

# Numpy及其应用

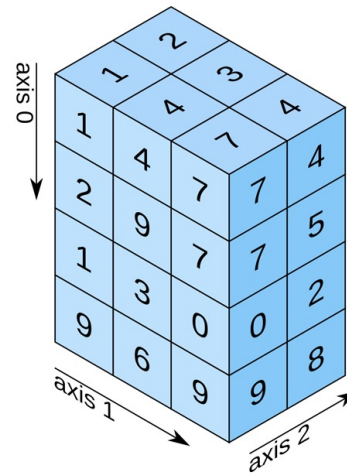
黄书剑



- **NumPy is the fundamental package for scientific computing in Python.**
  - **a multidimensional array object**
  - various derived objects (such as masked arrays and matrices)
  - an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

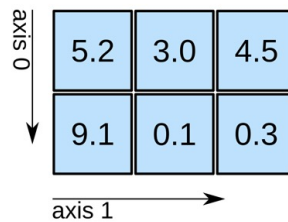
# multi-dimensional array

3D array



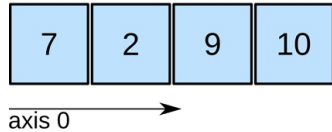
shape: (4, 3, 2)

2D array



shape: (2, 3)

1D array



shape: (4,)

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ 2 \\ \hline \end{array} - \begin{array}{|c|} \hline \text{ones} \\ \hline 1 \\ 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ 1 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ 2 \\ \hline \end{array} * \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ 4 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ 2 \\ \hline \end{array} / \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline \text{data} \\ \hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ \hline \end{array} .\text{max}() = 6$$

$$\begin{array}{|c|c|} \hline \text{data} \\ \hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ \hline \end{array} .\text{min}() = 1$$

$$\begin{array}{|c|c|} \hline \text{data} \\ \hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ \hline \end{array} .\text{sum}() = 21$$

[https://numpy.org/doc/stable/user/absolute\\_beginners.html](https://numpy.org/doc/stable/user/absolute_beginners.html)

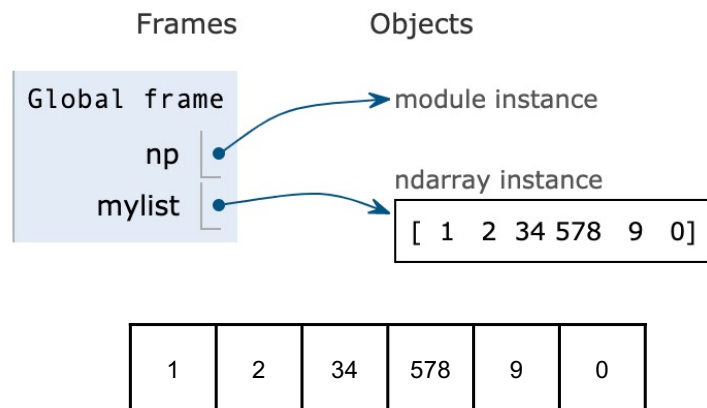


# NumPy ndarray v.s. Python List

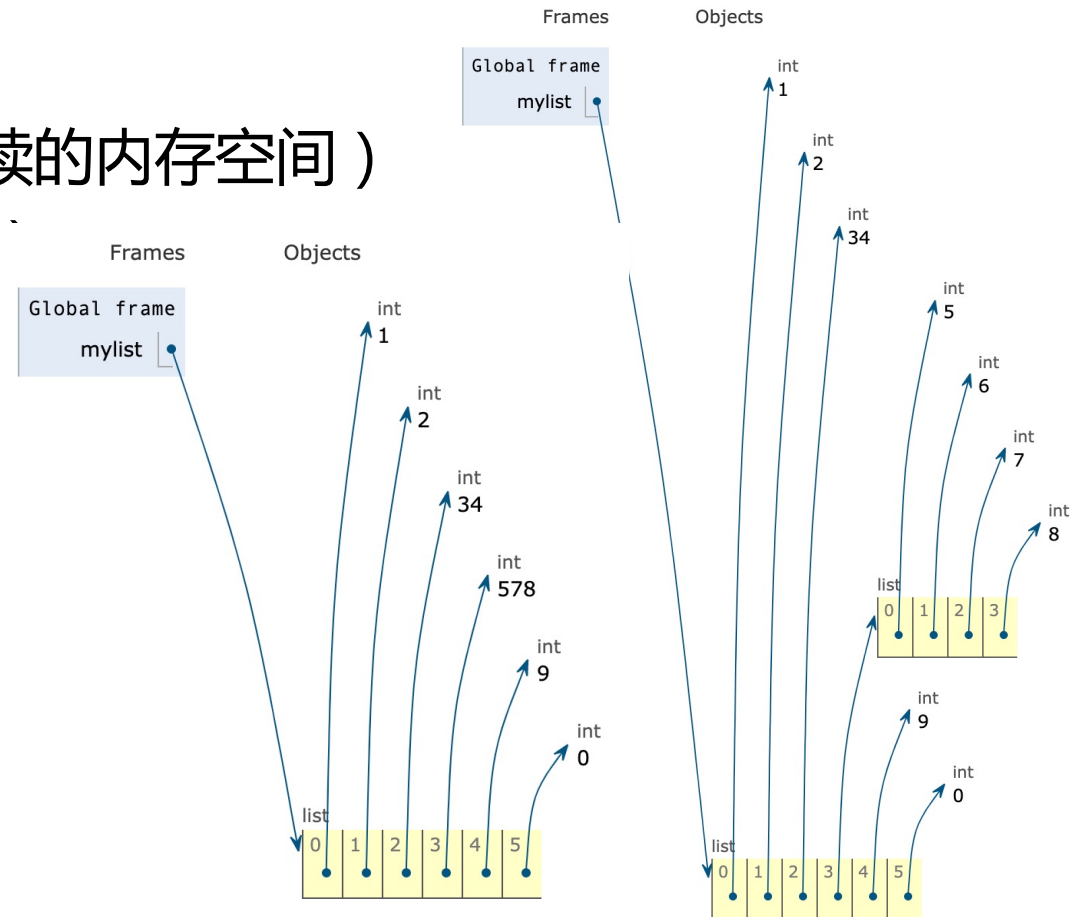
- **目标：更高的计算效率**
- **更高效的数据组织**
  - 同类型元素 + 固定长度
- **更高效的计算方式**
  - 向量化编程 v.s. 循环
- **实现和底层支持**
  - 基于C的实现
  - 利用Basic Linear Algebra Subprograms, BLAS
    - Intel MTK、 Open BLAS、 CUDA等

# NumPy Array v.s. Python List

- 数据组织上的差异
  - 更贴近C的数组实现（连续的内存空间）
  - 同质结构（元素类型相同）



```
mylist = np.array([1, 2, 34, 578, 9, 0])
```

