fit@hcmus
VNUHCM - UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

University of Science – VNU-HCM
Faculty of Information Science
Department of Computer Science
MTH083 - Advanced Programming for Artificial Intelligence

Slot 07- Numpy – Part 02

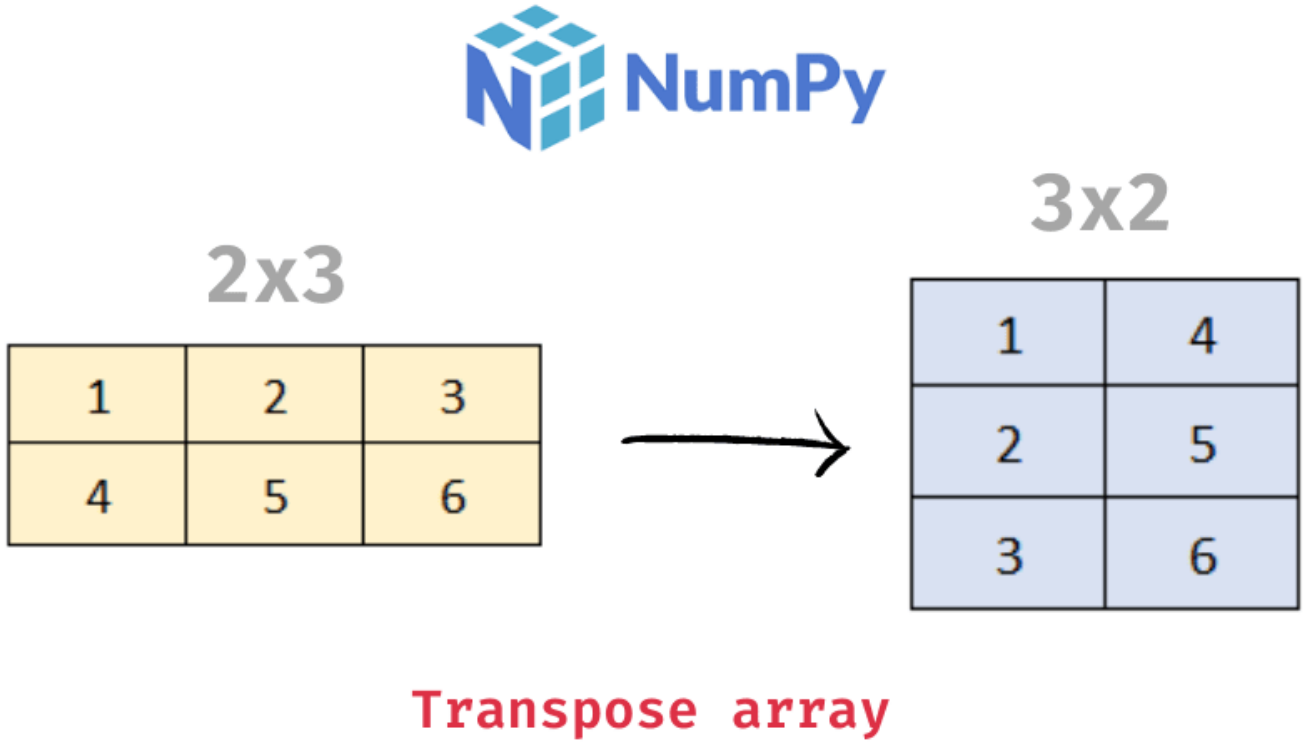
Advisor:

Dr. Nguyễn Tiến Huy

Dr. Lê Thanh Tùng

Transposing Arrays

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

Transposing Arrays

- The result of the following code:

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

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

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

```
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]]]
```

