 3)
print(newArr)
# [[ 1  2  3]
#  [ 4  5  6]
#  [ 7  8  9]
#  [10 11 12]]
arr[1] = 10
print(newArr)
# [[ 1 10  3]
#  [ 4  5  6]
#  [ 7  8  9]
#  [10 11 12]]
```

Reshape: Unknown Dimension

- Pass -1 as the value, and NumPy will calculate this number for you

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newArr = arr.reshape(2, 3, -1)
print(newArr)
# [[[ 1  2]
#   [ 3  4]
#   [ 5  6]]
#  [[ 7  8]
#   [ 9 10]
#   [11 12]]]
```

Exercise

- Ex1: Given an integer matrix of NxM, set negative values in the matrix to 0
- Ex2: Given an integer matrix of NxM, set negative values in even rows to 0

Exercise

```python
def setZero4Negative_Index(arr):
    shape = arr.shape
    for row in range(shape[0]):
        for col in range(shape[1]):
            if (arr[row][col] < 0):
                arr[row][col] = 0


arr1 = np.array([[1, 2, -3], [-1, -2, 3]])
setZero4Negative_Index(arr1)
print(arr1)
```

Exercise

```python
def setZero4Negative(arr):
    arr[arr < 0] = 0

arr2 = np.array([[1, 2, -3], [-1, -2, 3]])
setZero4Negative(arr2)
print(arr2)
```

Exercise

```python
def setZero4Negative_Index(arr):
    shape = arr.shape
    for row in range(shape[0]):
        for col in range(shape[1]):
            if (row % 2 == 0):
                if (arr[row][col] < 0):
                    arr[row][col] = 0


arr1 = np.array([[1, 2, -3], [-1, -2, 3], [-3, -2, 4]])
setZero4Negative_Index(arr1)
print(arr1)
```

Exercise

```
def setZero4Negative(arr):
    index = arr < 0
    for i in range(len(arr)): # range(arr.shape[0])
        if (i % 2 == 0): arr[i][index[i]] = 0

arr2 = np.array([[1, 2, -3], [-1, -2, 3], [-3, -2, 4]])
setZero4Negative(arr2)
print(arr2)
```

Exercise

```
arr3 = np.array([[1, 2, -3], [-1, -2, 3], [-3, -2, 4]])
arrEvenRow = arr3[::2]
arrEvenRow[arrEvenRow < 0] = 0
print(arr3)
```

Exercise

Ex3: Create a checkboard matrix NxN as follows:

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

Exercise

```
def createMatrix(n: int):
    arr = np.zeros((n, n), dtype = np.int64)
    mark = 0
    for row in range(n):
        for col in range(n):
            arr[row][col] = mark
            mark = not(mark)
        mark = not(mark)
    return arr
```

Exercise

```python
def createMatrixVer2(n: int):
    arr = np.zeros((n, n), dtype = np.int64)
    arr[1::2, ::2] = 1
    arr[::2, 1::2] = 1
    return arr
```

Exercise

Ex4: Create a matrix NxM with 1 in borders and 0 for inside:

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

Exercise

```python
def createMatrix(n: int, m: int):
    arr = np.zeros((n, m), dtype = np.int64)
    for row in range(n):
        for col in range(n):
            if (row == 0 or row == n-1):
                arr[row][col] = 1
            if (col == 0 or col == m - 1):
                arr[row][col] = 1

    return arr
```

Exercise

```python
def createMatrixVer2(n: int, m: int):
    arr = np.zeros((n, m), dtype = np.int64)
    arr[0,:] = 1
    arr[n-1,:] = 1
    arr[:, 0] = 1
    arr[:, m-1] = 1
    return arr
```

Exercise

```python
def createMatrixVer3(n: int, m: int):
    arr = np.ones((n, m), dtype = np.int64)
    arr[1:-1, 1:-1] = 0
    return arr
```

Exercise

Ex5: Create a NxN matrix as follows:

N = 4

```
[[100   1   2   3]
 [  4 100   6   7]
 [  8   9 100  11]
 [ 12  13  14 100]]
```

Exercise

```python
def createMatrix(n: int):
    content = list(range(n*n))
    arr = np.array(content)
    arr = arr.reshape(n, -1)
    for row in range(n):
        for col in range(n):
            if (col == row):
                arr[row][col] = 100
    return arr
```

Exercise

```python
def createMatrixVer2(n: int):
    arr = np.arange(n*n)
    arr[0::5] = 100
    return arr.reshape(n, -1)
```

Exercise

Ex6: Write a function to multiply two matrices NxN without using np.dot()

Exercise

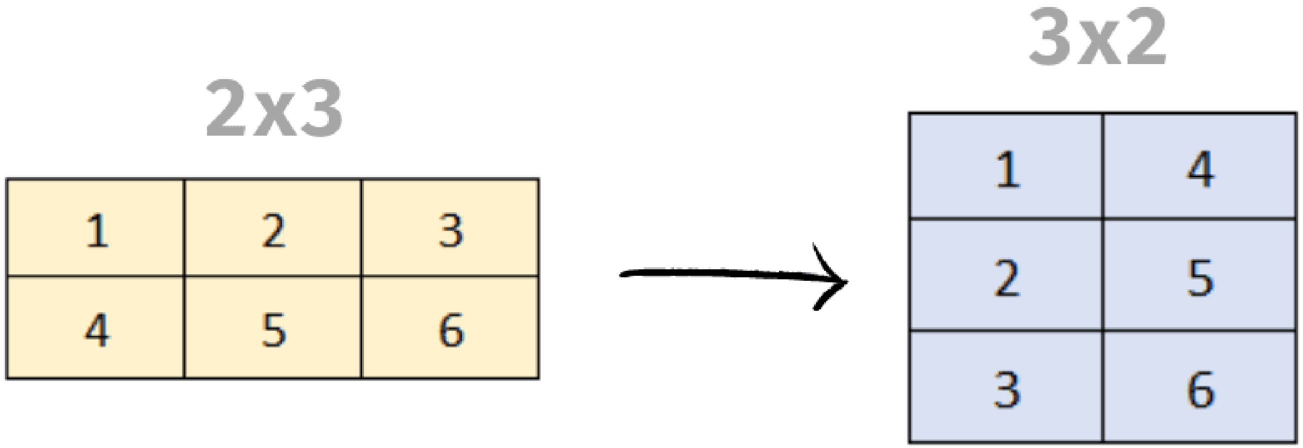
```
def matrix_multiply(a, b):
    n = len(a)
    c = np.array([[0 for i in range(n)] for j in range(n)])
    for i in range(n):
        for j in range(n):
            for k in range(n):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

Exercise