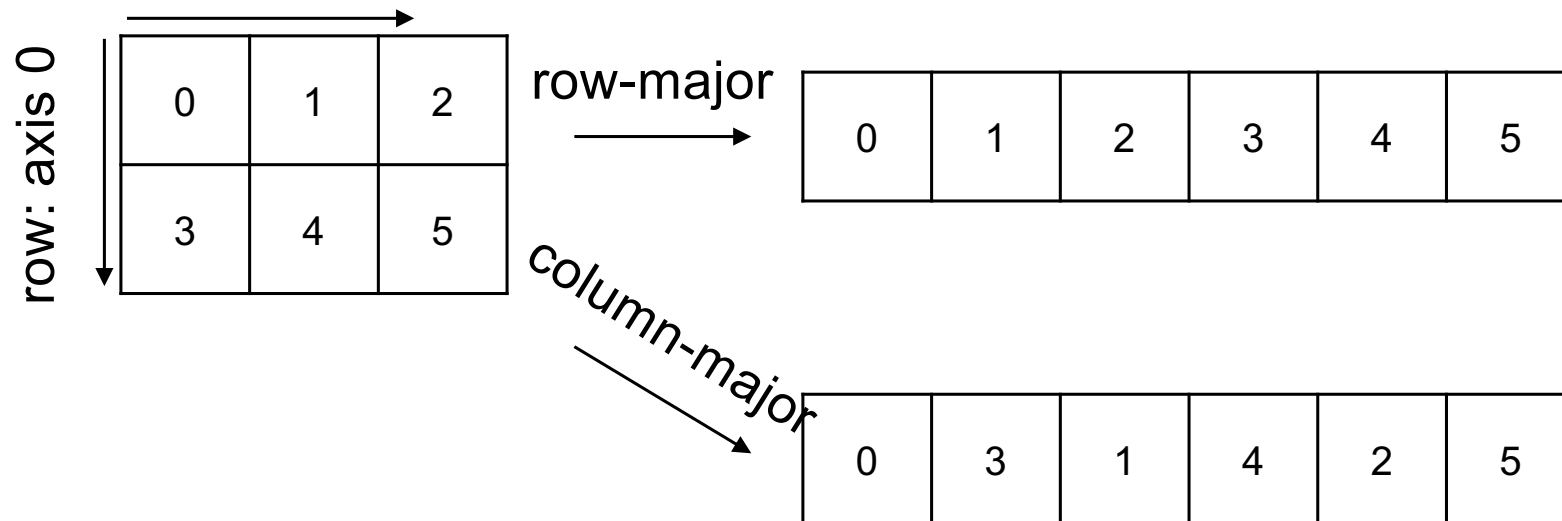




# 数组的存储表示

- 行主序 v.s. 列主序

col: axis 1



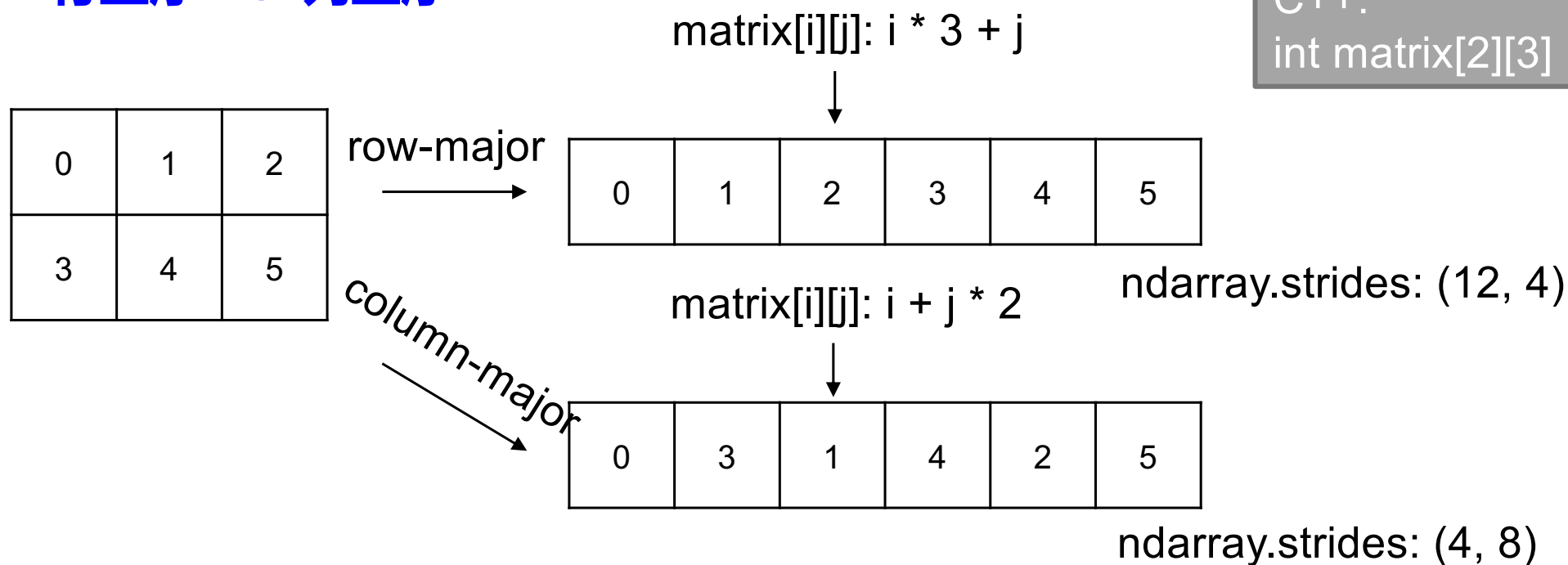
```
C++:  
int matrix[2][3]
```

行主序是C等语言的格式，列主序是Fortran等语言的格式 6



# 数组的存储表示

- 行主序 v.s. 列主序



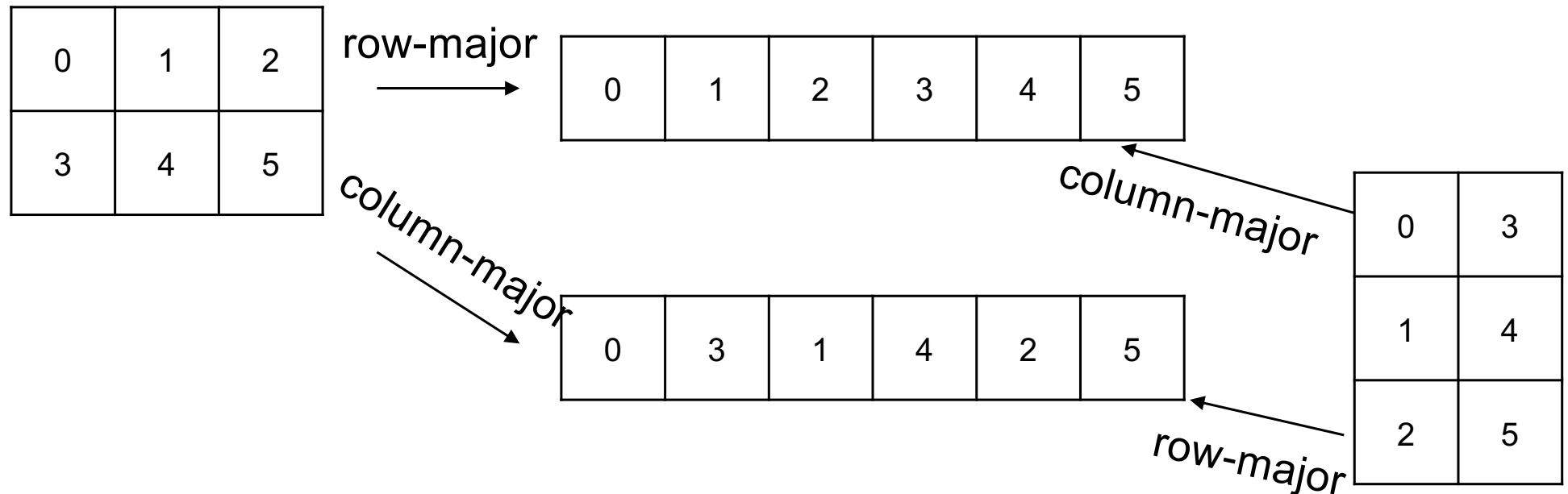
strides: 表示高维数组每一维对应的下标每次偏移时，在一维数组的存储中发生的偏移量(此处假设元素为int32，4个字节)。



# 数组的存储表示

- 行主序 v.s. 列主序

```
C++:  
int matrix[2][3]
```



不用改变数据表示，仅需要改变访问方式，即可完成转置！





## numpy中的转置

`import numpy as np` #默认引用和别名，后续大多省略

```
def print_array(arr):  
    print(arr.dtype)  
    print(arr)  
    print(arr.strides)  
    print()  
myarray = np.array([[1,2,3],[4,5,6]])  
print_array(myarray)
```

```
myarrayT = myarray.T  
print_array(myarrayT)
```

```
myarray2 = np.array([[1,4],[2,5],[3,6]])  
print_array(myarray2)
```