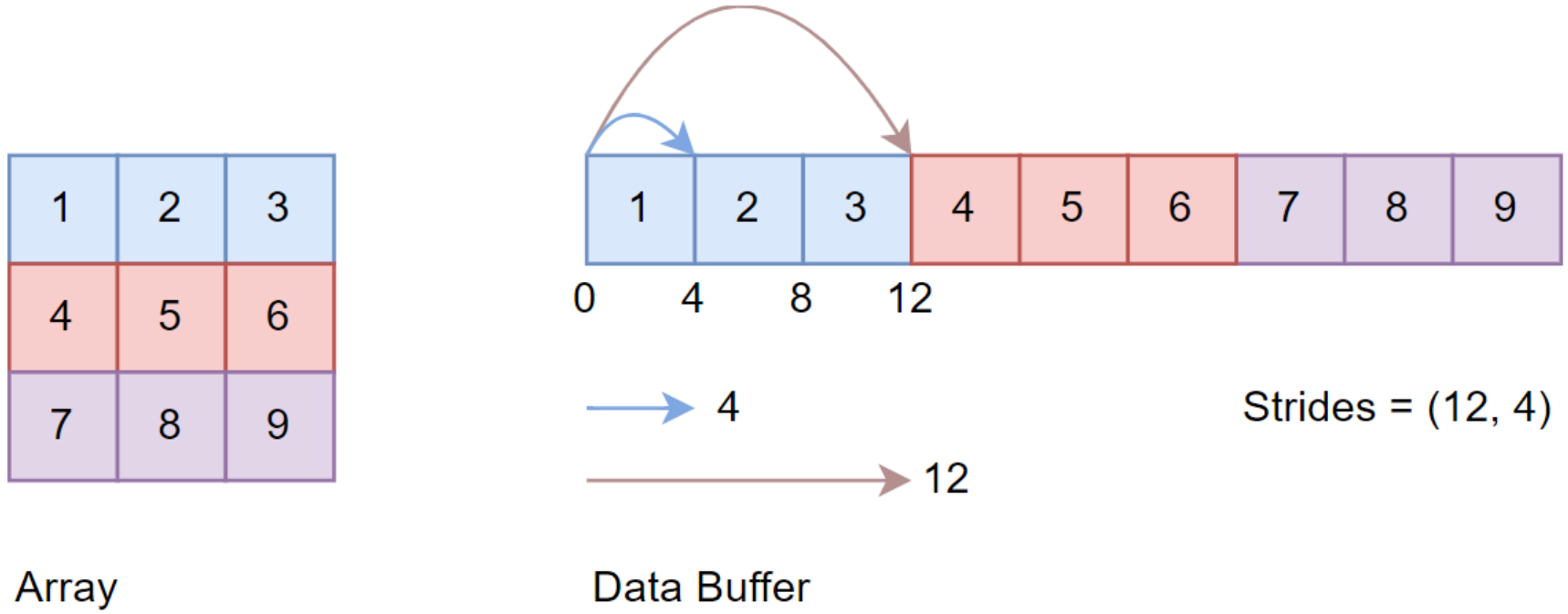
Transposing Arrays

- 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



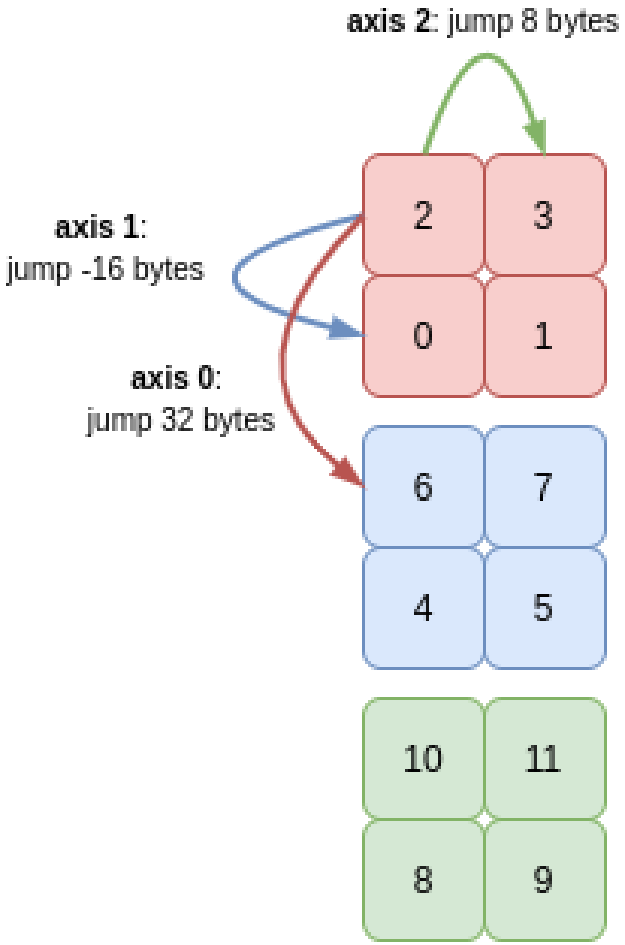
Strides in ndarray

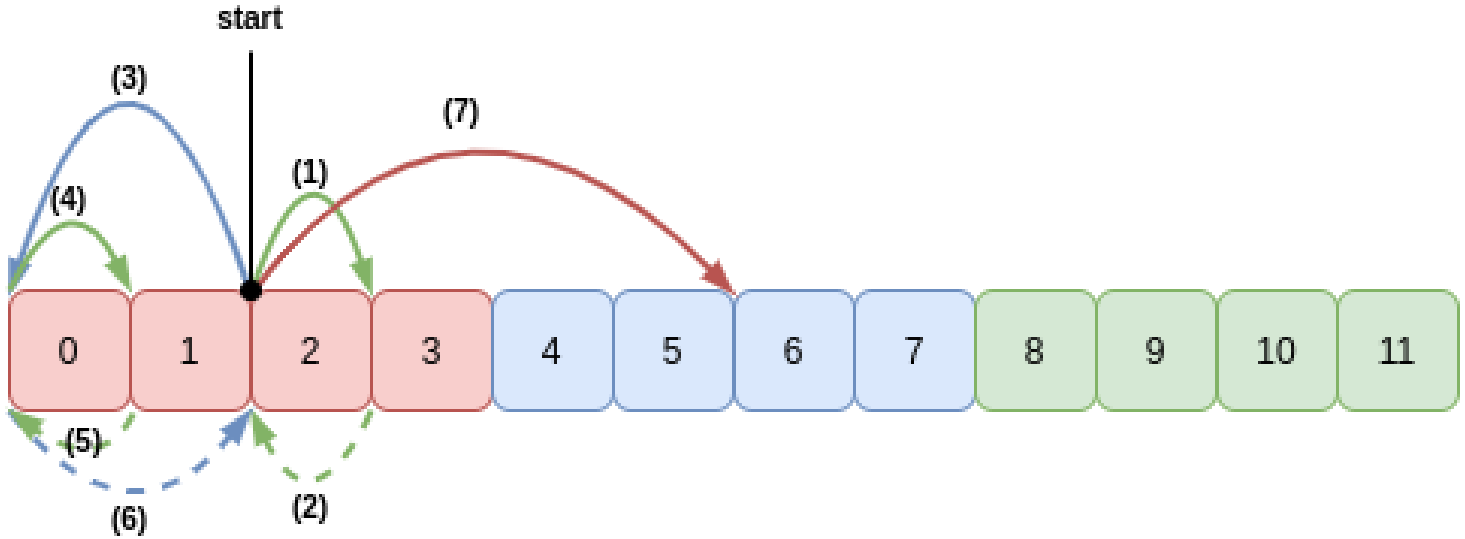
- When we transpose an array, its stride is also changed via its corresponding axis.