```
def matrixMul(a, b):
    return a@b
```

Transposing Arrays

- In Python, numpy.`transpose()` is used to get the permute or reserve the dimension of the input array



`Transpose array`

Transposing Arrays

`numpy.transpose(a, axes=None)` [source]

Returns an array with axes transposed.

For a 1-D array, this returns an unchanged view of the original array, as a transposed vector is simply the same vector. To convert a 1-D array into a 2-D column vector, an additional dimension must be added, e.g., `np.atleast2d(a).T` achieves this, as does `a[:, np.newaxis]`. For a 2-D array, this is the standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided, then `transpose(a).shape == a.shape[::-1]`.

Transposing Arrays

`numpy.transpose(a, axes=None)`

Returns an array with axes transposed.

Parameters: **a** : *array_like*

Input array.

axes : *tuple or list of ints, optional*

If specified, it must be a tuple or list which contains a permutation of [0,1,..,N-1] where N is the number of axes of *a*. The *i*'th axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, defaults to `range(a.ndim)[::-1]`, which reverses the order of the axes.

Returns: **p** : *ndarray*

a with its axes permuted. A view is returned whenever possible.

Transposing Arrays

```
# 1-D array
a = np.array([1, 2, 3, 4])
print(a.transpose()) # [1 2 3 4]


#2-D array
a = np.array([[1, 2], [3, 4]])
print(a.transpose())
# [[1 3]
#  [2 4]]
```

Transposing Arrays

- The result of the following code:

```
a = np.arange(8)
a = a.reshape((4, 2))
print(a.T)
```

Transposing Arrays

- The result of the following code:

```
a = np.arange(8)
a = a.reshape((4, 2))
print(a.T)
# [[0 2 4 6]
#  [1 3 5 7]]
```

- The result of the following code:

```
a = np.arange(6).reshape((3, 2))
print(a.dot(a.T))
```

$$
\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 13 & 23 \\ 5 & 23 & 41 \end{bmatrix}
$$

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

×

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

=

```
[[ 1  3  5]
 [ 3 13 23]
 [ 5 23 41]]
```

Transposing Arrays

- 3-d Array

```
a = np.arange(2*2*3).reshape((2,2,3))
print(a)
# [[[ 0  1  2]
#   [ 3  4  5]]
#  [[ 6  7  8]
#   [ 9 10 11]]]
print(a.T)
# [[[ 0  6]
#   [ 3  9]]
#  [[ 1  7]
#   [ 4 10]]
#  [[ 2  8]
#   [ 5 11]]]
```

Transposing Arrays

- 3-d Array

```python
a = np.arange(2*2*3).reshape((2,2,3))
print(a.transpose())
# equivalent to
print(a.transpose(2, 1, 0))
# [[[ 0  6]
#   [ 3  9]]
#  [[ 1  7]
#   [ 4 10]]
#  [[ 2  8]
#   [ 5 11]]]
```

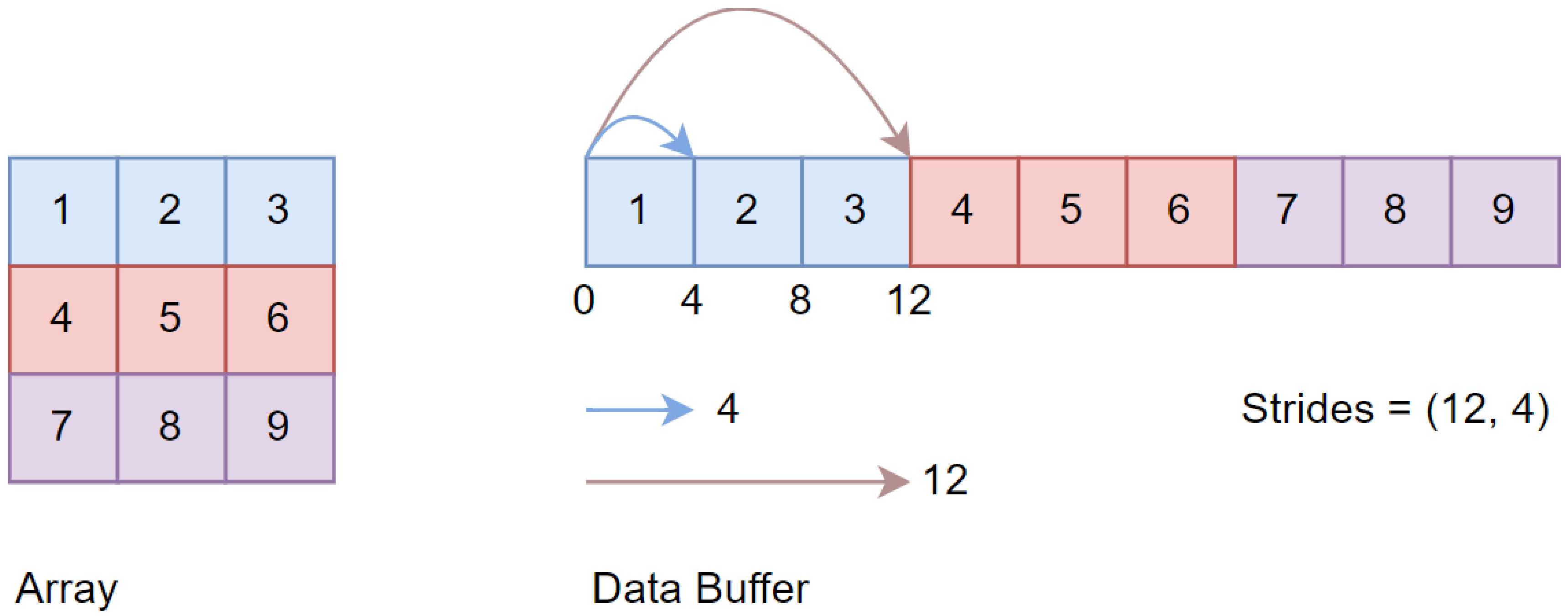
Transposing Arrays

- 3-d Array