同样的数组，不一样的内部表示！

# 数组的存储表示



C++:  
`int matrix[2][3]`

0	1	2
3	4	5

row-major  
→

0	1	2	3	4	5
---	---	---	---	---	---

row-major  
↘

0	1
2	3
4	5

同样的内部表示，不一样的外部形式！



## numpy中改变数组的形状

```
data = np.arange(1, 7)
print_array(data)

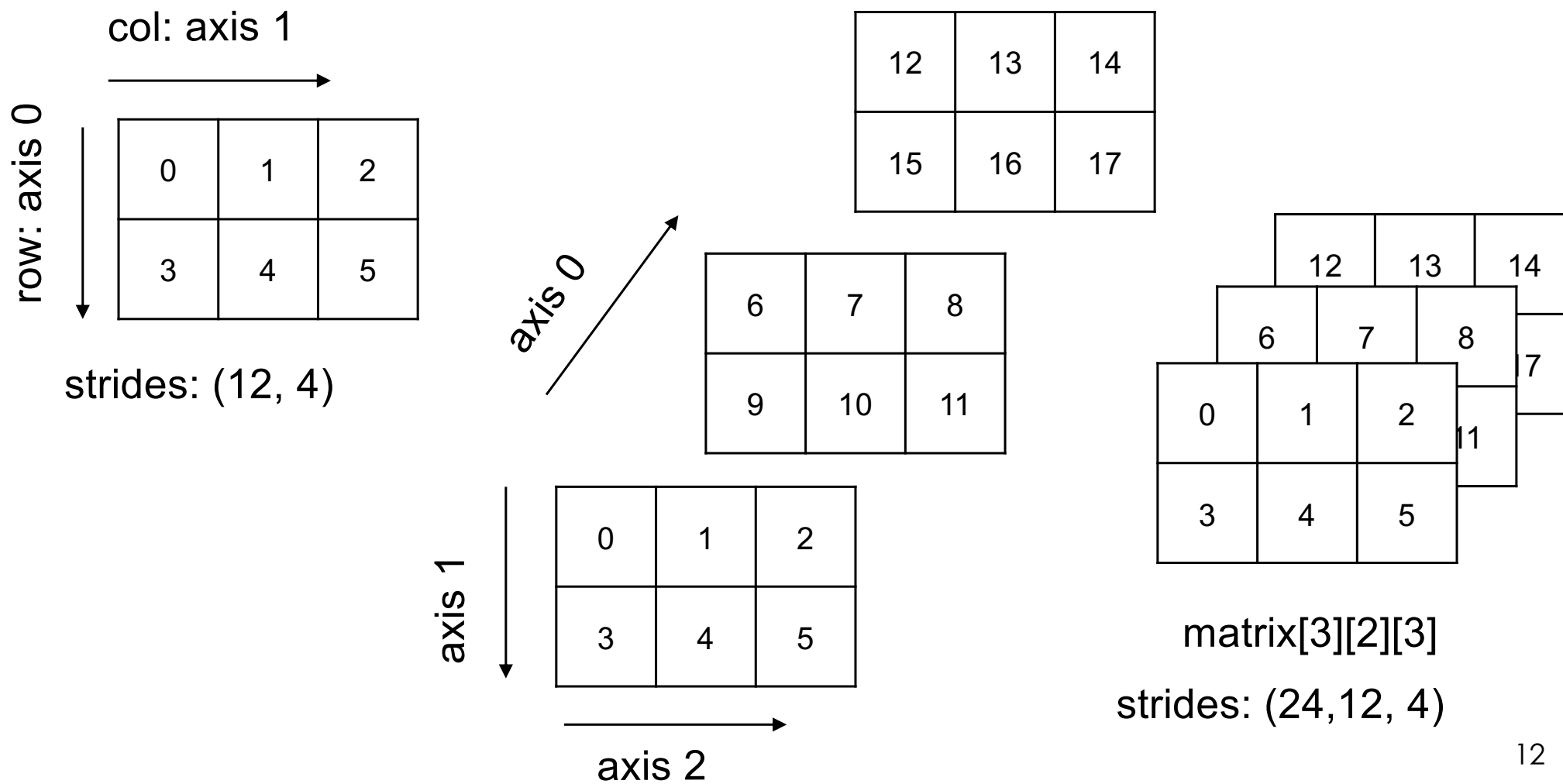
data1 = np.arange(1, 7).reshape((3,2))
print_array(data1)

data2 = np.arange(1, 7).reshape((2,3))
print_array(data2)
```

同样的数组，不一样的内部表示！  
同样的内部表示，不一样的外部形式！

抽象和封装：数据内部表示 v.s. 数据的外部使用

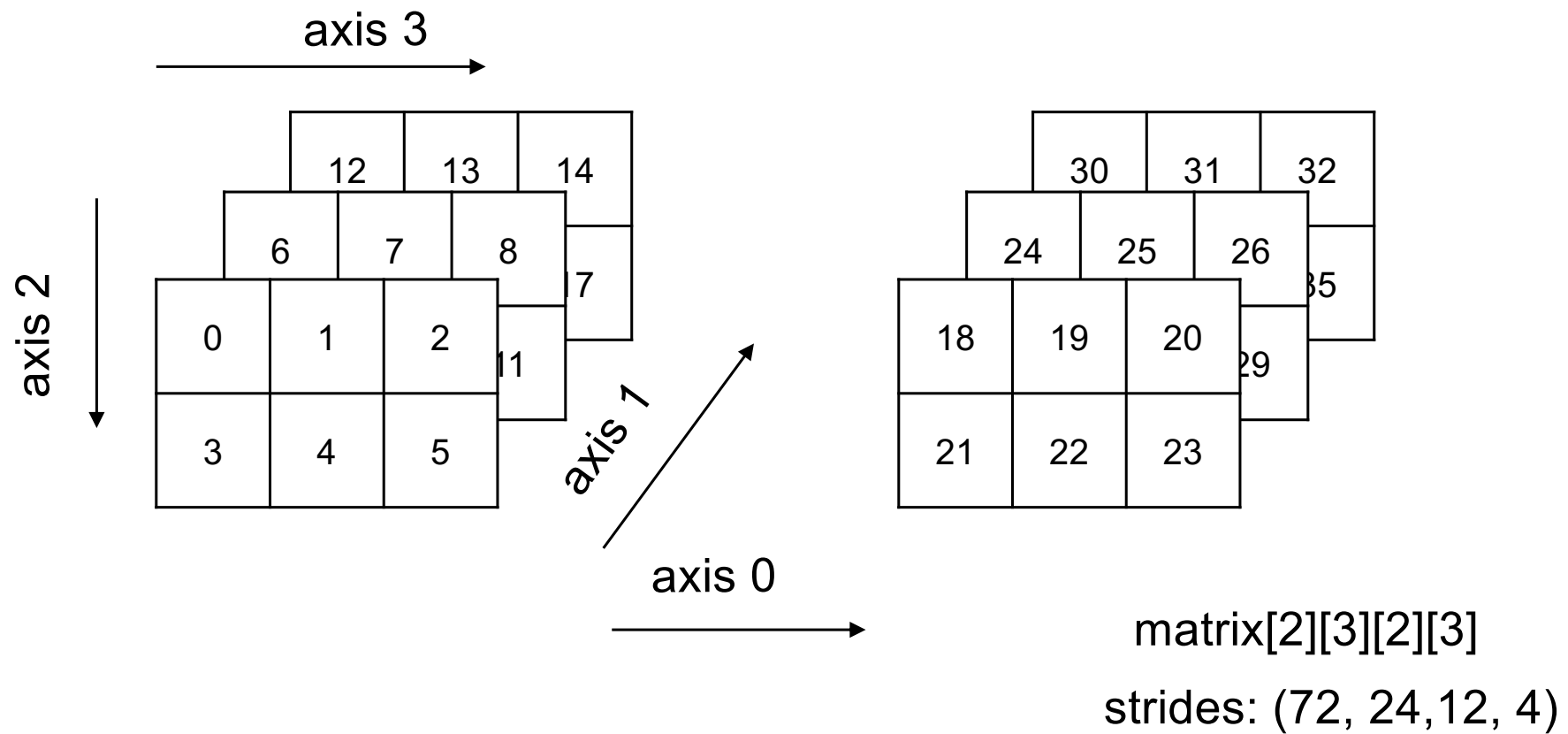
# 高维数组



# 高维数组



Tensor





```
data = np.arange(1, 19)
print_array(data)

data1 = np.arange(1, 19).reshape((3,2,3))
print_array(data1)

data2 = np.arange(1, 19).reshape((2,3,3))
print_array(data2)
```