```
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]
  [ 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])
```

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))
```

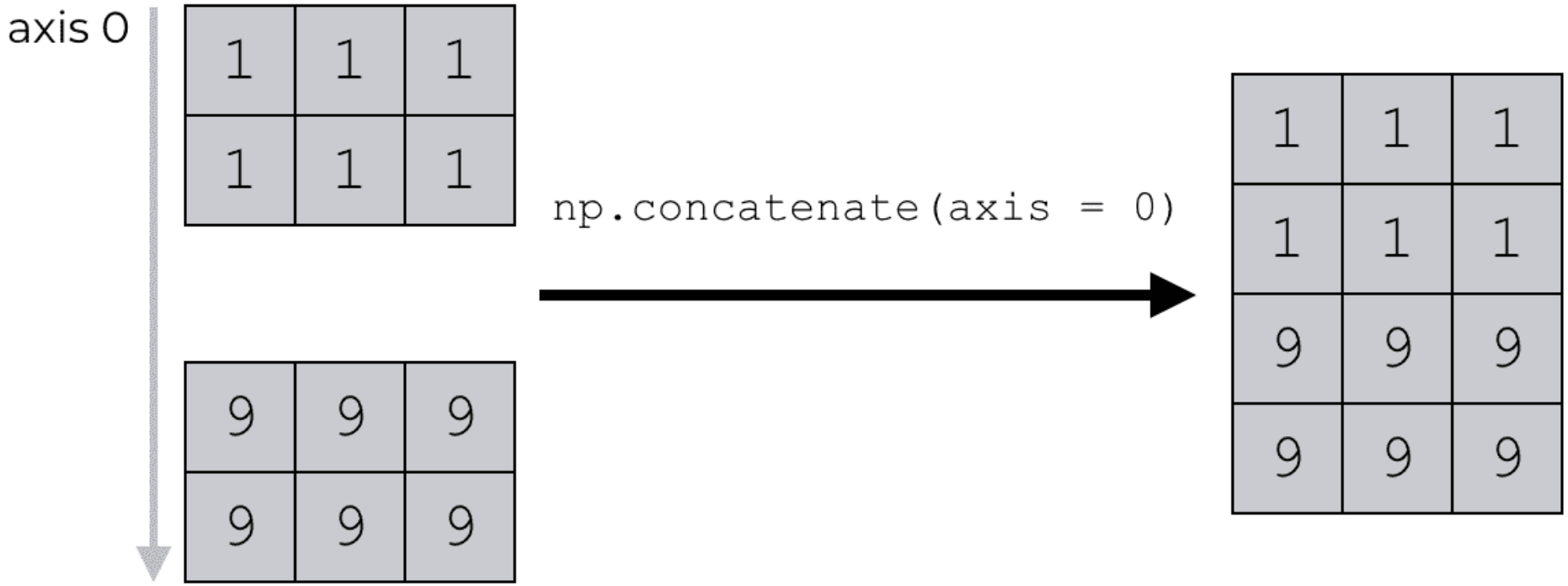
```
[[[ 0  2]
  [ 1  3]]