```
a = np.arange(12).reshape((2, 2, 3))
print(a.transpose(1, 0, 2))
# [[[ 0  1  2]
#   [ 6  7  8]]
#
#  [[ 3  4  5]
#   [ 9 10 11]]]
```

Strides in ndarray

- A stride is a tuple of integer numbers, each of which indicates the bytes for a particular dimension
- NumPy uses strides to tell how many bytes to jump in the data buffer



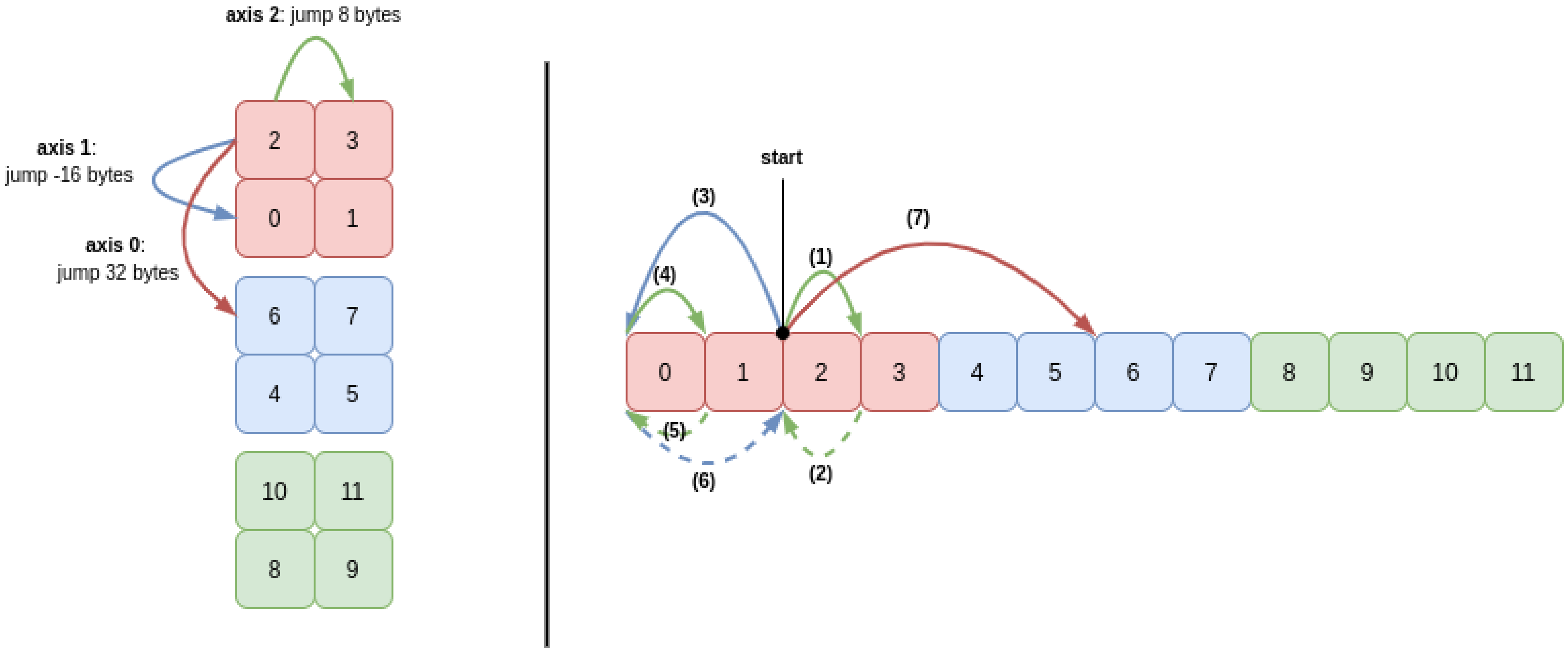
Strides in ndarray

- When we transpose an array, its stride is also changed via its corresponding axis.

```
a = np.arange(9).reshape(3,3)
print(a.strides) # (12, 4)
b = a.transpose()
print(b.strides) # (4, 12)
```

Strides in ndarray

- In 3-d Array:



Transposing Arrays

- What is the output of the following code

```
a = np.arange(16).reshape((2, 2, 4))
b = a.transpose(1, 0, 2)
print(b)
```

Transposing Arrays

- What is the output of the following code

```
a = np.arange(16).reshape((2, 2, 4))
b = a.transpose(1, 0, 2)
print(b)
```

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

 [[ 8  9 10 11]
  [12 13 14 15]]]
```

```
[[[ 0  1  2  3]
  [ 8  9 10 11]]

 [[ 4  5  6  7]
  [12 13 14 15]]]
```

Exercise 01

- Given an array as follows:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

- Let convert to

```
array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Exercise 01


```python
import numpy as np
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

matrix = arr.reshape(3, 4)

# Transpose the matrix
transposed_matrix = matrix.T

# Flatten the transposed matrix back into a 1D array
result_arr = transposed_matrix.flatten()

print(result_arr)
```

np.swapaxes()

- **`numpy.swapaxes()`** function interchange two axes of an array

`numpy.swapaxes(a, axis1, axis2)` [source]

Interchange two axes of an array.

Parameters:

a : *array_like*

Input array.

axis1 : *int*

First axis.

axis2 : *int*

Second axis.

Returns:

a_swapped : *ndarray*

For NumPy >= 1.10.0, if *a* is an ndarray, then a view of *a* is returned; otherwise a new array is created. For earlier NumPy versions a view of *a* is returned only if the order of the axes is changed, otherwise the input array is returned.

np.swapaxes()

- **`numpy.swapaxes()`** is quite similar to transpose

```
a = np.arange(9).reshape(3,3)
print(a.transpose())
print(a.swapaxes(0, 1))
# [[0 3 6]
#  [1 4 7]
#  [2 5 8]]
```

np.swapaxes()

- What is the output of the following code:

```
a = np.arange(12).reshape((3,2,2))
print(a.swapaxes(1, 2))
```

np.swapaxes()

- What is the output of the following code:

```
a = np.arange(12).reshape((3,2,2))
print(a.swapaxes(1, 2))
```

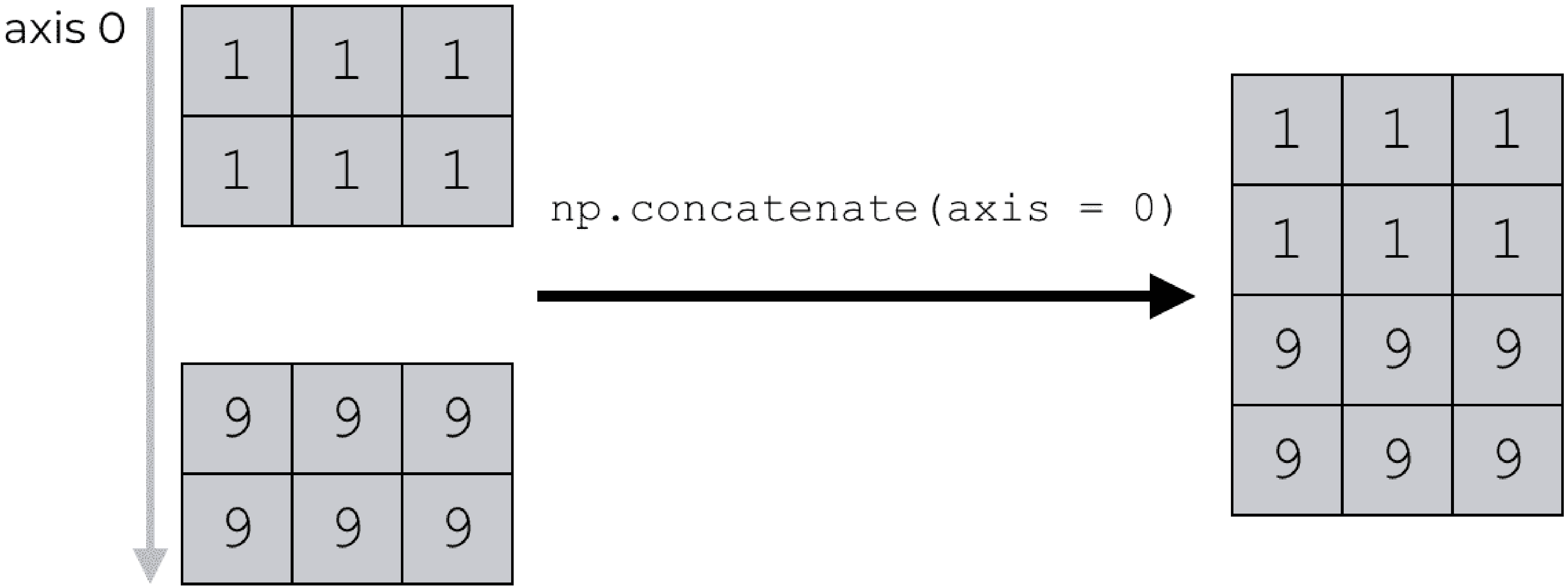
```
[[[ 0  2]
  [ 1  3]]

 [[ 4  6]
  [ 5  7]]

 [[ 8 10]
  [ 9 11]]]
```

Concatenation

- Concatenation refers to joining.
- This function is used to join two or more arrays of the same shape along a specified axis

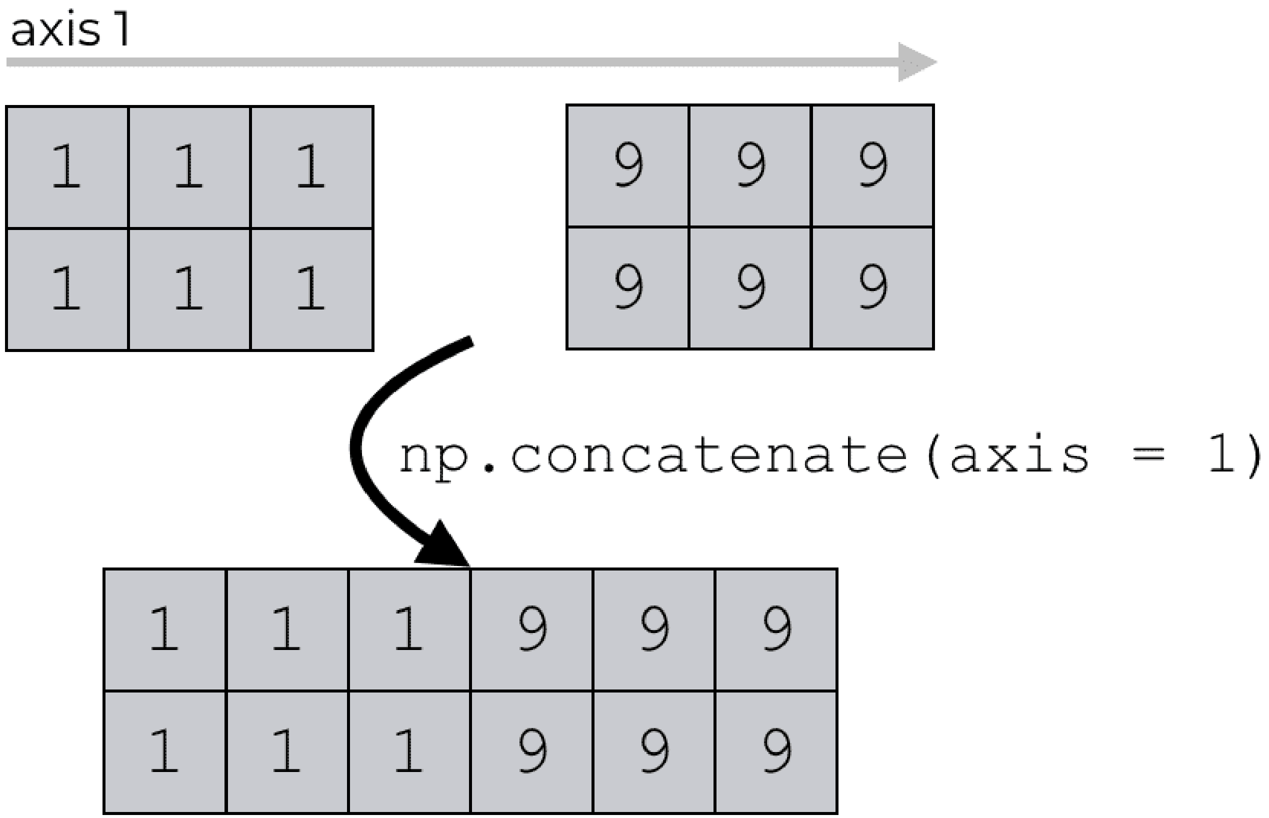
```
numpy.concatenate((a1, a2, ...), axis)
```

Setting `axis=0` concatenates along the row axis

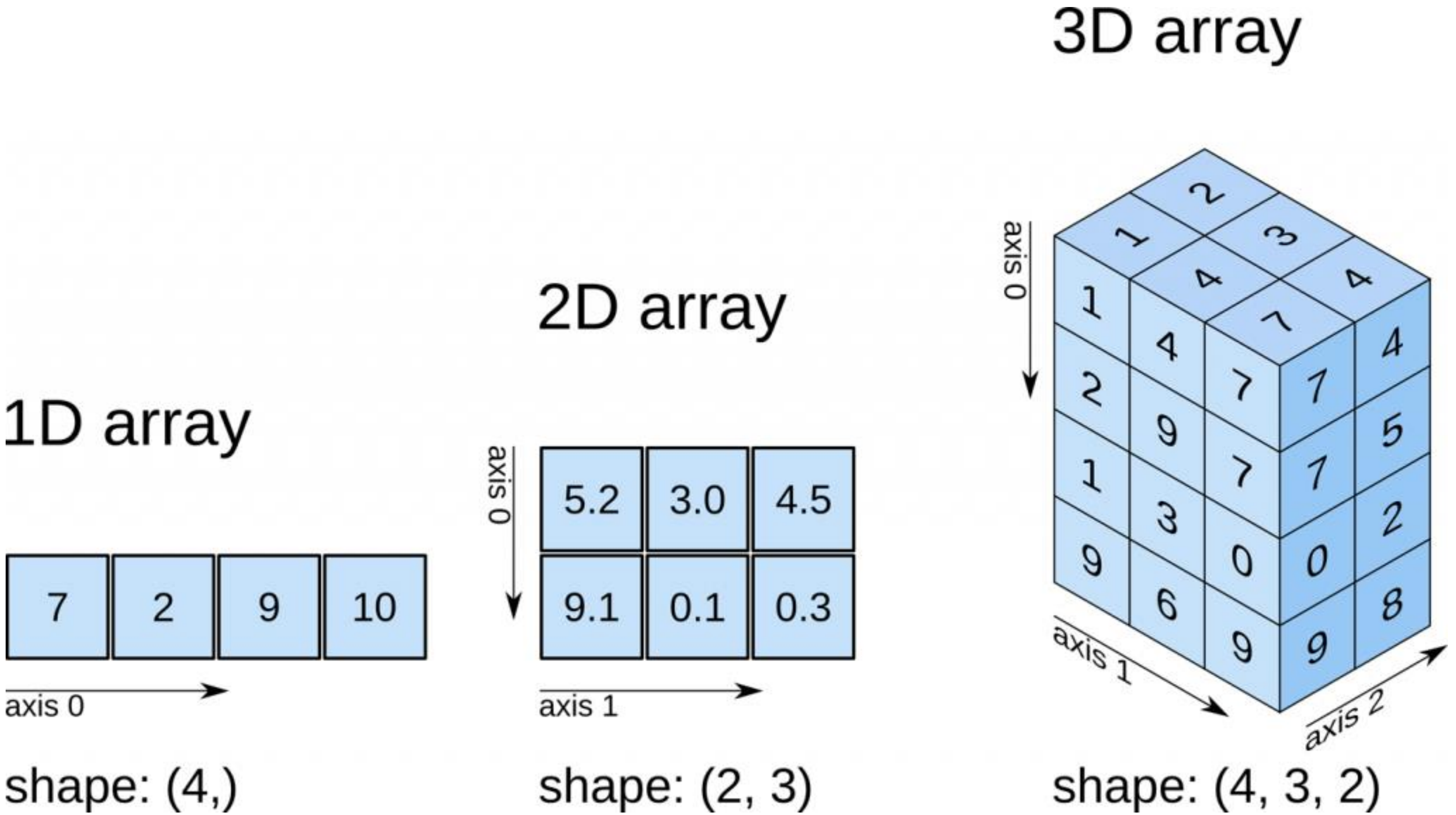


Concatenation

Setting `axis=1` concatenates along the column axis



Concatenation



Concatenation