试着自己实现一个n维矩阵类，创建任意维度的矩阵，提供元素访问操作、转置操作、reshape操作等功能。



# ndarray的基本属性

- **dtype : 元素类型**
- **ndim : 维数**
- **shape : 数组形状**
  - 整数构成的元组，如：(2, 3), (1, 4)等
  - 向量的形状是元素仅有1个的元组，须加逗号区分，如: (3, )
- **size : 所有元素个数**
- **itemsize : 单个元素大小 ( bytes )**
- **strides : 访问规则**
- **.....**

## Attributes

-----

T : ndarray

Transpose of the array.

data : buffer

The array's elements, in memory.

dtype : dtype object

Describes the format of the elements in the array.

size : int

Number of elements in the array.

itemsize : int

The memory use of each array element in bytes.

ndim : int

The array's number of dimensions.

shape : tuple of ints

Shape of the array.

strides : tuple of ints

The step-size required to move from one element to the next in memory. For example, a contiguous ``(3, 4)`` array of type ``int16`` in C-order has strides ``(8, 2)``. This implies that to move from element to element in memory requires jumps of 2 bytes. To move from row-to-row, one needs to jump 8 bytes at a time ( $2 * 4$ ).

`help(np.ndarray)`  
`np.ndarray?`



## NDARRAY的创建



# ndarray的创建

- 从已有列表创建
  - 使用np.array函数

查阅 `np.array?`

```
array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)
```

```
In [62]: np.array([1, 2, 3, 4])
```

```
Out[62]: array([1, 2, 3, 4])
```

```
In [63]: a1 = np.array([1, 2, 3, 4])
```

```
Out[63]: a2 = np.array(a1, order = 'F', ndmin = 2)
```



# ndarray的创建

- 从已有列表创建
- 数值自动填充
  - zeros, ones, empty, full ...
  - zeros\_like, ones\_like, empty\_like, full\_like ...
  - identity, eye
  - diag, arrange
  - linspace, logspace, meshgrid
  - fromfunction, fromfile
  - np.random.rand



```
In [58]: np.zeros(2)
Out[58]: array([0., 0.])
```

```
In [59]: np.ones(3)
Out[59]: array([1., 1., 1.])
```

```
In [60]: np.diag(range(1,5))
Out[60]:
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```
In [69]: np.arange(1,6,2)
Out[69]: array([1, 3, 5])
```

more examples on [examples\\_numpy.ipynb](#)

```
In [71]: np.linspace(0, 10, num=5)
Out[71]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```



# NumPy支持的数据类型

- **各种字长的5种基础数据类型**
  - booleans, integers, unsigned integers, floating point, complex
  - 例如: bool\_ , int8, int16, int32, float32, uint64, complex64 等
- **更精确指定数据类型 , 以获取存储和计算上的性能提升**
- NumPy也可以用于支持自定义类型 ( Structured arrays )
  - <https://numpy.org/doc/stable/user/basics.rec.html#structured-arrays>

索引、切片、视图

**从NDARRAY创建NDARRAY**

# 索引和切片

- **定位单个元素**
  - 下标 ( 正值、负值 )
- **选择多个元素**
  - 切片 ( ( start, end, step ) 序列开始、序列结尾 )
- **对于高维数组，须从0-ndim指明每个需要切片的维度**
- **每个维度的索引和切片操作相互独立**
  - `myarray[-3:,-3:]`

- 语法说明

```
newlist = listA[start : end : step]
```



- 结束位置元素不在结果列表中
- 步长用于跳过部分元素
- start、end 可以省略，分别表示从列表开始、直到列表结束
- step可以省略，表示默认步长为1



```
In [7]: myarray = np.arange(100).reshape(10, 10)
```

```
In [8]: myarray
```

```
Out[8]:
```

```
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],
       ...
```

```
In [9]: myarray[:,1]
```

```
In [10]: myarray[-3:,-3:]
```

```
In [11]: myarray[:,-3:]
```

```
In [12]: myarray[-3:]
```





## 切片和视图

- 切片操作得到的数据子集是原数组的一种展示方式（称为视图）
  - 视图仍然具有正常数组的行为、效率得到提升
  - 视图中的改动将影响原数组数据

```
In [19]: newarray =  
myarray[:,5,-3:]  
...: print_array(newarray)
```

```
Element Type:  int64  
Shape:      (2, 3)  
Strides:    (400, 8)  
[[ 7  8  9]  
 [57 58 59]]
```

```
In [20]: newarray2 =  
np.array([[ 7, 8, 9], [57, 58,  
59]], dtype="int32")  
...:
```

```
print_array(newarray2)
```

```
Element Type:  int32  
Shape:      (2, 3)  
Strides:    (12, 4)  
[[ 7  8  9]  
 [57 58 59]]
```

抽象和封装：数据内部表示 v.s. 数据的外部使用



```
In [21]: newarray3 = newarray[:, -2:]
```

创建切片的切片

```
In [22]: newarray3.fill(0)
```

将切片数组置零

```
In [23]: print(newarray3)
...: print(newarray)
...: print(myarray)
```

原数组也被置零，除非显式指明copy：  
`newarray3 = newarray[:, -2:].copy()`

试着自己实现一个三维矩阵类，让它能进行多次slicing操作，操作结果是原矩阵的一个视图。

```
[[0 0]
 [0 0]]
[[ 7  0  0]
 [57  0  0]]
[[ 0  1  2  3  4  5  6  7  0  0]
 [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]
 [50 51 52 53 54 55 56 57  0  0]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```



# fancy indexing & boolean indexing

- 花式索引

- 使用一个索引序列从ndarray中选择元素，并创建新数组

```
In [24]: myarray = np.arange(0, 100, 10)
...: indices = [1, 5, -1]
...: newarray = myarray[indices]
```

- 布尔索引

- 使用一个bool序列进行元素选择，并创建新数组

```
In [25]: myarray = np.arange(8)
...: b = [False, True, False, False, False, False, True, False]
...: myarray[b]
Out[25]: array([1, 6])
```