 [[ 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

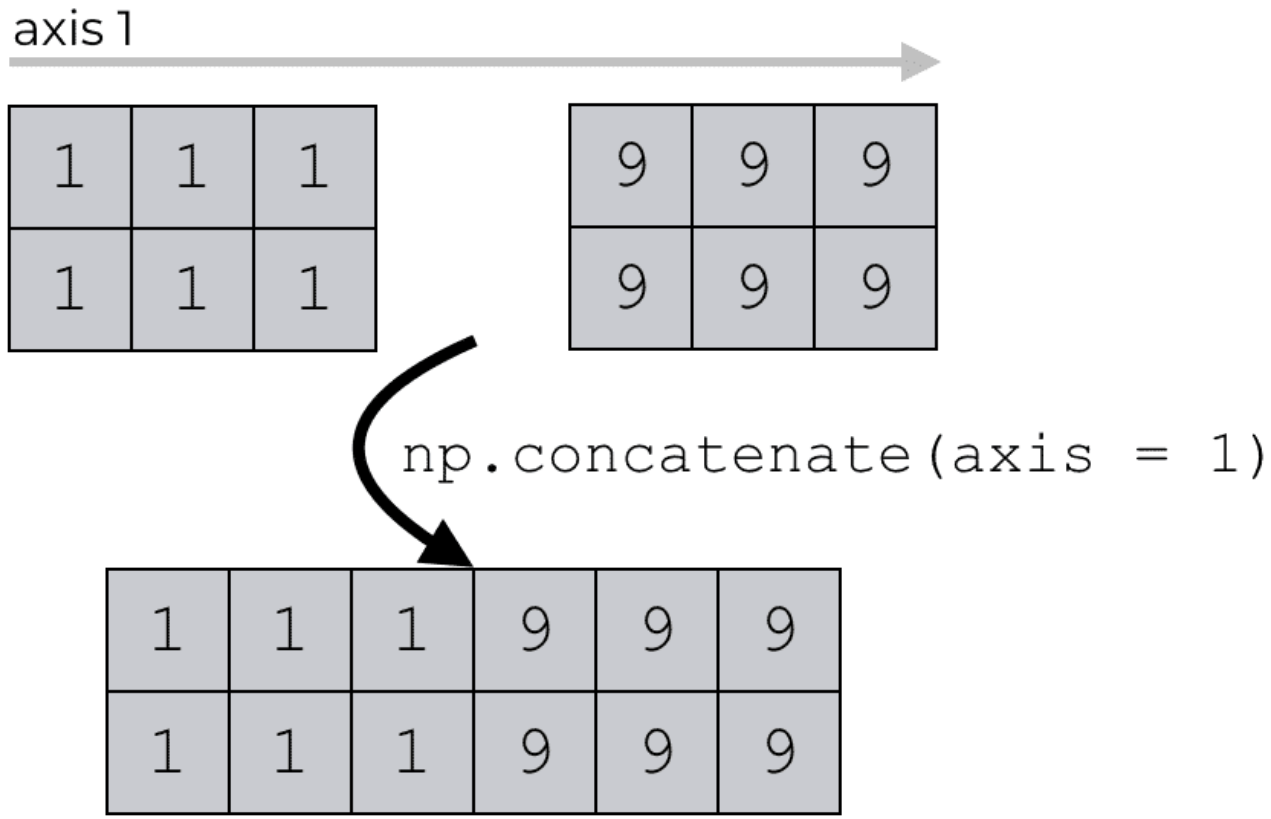
```
numpy.concatenate((a1, a2, ...), axis)
```