```python
a = np.arange(12).reshape((3,2,2))
b = a + 2
c = np.concatenate([a, b], axis = 2)
print(c)
```

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

 [[ 4  5  6  7]
  [ 6  7  8  9]]

 [[ 8  9 10 11]
  [10 11 12 13]]]
```

Concatenation


```python
a = np.arange(12).reshape((3,2,2))
b = a + 2
c = np.concatenate([a, b], axis = 1)
print(c)
```

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

 [[ 4  5]
  [ 6  7]
  [ 6  7]
  [ 8  9]]

 [[ 8  9]
  [10 11]
  [10 11]
  [12 13]]]
```

np.repeat()


```
arr = np.arange(3) # array([0, 1, 2])

arr.repeat(3) # array([0, 0, 0, 1, 1, 1, 2, 2, 2])

arr.repeat([2, 3, 4]) # array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

np.repeat()

```python
a = np.arange(4).reshape((1,2,2))
print(a)
# [[[0 1]
#   [2 3]]]
b = a.repeat(2)
print(b) # [0 0 1 1 2 2 3 3]
```

np.repeat()

```
arr = np.arange(4).reshape(2, 2)
arr
```

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

```
arr.repeat(2, axis=0)
```

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

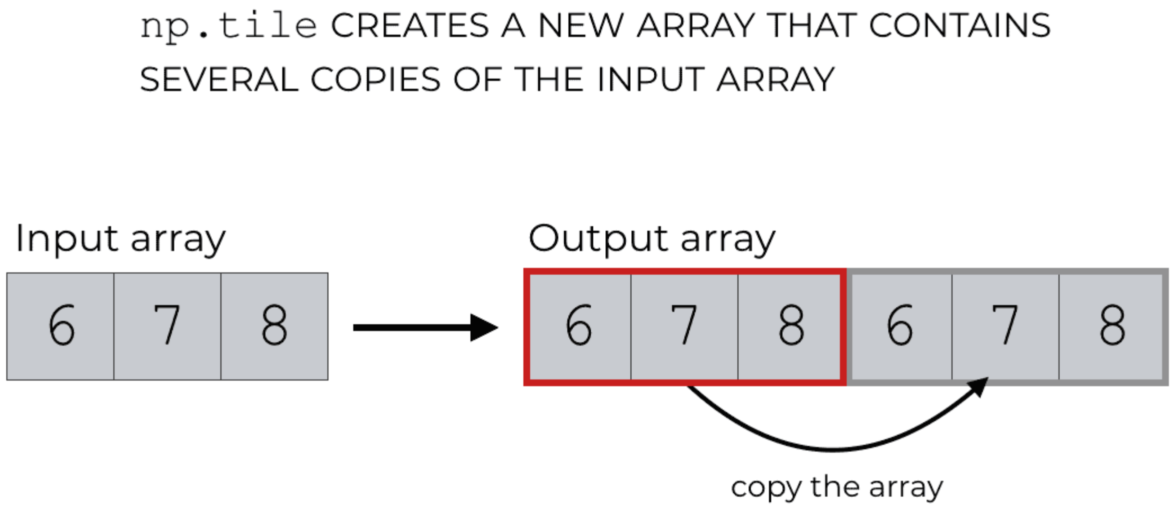
```
arr.repeat([2, 3], axis=0)
```

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

```
arr.repeat([2, 3], axis=1) #?
```

numpy.tile()