## 调整数组形状和值



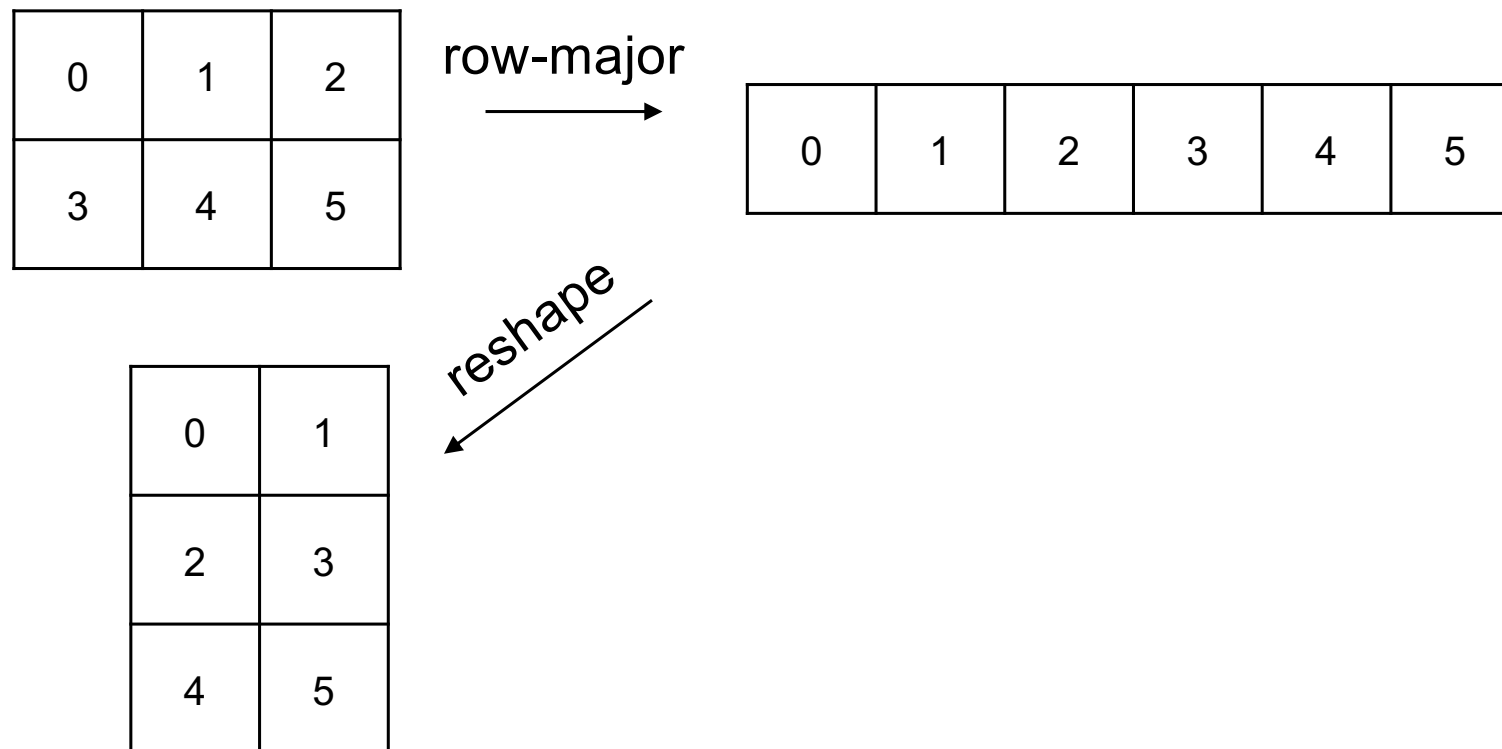
## 改变数组维度

- 基于底层表示支持上层的高效实现（一般返回视图）
  - `np.reshape/np.ndarray.reshape`
  - `np.ravel/np.ndarry.ravel`（折叠为一维数组）
  - `np.squeeze`（删除长度为1的维度）
  - `np.expand_dims`和`np.newaxis`（增加新的维度）
  - `np.transpose/np.ndarray.transpose/np.ndarray.T`



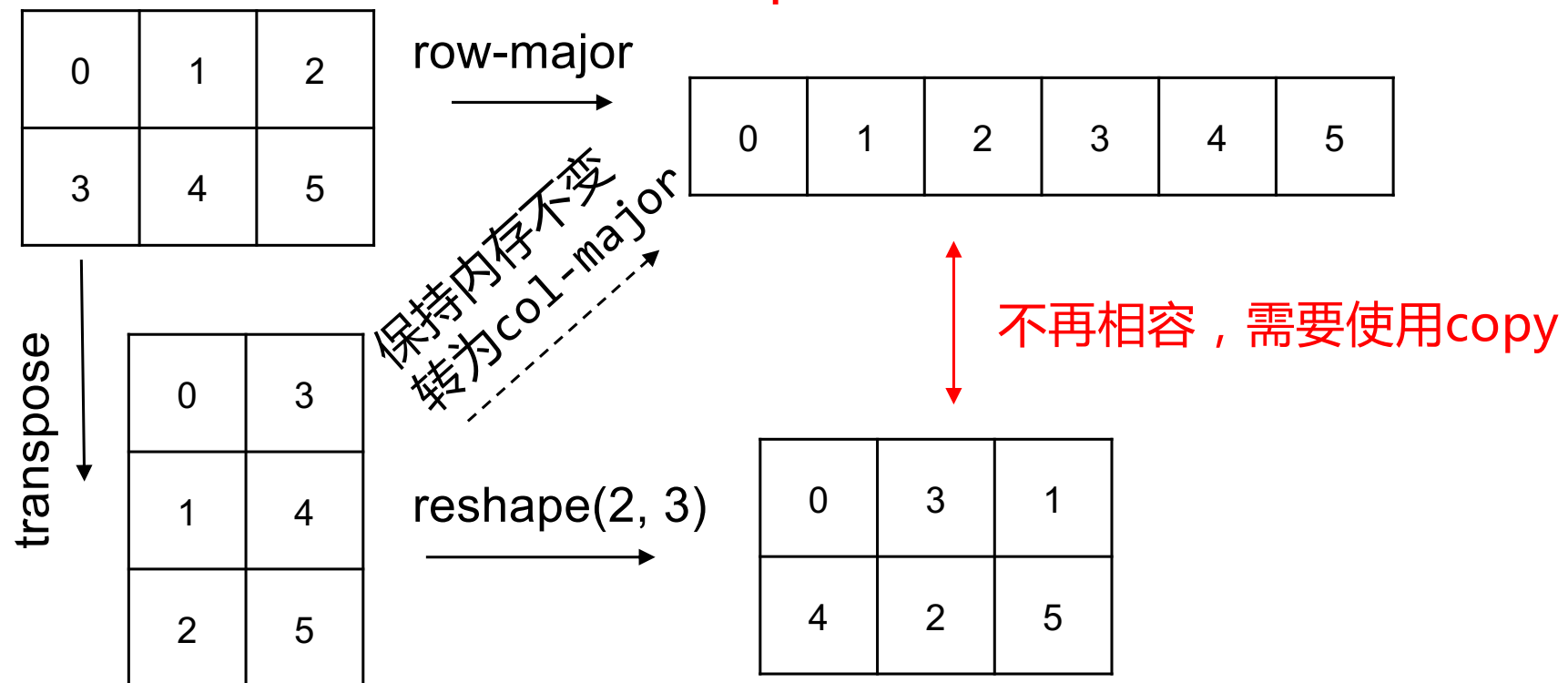
## 改变形状与内存布局

- **reshape默认按照行主序进行（C style）**
  - 对于以行主序存储的数组来说，不需要改变内存布局



## 改变形状与内存布局

- **reshape默认按照行主序进行 ( C style )**
  - 对于以行主序存储的数组来说，不需要改变内存布局
  - 如果转置后，再进行reshape，则难以使用原内存布局





## 改变形状与内存布局

- **reshape默认按照逻辑上的行主序进行 ( C style )**
  - 对于以行主序存储的数组来说，不需要改变内存布局
- **reshape时可以指定顺序 ( C style or F style or Automatic )**
  - A参数根据内存布局自动选择，此时reshape可能有不同结果

```
In [122]: x = np.arange(6).reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

```
In [124]: x.reshape(3, 2, order = 'F')
Out[124]:
array([[0, 4],
       [3, 2],
       [1, 5]])
```

```
In [125]: x.reshape(3, 2, order = 'A')
Out[125]:
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
In [126]: x.copy(order = 'F').reshape(3, 2,
order = 'A')
Out[126]:
array([[0, 4],
       [3, 2],
       [1, 5]])
```





## 改变数组大小、形状、内容

- 一般将会按照要求创建新的数组
- 改变形状
  - np.resize
  - np.ndarray.flatten
- 改变元素顺序
  - np.rot90, np.fliplr, np.flipud, np.sort
- 堆叠
  - np.hstack, np.vstack, np.dstack, np.concatenate
- 修改单个元素
  - np.append, np.insert, np.delete

## 矩阵的逐元素运算 (ELEMENT WISE OPERATIONS)

# 基本算术运算

- **算术运算定义为对应元素进行运算**
  - 对于大小相同的矩阵，运算结果与原矩阵形状相同

$$\begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} + \begin{array}{|c|} \hline \text{ones} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline \end{array} \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} - \begin{array}{|c|} \hline \text{ones} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline \end{array} \end{array}$$
$$\begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * \begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 4 \\ \hline \end{array} \end{array}$$
$$\begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} / \begin{array}{|c|} \hline \text{data} \\ \hline \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} \end{array}$$



## 循环 v.s. 向量化运算(vectorized operation)

- 通过python的循环依次访问每个元素并运算效率较低

- 创建访问迭代器
- 类型匹配和操作符运算函数调用