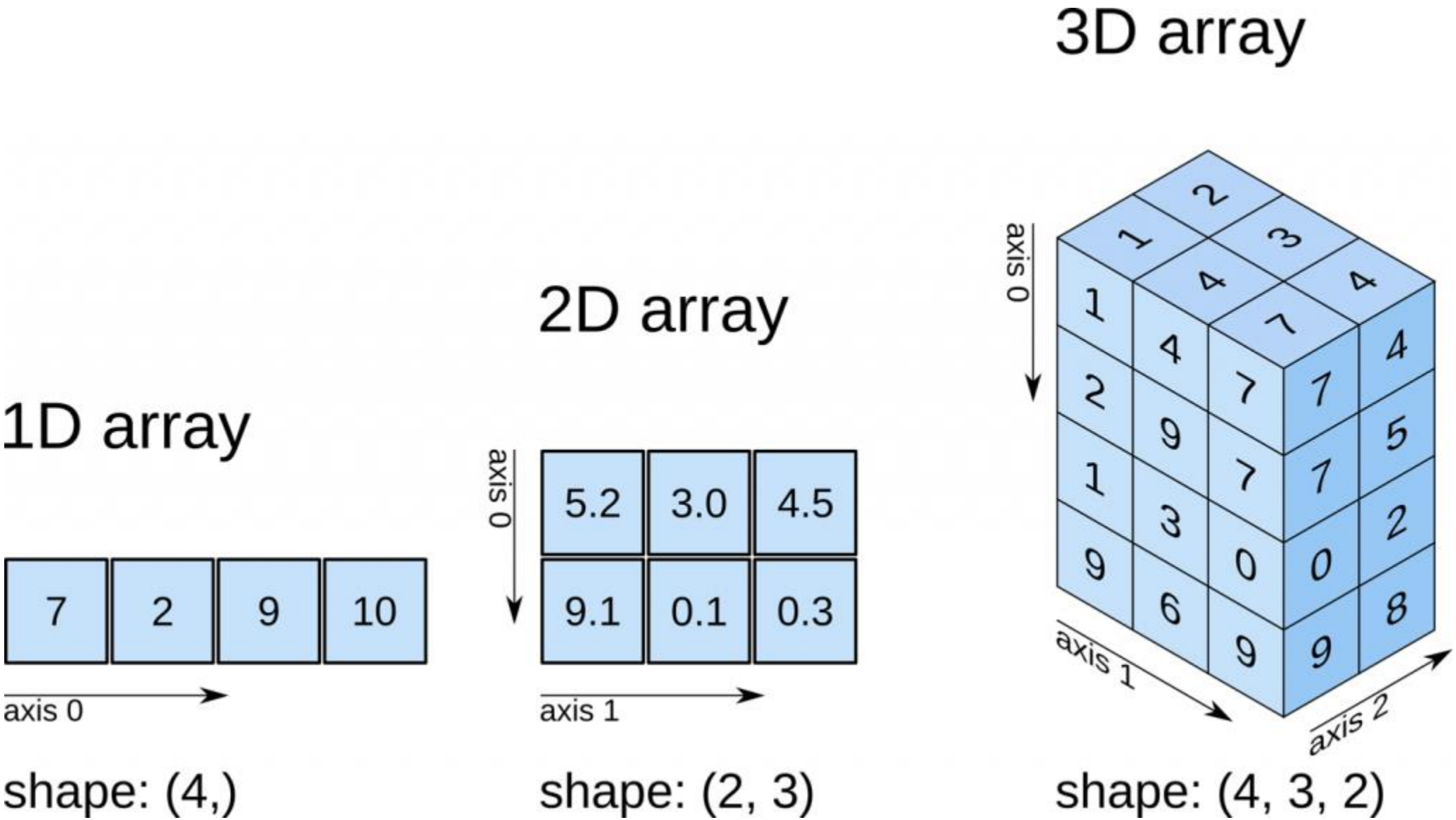
Setting `axis=0` concatenates along the row axis