- numpy.tile() function constructs a new array by repeating array




```python
a = np.array([1, 2, 3, 4, 5])
b = np.tile(a, 2)
c = a.repeat(2)
print(b) # [1 2 3 4 5 1 2 3 4 5]
print(c) # [1 1 2 2 3 3 4 4 5 5]
```

numpy.tile()

```
[ ]  np.tile(arr,2)
```

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

```
[ ]  np.tile(arr,(2,1))
```

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

```
[ ]  np.tile(arr,(2,2)) #?
```

numpy.tile()


```
[ ]  np.tile(arr,2)
```

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

```
[ ]  np.tile(arr,(2,1))
```

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

```
[ ]  np.tile(arr,(2,2)) #?
```

```
[[0 1 0 1]
 [2 3 2 3]
 [0 1 0 1]
 [2 3 2 3]]
```

```
#Universal functions which perform element-wise operations on ndarrays

arr = np.random.randn(10)

np.square(arr)
```

```
array([3.16274687e+00, 2.29088562e+00, 6.14659767e-06, 1.11834661e-01,
       7.97263403e-02, 1.15980131e+00, 7.93531615e-01, 5.14354487e-01,
       2.02893154e+00, 2.65375327e-01])
```

```
np.exp(arr)
```

```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

```
x = np.random.randn(8) #array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,-0.6605])
y = np.random.randn(8) #array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,-2.3042])
np.maximum(x,y) # array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,-0.6605])
```

Universal functions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

Binary functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `& |, ^`)

Conditional logic as Array operations


```
[ ]  #Conditional logis as Array Operations
     xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
     yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
     cond = np.array([True, False, True, True, False])

     result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]
     result
```

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

```
[ ]  result = np.where(cond, xarr, yarr)
     result
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Exercise 02

- Create a random `NxM` matrix. Set all positive values `x` to `2x`, and `-2` for negative values

Exercise 02


```python
# Parameters for the matrix dimensions
N = 4  # Number of rows
M = 5  # Number of columns

# Create a random NxM matrix with values from -10 to 10
matrix = np.random.randn(N, M) * 10

# Apply the condition:
# Double positive values, set negative values to -2
modified_matrix = np.where(matrix > 0, 2 * matrix, -2)

# Print the modified matrix
print("Modified Matrix:")
print(modified_matrix)
```

```
[ ] #Mathematical and Statistical methods
    arr = np.random.randn(5, 4)
    arr
```

```
array([[-0.54926229,  0.18278111, -0.83024367,  0.69999463],
       [-0.1905132 , -0.34356363,  1.0181388 ,  0.24849718],
       [ 0.46647855,  0.33590169,  0.44325881,  0.20612658],
       [ 0.2589447 ,  0.29961133, -0.24995255,  0.49049097],
       [-0.46239346, -0.30404133,  1.20203899, -1.21725287]])
```

```
[ ] arr.mean()
    arr.sum()
```

```
1.7050403500621718
```

```
[ ] arr.mean(axis=1)
```

```
array([-0.12418255,  0.18313979,  0.36294141,  0.19977361, -0.19541217])
```

```
[ ] arr.mean(axis=0)
```

```
array([-0.09534914,  0.03413784,  0.31664808,  0.0855713 ])
```

Mathematical & Statistical methods

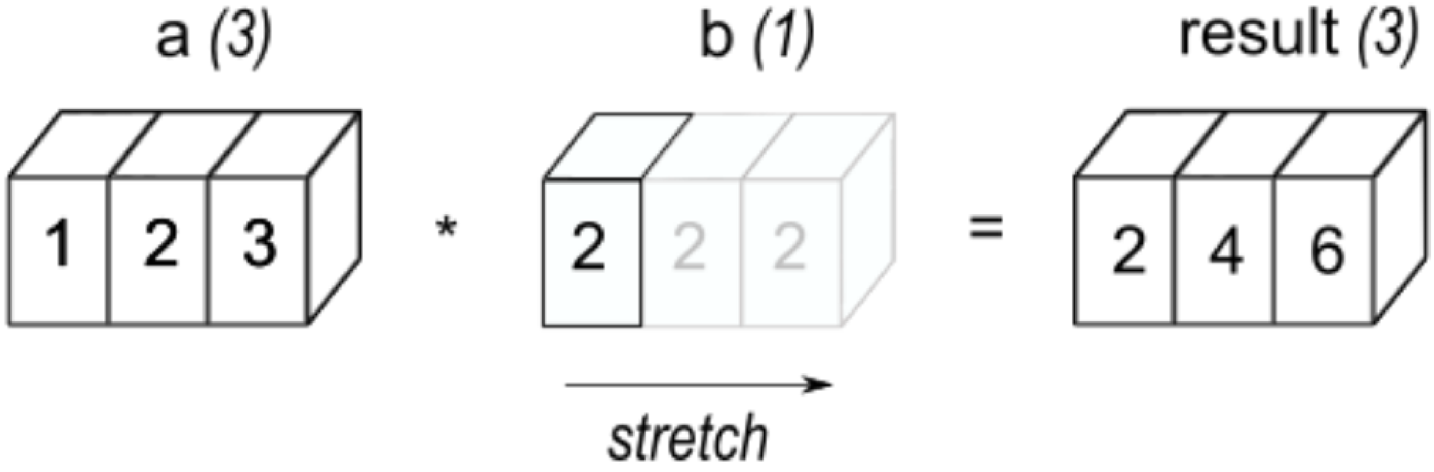
Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; zero-length arrays have NaN mean
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`)
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

File with numpy


```
[ ]  #File with numpy

     arr = np.arange(10)
     np.save('some_array', arr)
```

```
[ ]  np.load('some_array.npy')

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

