University of Science - VNU-HCM
Faculty of Information Science
Department of Computer Science

MTH083 - Advanced Programming for Artificial Intelligence

Slot 06-Numpy

Advisor:

Dr. Nguyễn Tiến Huy

Dr. Lê Thanh Tùng

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

```python
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

```python
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:

```
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:

```
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

```python
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]]
```

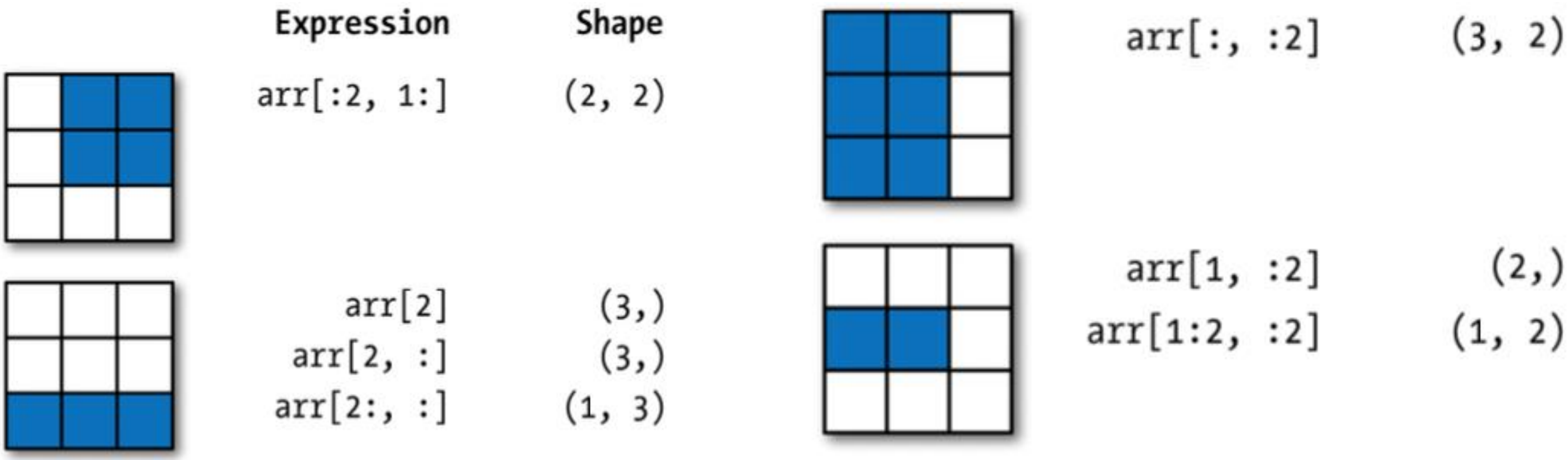
Indexing & Slicing

```python
# 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



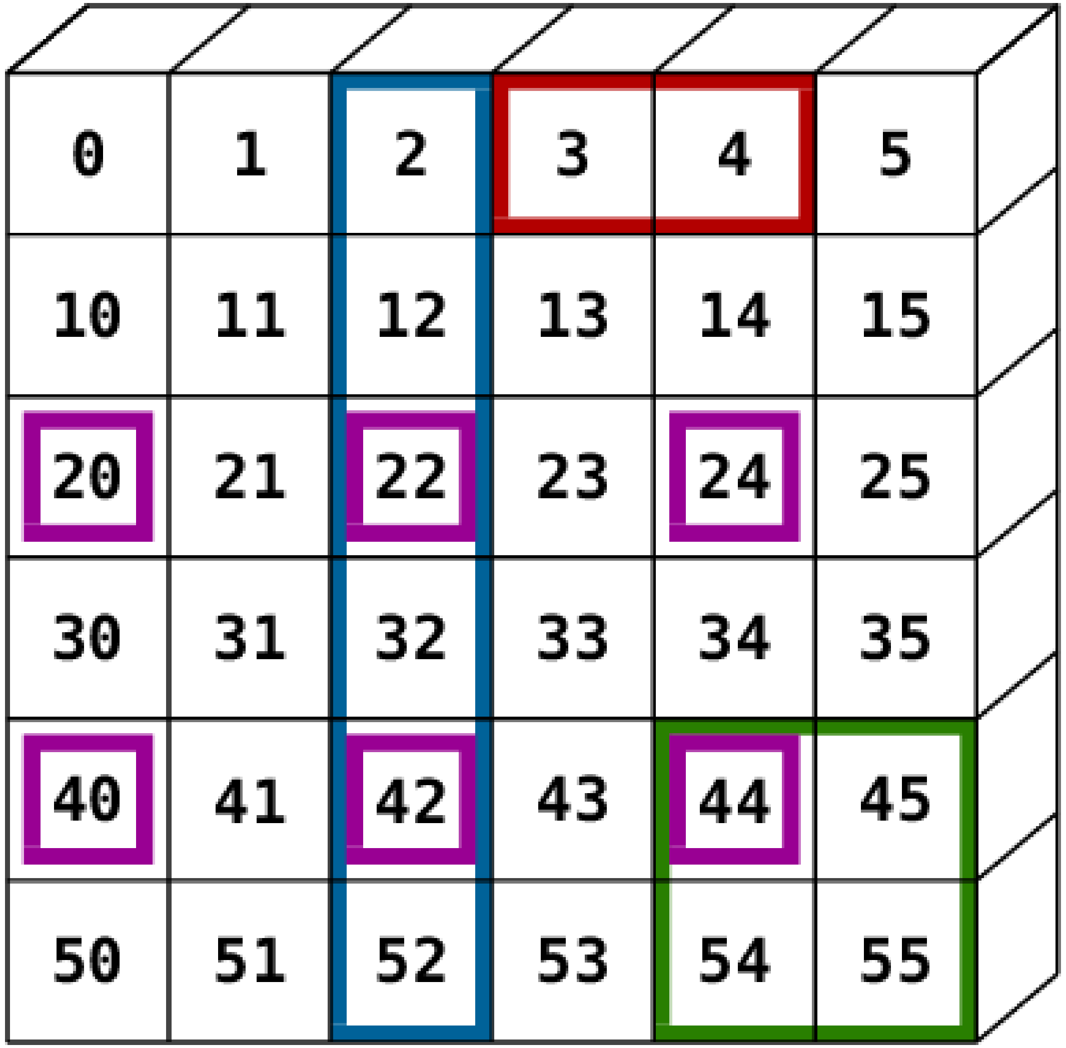
Indexing & Slicing

```
>>> 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

```python
# 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

```python
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'])
# [[-1.71144185  1.04257142  2.49841375]
#  [ 0.32656577  0.30792468  1.18419888]
#  [-0.55027386  0.80837473 -2.06744318]]
# Negative condition
print(data[~(names != 'Bob')])
# [[-1.65340362  0.20757075  0.97750883]
#  [-0.9188661  -1.23559978  0.13120124]]
```

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

```python
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

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

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

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

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

THANK YOU

for YOUR ATTENTION