Concatenation

Setting `axis=1` concatenates along the column axis




1D array
7
2
9
10
axis 0
shape: (4,)
2D array
axis 0
5.2
3.0
4.5
9.1
0.1
0.3
axis 1
shape: (2, 3)
3D array
axis 0
axis 1
axis 2
shape: (4, 3, 2)

Concatenation

```
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


```
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()

```
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)
```

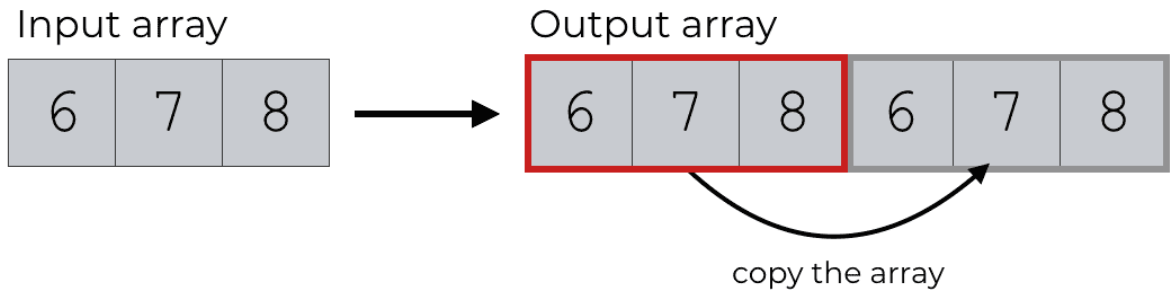
```
array([[0, 1],
       [0, 1],
       [2, 3],
       [2, 3],
       [2, 3]])
```

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

numpy.tile()

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

`np.tile` CREATES A NEW ARRAY THAT CONTAINS SEVERAL COPIES OF THE INPUT ARRAY




```
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

```
#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

```
[ ]  #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 ])
```

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

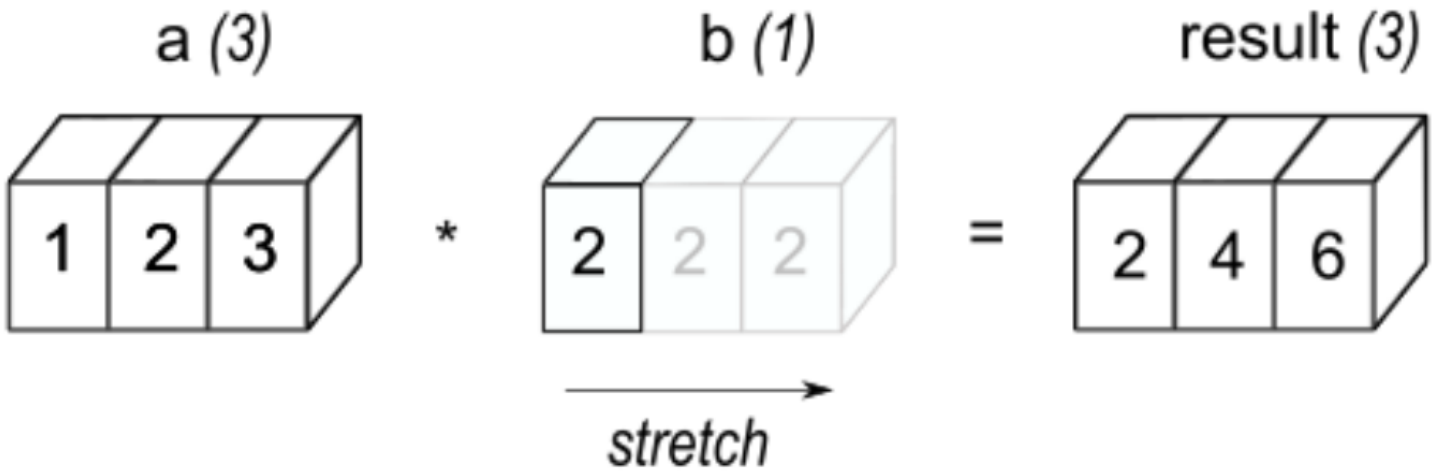
```
[ ]  #File with numpy

     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])
```