```
a = np.array([1, 2, 3])
print(a *2)
```



Broadcasting


```
a = np.arange(12).reshape(4, 3) # shape = (4, 3)
b = a.mean(0)
print(b) # [4.5 5.5 6.5], shape = (3,)
print(a - b) # shape = (4, 3)
# [[-4.5 -4.5 -4.5]
#  [-1.5 -1.5 -1.5]
#  [ 1.5  1.5  1.5]
#  [ 4.5  4.5  4.5]]
```

```
[ ] arr = np.arange(5) #array([0, 1, 2, 3, 4])
    arr*4
```

```
array([ 0,  4,  8, 12, 16])
```

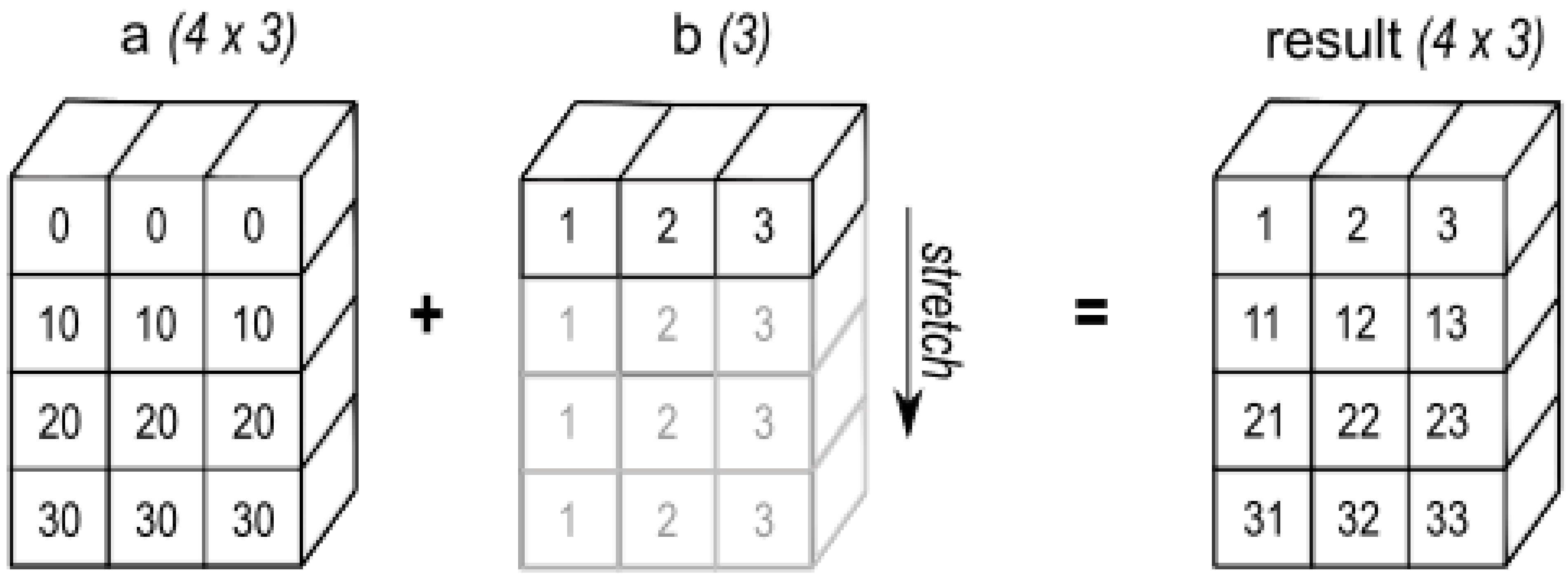
```
[ ] arr = np.random.randn(4, 3)
    arr.mean(0)
```

```
array([0.48837945, 0.52888814, 0.08470346])
```

```
[ ] de = arr - arr.mean(0)
    de
```

```
array([[ 0.04160849,  0.38942007,  0.77749224],
       [ 0.49715569, -1.44295791, -0.95987177],
       [ 0.39977265,  0.2456237 ,  1.06945136],
       [-0.93853683,  0.80791414, -0.88707184]])
```

Broadcasting



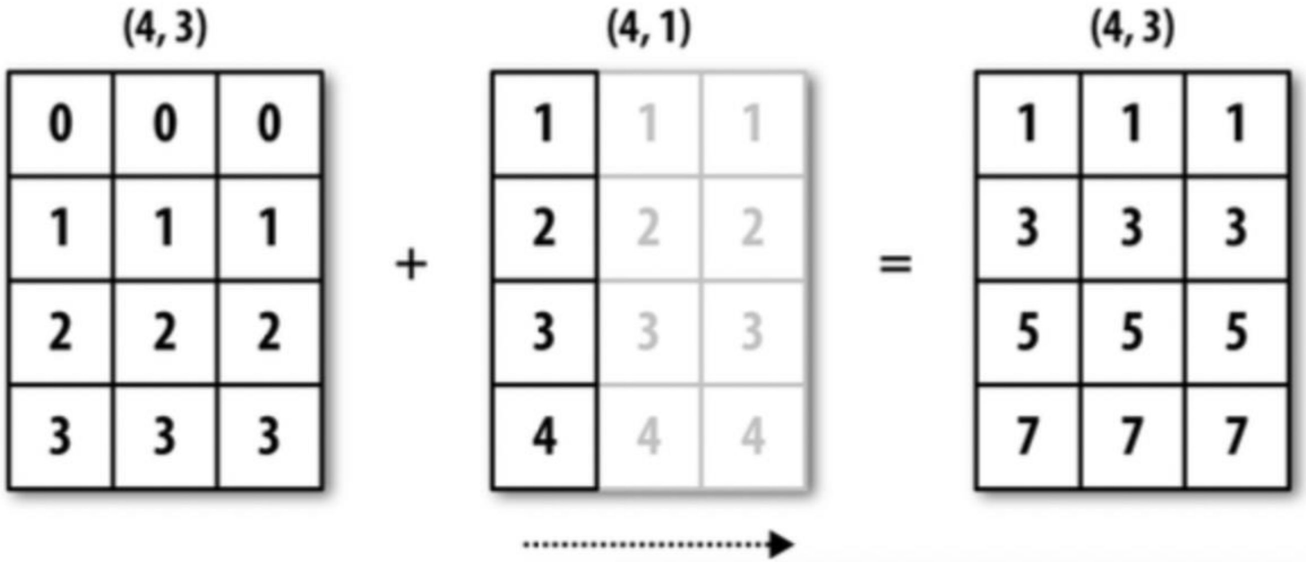
Broadcasting