```
c = np.empty(a.size)
for i in range(a.size):
    c[i] += a[i] + b[i]
```



```
In [27]: size = 10000
...: a = np.random.rand(size)
...: b = np.random.rand(size)
...:
...: %timeit c = np.add(a, b)
...:
...: def myadd(a , b):
...:     c = np.empty(a.size)
...:     for i in range(a.size):
...:         c[i] += a[i] + b[i]
...:     return c
...:
...: %timeit myadd(a, b)
5.39  $\mu$ s  $\pm$  39.7 ns per loop (mean  $\pm$ 
std. dev. of 7 runs, 100000 loops each)
5.43 ms  $\pm$  22.5  $\mu$ s per loop (mean  $\pm$ 
std. dev. of 7 runs, 100 loops each)
```

```
In [28]: size = 10000
...: a = np.random.rand(size)
...: b = np.random.rand(size)
...:
...: %timeit c = np.dot(a, b)
...:
...: def mydot(a , b):
...:     c = 0
...:     for i in range(a.size):
...:         c += a[i] * b[i]
...:     return c
...:
...: %timeit mydot(a, b)
2.41  $\mu$ s  $\pm$  140 ns per loop (mean  $\pm$ 
std. dev. of 7 runs, 100000 loops
each)
4.13 ms  $\pm$  23.7  $\mu$ s per loop (mean  $\pm$ 
std. dev. of 7 runs, 100 loops each)
```



## 循环 v.s. 向量化运算(vectorized operation)

- 通过python的循环依次访问每个元素并运算效率较低

- 创建访问迭代器
- 类型匹配和操作符运算函数调用

```
c = np.empty(a.size)
for i in range(a.size):
    c[i] += a[i] + b[i]
```

- 将对矩阵中每个元素的运算 封装为 对矩阵整体的运算
  - 通过基于c的实现和底层优化提升计算效率
- 提供一系列封装好的计算函数 ( Universal functions, **ufunc** ) ,  
避免显式使用元素遍历循环



# Universal functions (ufuncs)

---

- **Math operations 数学运算**
  - add, subtract, multiply, matmul, divide, power, remainder, ...
- **Trigonometric functions 三角函数**
  - sin, cos, tan, sinh, cosh, tanh, arcsin, arccos, ...
- **Bit-twiddling functions 位运算**
  - bitwise\_add, bitwise\_or, bitwise\_xor, invert, left\_shift, ...
- **Comparison functions 比较运算**
  - greater, greater\_equal, less, equal, logical\_and, logical\_or, ...
- **Floating functions**
  - isfinite, isinf, isnan, fabs, fmod, floor, ceil, ...

<https://numpy.org/doc/stable/reference/ufuncs.html>



Calling ufuncs:

=====

`op(*x[, out], where=True, **kwargs)`

Apply `op` to the arguments `\*x` elementwise, broadcasting the arguments.

Parameters

-----

`*x` : array\_like

Input arrays.

`out` : ndarray, None, or tuple of ndarray and None, optional

指定输出位置

Alternate array object(s) in which to put the result; if provided, it must have a shape that the inputs broadcast to. A tuple of arrays (possible only as a keyword argument) must have length equal to the number of outputs; use `None` for uninitialized outputs to be allocated by the ufunc.

指定作用元素

`where` : array\_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default ``out=None``, locations within it where the condition is False will remain uninitialized.

`**kwargs`

For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`. 40

`class numpy.ufunc`  
`from np.ufunc?`





## 操作数匹配和广播 (Broadcasting)

- 当运算的两个操作数矩阵形状不同时，运算需要重新定义
  - NumPy通过广播机制自动扩展小矩阵，以匹配大矩阵
  - 将不同大小的矩阵转换为同样大小，便于进行统一运算

$$\begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * 1.6 = \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * \begin{array}{|c|} \hline 1.6 \\ \hline 1.6 \\ \hline \end{array} = \begin{array}{|c|} \hline 1.6 \\ \hline 3.2 \\ \hline \end{array}$$
  
$$\text{data} + \text{ones\_row} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline 6 & 7 \\ \hline \end{array}$$

<https://numpy.org/doc/stable/user/basics.broadcasting.html>



## 广播和向量化运算

- 广播后的计算可以采用通用的ufunc进行，以保证运算效率



```
In [26]: x = np.arange(10000)
....: %timeit x * 1.6
....: %timeit [i * 1.6 for i in x]
```

9.38  $\mu$ s  $\pm$  68 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

19.8 ms  $\pm$  109  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)



# Broadcasting广播

- **从右到左依次匹配两个矩阵各维度的长度：**
  - 当 **两个维度长度相等 或者 其中一个为1** 时，两者compatible
  - 当 其中一个矩阵维度不足 时，用**长度为1的新维度**进行补充
- **广播：**
  - 将长度为1的维度内容复制匹配另一矩阵对应长度

$$\begin{array}{c} \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * 1.6 = \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * \begin{array}{|c|} \hline 1.6 \\ \hline 1.6 \\ \hline \end{array} = \begin{array}{|c|} \hline 1.6 \\ \hline 3.2 \\ \hline \end{array} \quad \begin{array}{l} (1, ) \rightarrow (2, 1) \\ (2, 1) \end{array}$$
  
$$\begin{array}{c} \text{data} + \text{ones\_row} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline 6 & 7 \\ \hline \end{array} \quad \begin{array}{l} (1, 2) \rightarrow (3, 2) \\ (3, 2) \end{array}$$

<https://numpy.org/doc/stable/user/basics.broadcasting.html>

# Broadcasting广播



- 常见的广播是用于扩展标量、向量和低维矩阵

A	(2d array):	5 x 4
B	(1d array):	1
Result	(2d array):	5 x 4

A	(2d array):	5 x 4
B	(1d array):	4
Result	(2d array):	5 x 4

A	(3d array):	15 x 3 x 5
B	(2d array):	3 x 5
Result	(3d array):	15 x 3 x 5



# Broadcasting广播

- 常见的广播是用于扩展标量、向量和低维矩阵
  - 可能扩展的是中间的维度
  - 或扩展多个不连续维度

A	(3d array):	15 x 3 x 5
B	(2d array):	15 x 1 x 5
Result	(3d array):	15 x 3 x 5

A	(3d array):	15 x 3 x 5
B	(2d array):	3 x 1
Result	(3d array):	15 x 3 x 5



# Broadcasting广播

- 常见的广播是用于扩展标量、向量和低维矩阵
  - 可能扩展的是中间的维度
  - 或扩展多个不连续维度
  - 广播也可能同时扩展两个矩阵

A	(4d array):	8 x 1 x 6 x 1
B	(3d array):	7 x 1 x 5
Result	(4d array):	8 x 7 x 6 x 5



# Broadcasting广播

- 一些广播失败的例子

- ValueError: operands could not be broadcast together

A (1d array): 3

B (1d array): 4 # *trailing dimensions do not match*

A (2d array): 2 x 1

B (3d array): 8 x 4 x 3 # *second from last dimensions mismatched*

# Broadcasting广播



- 以下矩阵运算是否可以成功广播？

A (3d array): 4 x 1 x 5

B (2d array): 4 x 5

A (3d array): 4 x 3 x 5

B (2d array): 4 x 5





# Broadcasting广播

- **从右到左依次匹配两个矩阵各维度的长度：**
  - 当 **两个维度长度相等 或者 其中一个为1** 时，两者compatible
  - 当 其中一个矩阵维度不足 时，用**长度为1的新维度**进行补充
- **广播：**
  - 将长度为1的维度内容复制匹配另一矩阵对应长度
    - `strides = 0` **可以在ndarray基础上简单高效的实现！**

$$\begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * 1.6 = \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array} * \begin{array}{|c|} \hline 1.6 \\ \hline 1.6 \\ \hline \end{array} = \begin{array}{|c|} \hline 1.6 \\ \hline 3.2 \\ \hline \end{array} \quad \begin{array}{l} (1, ) \\ (2, 1) \end{array} \rightarrow (2, 1)$$
  
$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline 6 & 7 \\ \hline \end{array} \quad \begin{array}{l} (1, 2) \\ (3, 2) \end{array} \rightarrow (3, 2)$$

<https://numpy.org/doc/stable/user/basics.broadcasting.html>