Broadcasting

```
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]]
```

Broadcasting

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

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

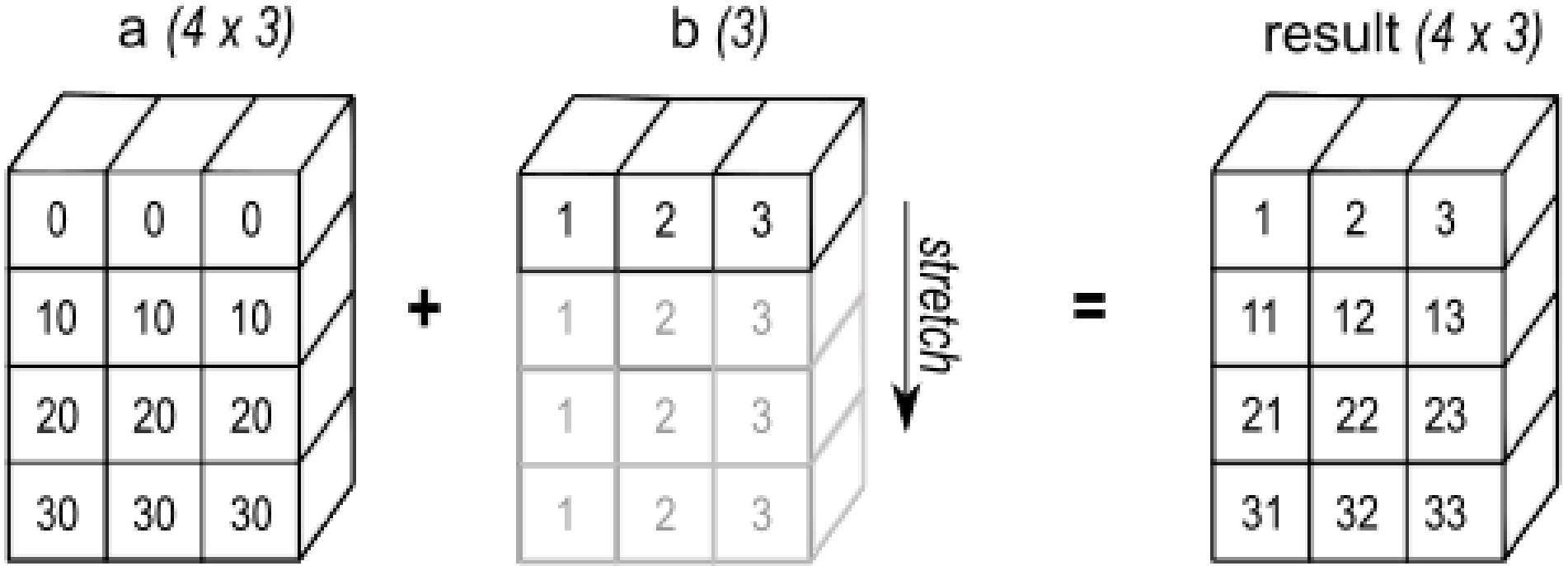
```
[ ]  arr = np.random.randn(4, 3)
     arr.mean(0)
```

```
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

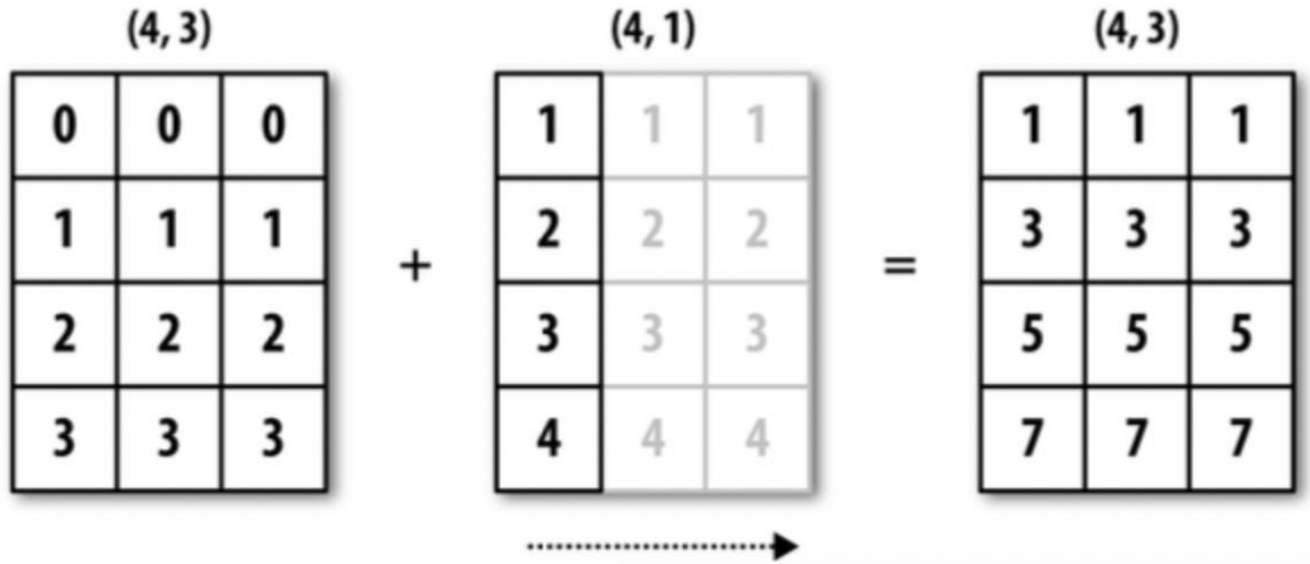
```
[ ]  arr.mean(1)
```