```
[ ] arr.mean(1)

array([ 0.77016395, -0.26790098,  0.93893959,  0.02809217])
```

```
[ ] de = arr - arr.mean(1) #?
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-952255e5b562> in <module>()
----> 1 de = arr - arr.mean(1) #?
      2

ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

SEARCH STACK OVERFLOW



```python
a = np.arange(12).reshape(4, 3)
b = np.array([1, 2, 3, 4]).reshape(4, 1)
print(a + b)
# [[ 1  2  3]
#  [ 5  6  7]
#  [ 9 10 11]
#  [13 14 15]]
```

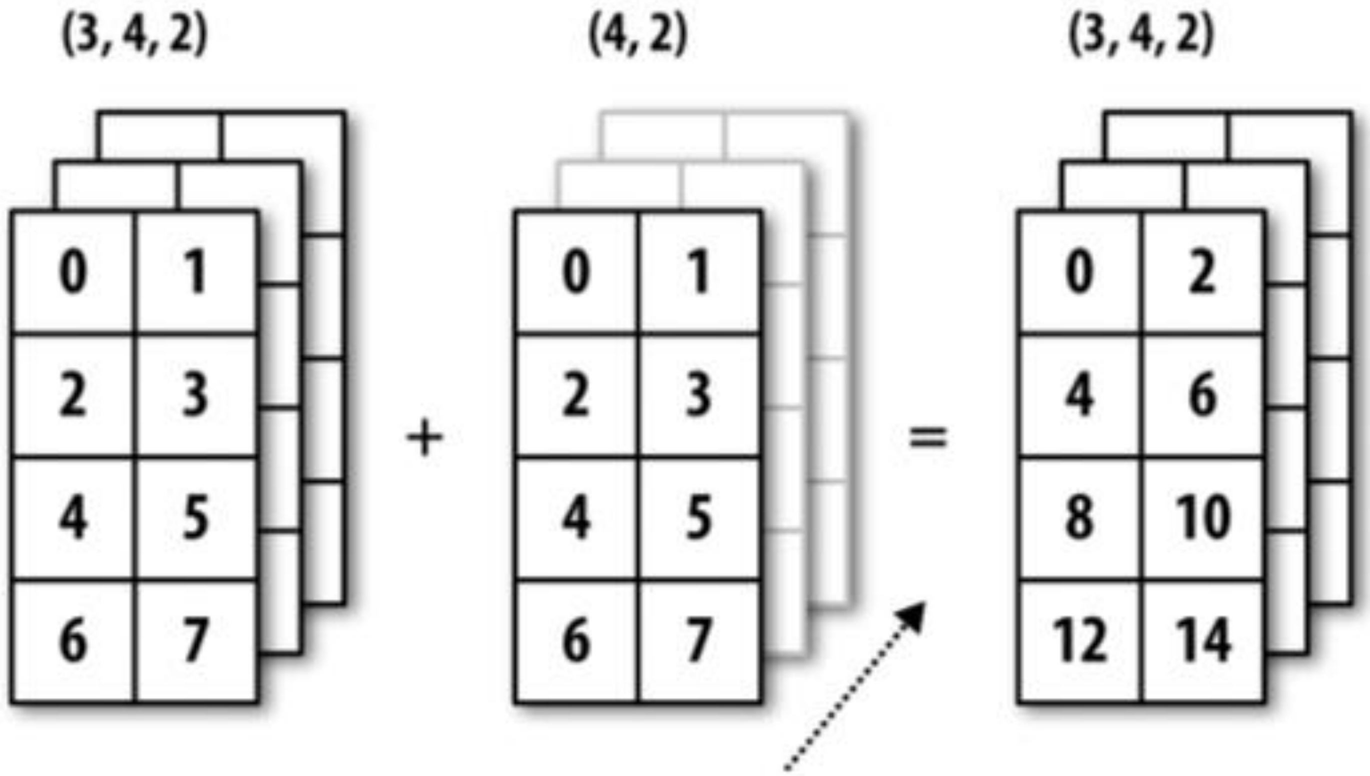
Broadcasting


```
a = np.arange(12).reshape(4, 3)
b = np.array([1, 2, 3, 4])
#print(a - b) # error
print(a - b[:, np.newaxis])
# [[-1  0  1]
#  [ 1  2  3]
#  [ 3  4  5]
#  [ 5  6  7]]
```

Broadcasting

- Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays
  - Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side. (e.g: (4,3) and (4,) => (4,3) and (1, 4))
  - Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape. (e.g: (4, 3) and (1, 4) => (4, 3) and (4, 4))
  - Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Broadcasting



Exercise 03

- Create the following array with n inputted from keyboard.
- Ex: n = 10

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

Exercise 03

```python
import numpy as np

def create_matrix(n: int) -> np.ndarray:
    arr = np.arange(n).reshape(2, -1)
    ones = np.ones(n, dtype= int).reshape(2, -1)
    result = np.concatenate([arr, ones], axis = 1)
    return result

print(create_matrix(10))
```

Exercise 04

- Given `a = np.array([1,2,3])`
- Create a following array without using iteration and array initialization:

[1, 1, 1, 2, 2, 2, 3, 3, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

Exercise 04


```
# Given array
a = np.array([1, 2, 3])

# Repeat each element three times
part1 = np.repeat(a, 3)
# Tile the entire array three times
part2 = np.tile(a, 3)
# Concatenate the two parts
result = np.concatenate((part1, part2))

# Print the result
print(result)
```

Exercise 05

- What is the output of the code fragment

```
import numpy as np

arr = np.arange(9).reshape(3,3)
print(arr[::-1])
```

Exercise 05

- What is the output of the code fragment

```
import numpy as np

arr = np.arange(9).reshape(3,3)
print(arr[::-1])
```

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

Exercise 06

- Given a even number n from keyboards, create a 2*((N*N)//2) array as follows:
- Ex: n = 4

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

Exercise 06

```
import numpy as np

def create_interleaved_array(n):
    if n % 2 != 0: return None
    positive_numbers = np.arange(n * n // 2)
    negative_numbers = -1*np.arange(n * n // 2, n*n)

    result_array = np.empty(n * n, dtype=int)
    result_array[::2] = positive_numbers
    result_array[1::2] = negative_numbers
    # Reshape the result into a 2x(N*N//2) array
    final_array = result_array.reshape(2, n * n // 2)
    return final_array

print(create_interleaved_array(4))
```

THANK YOU

for YOUR ATTENTION