```
In [3]: x = np.arange(4)
....: xx = x.reshape(4,1)
....: y = np.ones(5)
....: z =
np.ones((3,4))
```

```
In [4]: x + y
ValueError: operands could not be
broadcast together with shapes (4,)
(5,)
```

```
In [5]: xx + y
Out[5]:
array([[1., 1., 1., 1., 1.],
       [2., 2., 2., 2., 2.],
       [3., 3., 3., 3., 3.],
       [4., 4., 4., 4., 4.]])
```

```
In [6]: x + z
Out[6]:
array([[1., 2., 3., 4.],
       [1., 2., 3., 4.],
       [1., 2., 3., 4.]])
```



## 广播运算示例（比较运算）

```
In [2]: x = np.random.rand(10)
...: print(x)
...: print(x > 0.2)
...: print((-0.2 < x) * (x < 0.2))
[0.09757896 0.66117358 0.57061581 0.15890894 0.4839853  0.27529159
 0.5513947  0.23150319 0.92999985 0.46038401]
[False  True  True False  True  True  True  True  True  True]
[ True False False  True False False False False False False]
```



## ufunc的out参数

- 用于指定结果存储位置（有些时候可以提升计算效率）

```
In [32]: x = np.arange(1000000)
...: %time y = x * 2
...: %time x = x * 2
...: %time x *= 2
CPU times: user 2.33 ms, sys:
3.06 ms, total: 5.39 ms
Wall time: 4.74 ms
CPU times: user 2.17 ms, sys:
3.43 ms, total: 5.6 ms
Wall time: 5.64 ms
CPU times: user 858 µs, sys: 0
ns, total: 858 µs
Wall time: 862 µs
```

```
In [33]: y = np.random.rand(100,100)
...:
...: %time np.round(y, decimals = 4)
...: %time np.round(y, decimals = 4,
out = y)
CPU times: user 775 µs, sys: 970 µs, total:
1.74 ms
Wall time: 3.73 ms
CPU times: user 58 µs, sys: 6 µs, total:
64 µs
Wall time: 60.1 µs
```



## ufunc的where参数

- 用于指定参与运算的元素

```
In [1]: x = np.arange(5)
...: y = [True, True, False, True, False]
...:
...: #z = np.add(x, 5, where = y)
...: z = np.add(x, 5, where = (x%3==0))
...: print(x)
...: print(z)
```



## ufunc的相关功能

---

- **reduce**
  - 在指定的某个维度上应用ufunc
- **accumulate**
  - 在指定的某个维度上逐步应用ufunc



```
In [109]: x = np.arange(24).reshape(4, 6)
...: xacc = np.multiply.accumulate(x) #along axis 0
...: xred = np.multiply.reduce(x)
...: print(x)
...: print(xacc, xred)
```

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

```
[[    0     1     2     3     4     5]
 [    0     7    16    27    40    55]
 [    0    91   224   405   640   935]
 [    0  1729  4480  8505 14080 21505]]
```

```
[    0  1729  4480  8505 14080 21505]
```



## ufunc的相关功能

- **reduceat**

- 在指定维度的多个指定区间上应用reduce
- 传入数组indice，每个指定的区间为[indice[i], indice[i+1]]

```
In [110]: x = np.arange(8)
...: print(x)
...: np.add.reduceat(x, [0, 4, 1, 5])
#reduce for the slices (0, 4) (4, 1) (1, 5) (5, len(=8))
[0 1 2 3 4 5 6 7]
Out[110]: array([ 6,  4, 10, 18])

#reduce for slides (0, 4) adds 0, 1, 2, 3 together and get 6
#reduce for slides (4, 1) simply adds 4 together and get 4
#reduce for slides (5, ) adds till the end (5, 6, 7) and get 18
```





## ufunc的相关功能

- **reduceat**

- 在指定维度的多个指定区间上应用reduce
- 传入数组indices，每个指定的区间为[indices[i], indices[i+1]]
- 可以再对结果进行切片，只用成对的indices

```
In [111]: x = np.arange(8)
```

```
....: print(x)
```

```
....: np.add.reduceat(x, [0, 4, 1, 5])[:,2]
```

*#reduce for the slices (0, 4) (4, 1) (1, 5) (5, Len(=8)), and  
select the results for (0, 4) (1, 5)*

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

```
Out[111]: array([ 6, 10])
```



# ufunc的相关功能

- **at**

- 在参数indices指定位置应用ufunc

```
In [113]: x = np.array([1, 2, 3, 4])
...: np.add.at(x, [0, 1, 2, 2], 1) #add
1 to the index 0, 1, 2, 2 of array x
...: x
Out[113]: array([2, 3, 5, 4])
```

- **outer products**

- 对参数A、B矩阵中的每一对元素(a, b), 应用ufunc

```
In [114]: x = np.array([1, 2, 3, 4])
...: y = np.array([1, 2])
...: np.add.at(x, [0, 1], y) #
add y to the slice of x
...:
Out[114]: array([2, 4, 3, 4])
```

```
In [140]: x = np.arange(1,10)
...: print(np.multiply.outer(x, x))
[[ 1  2  3  4  5  6  7  8  9]
 [ 2  4  6  8 10 12 14 16 18]
 [ 3  6  9 12 15 18 21 24 27]
 [ 4  8 12 16 20 24 28 32 36]
 [ 5 10 15 20 25 30 35 40 45]
 [ 6 12 18 24 30 36 42 48 54]
 [ 7 14 21 28 35 42 49 56 63]
 [ 8 16 24 32 40 48 56 64 72]
 [ 9 18 27 36 45 54 63 72 81]]
```



# 自定义ufunc

- 自定义元素操作，并由numpy创建为对整个矩阵的操作
  - np.vectorize(myfunc)

```
In [42]: def myfunc(x, y):
...:     return x**2 + y**2
...:
...: myfunc_vectorized = np.vectorize(myfunc)
...:
...: x = np.random.rand(5, 5)
...: y = np.random.rand(5, 5)
...:
...: %timeit myfunc_vectorized(x, y)
...: %timeit x**2 + y**2
```

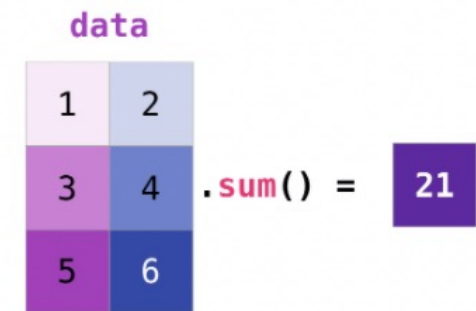
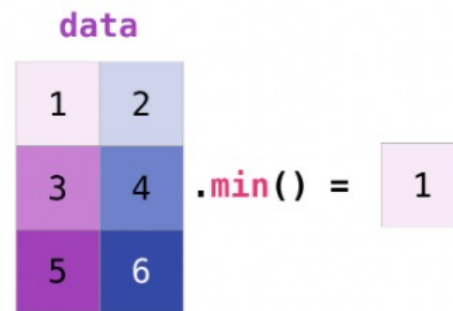
```
21.6 µs ± 199 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
2.05 µs ± 7.78 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

## 矩阵整体运算



# 聚合函数

- **用于对数组进行聚合计算**
  - np.mean, std, var, sum, prod, cumsum, cumprod,
  - min, max, argmin, argmax
  - all ( 所有元素不为零 , 返回True )
  - any ( 任一元素不为零 , 返回True )
- **默认对整个输入数组进行聚合 , 也可以用axis关键字参数指定聚合的轴**
  - data.max(axis=0)
  - data.max(axis=(0, 2))



指定axis表示沿着给定axis的方向进行聚合



```
In [89]: data = np.random.normal(size=(15,15))
...: np.mean(data) # data.mean()
Out[89]: 0.07822575388685046
```

```
In [90]: data = np.random.normal(size=(5, 10, 15))
...:
...: print(data.sum(axis=0).shape)
...: print(data.sum(axis=(0, 2)).shape)
```

```
(10, 15)
(10,)
```