```
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



Broadcasting

```
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]]
```

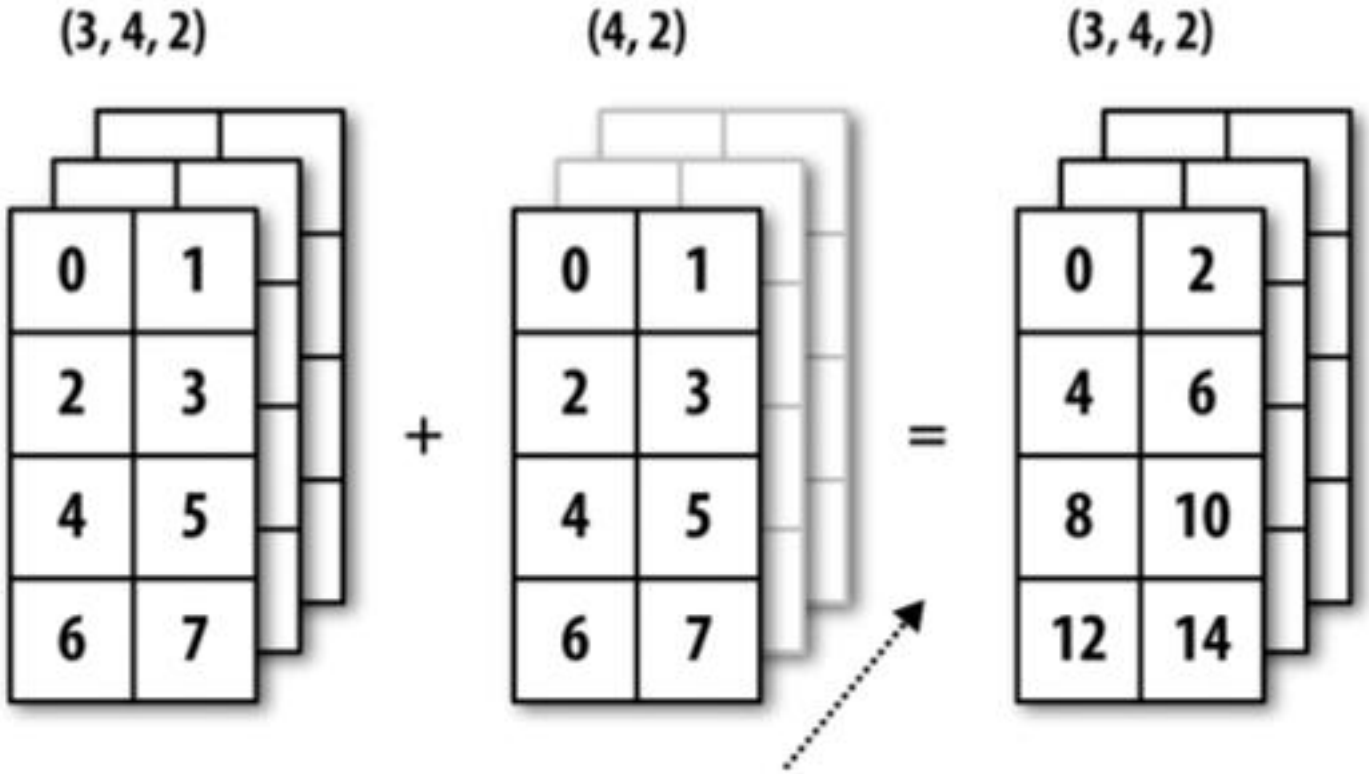
Broadcasting

```
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 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 05

- What is the output of the code fragment

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

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]]
```

THANK YOU
for YOUR ATTENTION