# 条件计算函数

- **np.where**
  - 从两个数组中选取值
- **np.choose**
  - 根据给定的索引数组从数组列表中选取值
- **np.select**
  - 根据条件列表从数组列表中选取值
- **np.nonzero**
  - 返回非零元素的索引





```
In [98]: x = np.linspace(-4, 4, 9)          x: [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

```
In [99]: np.where(x < 0, x**2, x**3)
```

```
Out[99]: array([16.,  9.,  4.,  1.,  0.,  1.,  8., 27., 64.])
```

```
In [100]: np.select([x < -1, x < 2, x >= 2], [x**2, x**3 ,  
x**4])
```

```
Out[100]: array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,  81., 256.])
```

```
In [101]: np.choose([0, 0, 0, 1, 1, 1, 2, 2, 2], [x**2, x**3, x**4])
```

```
Out[101]: array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,  81., 256.])
```



# 集合运算

- 将ndarray作为一个无序的集合使用
  - np.unique 创建具有唯一值的新数组
  - np.in1d 查询一个数组中的元素是否包含在另一数组中
  - np.intersect1d, np.setdiff1d, np.union1d 数组的交、差、并



# 矩阵功能

- 矩阵和向量的运算

- np.dot, np.inner, np.cross, np.outer
- np.tensordot 沿着某个指定的轴进行点积
- np.kron, np.einsum

## 矩阵访问的效率问题



## 对行与列求和的差别：

```
In [30]: a = np.random.rand(5000,
5000)
...: %timeit a[0, :].sum()
...: %timeit a[:, 0].sum()
...: b = a.T
...: %timeit b[0, :].sum()
...: %timeit b[:, 0].sum()
...: c = np.array(a, copy = True,
order = 'F')
...: %timeit c[0, :].sum()
...: %timeit c[:, 0].sum()
...: d = np.array(a, copy = True,
order = 'C')
...: %timeit d[0, :].sum()
...: %timeit d[:, 0].sum()
```

3.77  $\mu\text{s}$   $\pm$  42.3 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 100000 loops each)  
35.7  $\mu\text{s}$   $\pm$  149 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 10000 loops each)  
35.4  $\mu\text{s}$   $\pm$  180 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 10000 loops each)  
3.76  $\mu\text{s}$   $\pm$  24.8 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 100000 loops each)  
35.3  $\mu\text{s}$   $\pm$  162 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 10000 loops each)  
3.75  $\mu\text{s}$   $\pm$  15.2 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 100000 loops each)  
3.76  $\mu\text{s}$   $\pm$  20.3 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 100000 loops each)  
35.3  $\mu\text{s}$   $\pm$  177 ns per loop (mean  $\pm$  std.  
dev. of 7 runs, 10000 loops each)



```
In [31]: a = np.arange(6).reshape(2,3)
....:
....: def print_nditer(b):
....:     #print_array(b)
....:     #print(b.ravel())
....:     print("Iters: ")
....:     for x in np.nditer(b):
....:         print(x, end = ' ')
....:     print("\n")
....:
....: print_nditer(a)
....: print_nditer(a.T)
....: print_nditer(a.T.copy(order = 'C'))
```

np.nditer是numpy中用于高效遍历矩阵元素的对象，遍历顺序可以由数据存储方式决定。

Iters:

0 1 2 3 4 5

Iters:

0 1 2 3 4 5

Iters:

0 3 1 4 2 5

ufunc可以采用更高效的方式完成遍历，但涉及更多底层细节（抽象）



## numpy应用示例（一）

- 计算两个给定序列的均方误差。

$$MeanSquareError = \frac{1}{n} \sum_{i=1}^n (Y\_prediction_i - Y_i)^2$$

```
In [48]: def mean_square_error(y_hat, y):  
        ...:     return np.average(np.square(y_hat - y), axis = 0)
```



## numpy应用示例（二）

- 查找序列中的局部极值点（比相邻值大/小的值）

- 假设是一维数组:

```
a = np.random.randint(0, 10, [10])
```

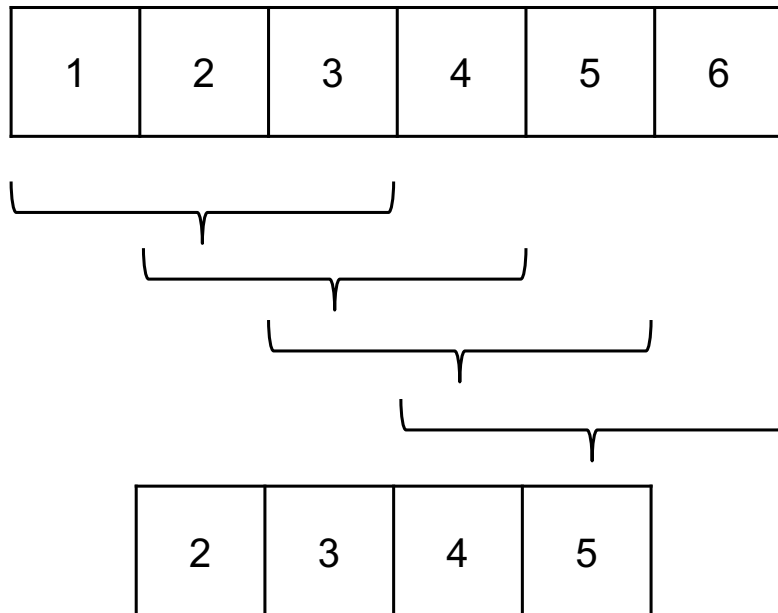
- 如何比较a与a两边的元素值？





## numpy应用示例（三）

- 计算数组中连续三个值的平均值



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

如何计算给定窗口大小w中的元素的平均值？



## numpy应用示例（四）

- 计算给定矩阵的平方根矩阵。其中，负数的平方根为其绝对值的平方根的相反数。

```
x = np.random.randint(-100, 101, (5,5))
```

```
np.sqrt(x)
```

- 组合现有ufunc，自定义ufunc？



```
In [46]: def method1(x):
...:     x = np.array(x, dtype = "float32")
...:     np.sqrt(x, where = x > 0, out = x)
...:     np.sqrt(-x, where = x < 0, out = x)
...:     return x
...:
...: def method2(x):
...:     mysqrt = np.vectorize(lambda x: np.sqrt(x) if x > 0 else np.sqrt(-x))
...:     return mysqrt(x)
```

```
In [47]: %timeit method1(x)
...: %timeit method2(x)
7.88  $\mu$ s  $\pm$  120 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)
44.6  $\mu$ s  $\pm$  174 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)
```



## numpy应用示例（五）

- **The Game of Life**
- **These rules, which compare the behavior of the automaton to real life, can be condensed into the following:**
  - Any live cell with two or three live neighbours survives.
  - Any dead cell with three live neighbours becomes a live cell.
  - All other live cells die in the next generation. Similarly, all other dead cells stay dead.
- **由随机初始的矩阵开始，输出每一代的生物存活情况**
  - 尝试发现生命的规律 ^\_^

[https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)



Still lifes	
Block	
Bee-hive	
Loaf	
Boat	
Tub	

Oscillators	
Blinker (period 2)	
Toad (period 2)	
Beacon (period 2)	

Oscillators	
Blinker (period 2)	
Toad (period 2)	
Beacon (period 2)	

- **NumPy**
  - 对高维数组的一种抽象：高效存储和计算
  - 数据创建，索引、切片和视图
  - 矩阵的逐元素运算
    - 向量化运算、ufunc、广播
  - 矩阵整体运算
    - 聚合函数、条件函数、集合函数、数学运算函数



- **其他教程阅读**

- [https://numpy.org/doc/stable/user/absolute\\_beginners.html](https://numpy.org/doc/stable/user/absolute_beginners.html)
- [https://www.numpy.org.cn/article/basics/understanding\\_numpy.html](https://www.numpy.org.cn/article/basics/understanding_numpy.html)

- **100 numpy exercises**

- <https://github.com/rougier/numpy-100>