

向量化计算

谢俊@浙江大学超算队

2024-07-06

Part-0 什么是向量化计算?



向量化计算是什么？

- Array Programming （今天第一部分內容）
- Automatic/Manual vectorization （今天第二部分內容）
- 其它： Image Tracing, Word Embedding （不讨论）



呼应前文

- 具体内容
这就来了

Code Optimization - Vectorization

What is vectorization?

- Scaler computation: $a = 2 \cdot a$
- Vector computation: $\vec{a} = 2 \cdot \vec{a}$
 - $[a, b, c, d] \Rightarrow [2a, 2b, 2c, 2d]$




Methods:

- High-level: vectorized computation graph
- Instruction-level: SIMD instructions

Enjoy your lab2~

Part-1 NumPy

NumPy 是什么?

- 使用 Python 进行**科学计算**的基础包 
- 低情商表述：一个来做**矩阵运算**的库 
- 高情商表述：通往**人工智能**的第一步 



GET STARTED



NumPy 学些啥?

- 学会用 NumPy 做**数据分析**处理
- 掌握**向量化**的思考和代码
- 理解一些**算法优化**的原理机制

到底什么是向量化?

GET STARTED



在开始之前

- 又是配环境
- 两个方法都行
- 法一： 直接安装Python + pip安装 Numpy
- 法二： 安装 Anaconda, 使用conda install进行安装



准备好了?

- 在你习惯的环境中输入 `import numpy as np`
 - 含义是，导入 `numpy` 这个包，并且给它起个名叫 `np`
 - 以后需要这个包的东西，只需要 `np.xxxx` 即可调用
 - 通常约定俗成就叫它 `np`
 - 没有提示就是运行正常



举个例子

- 像作业这种代码就是未经向量化的

```
A = [[1,2,3],[4,5,6],[7,8,9]]
B = [[3,2,1],[6,5,4],[9,8,7]]

matrix_sum = [[0,0,0],[0,0,0],[0,0,0]]
for i in range(3):
    for j in range(3):
        matrix_sum[i][j] = A[i][j] + B[i][j]
```

```
matrix_sum
```

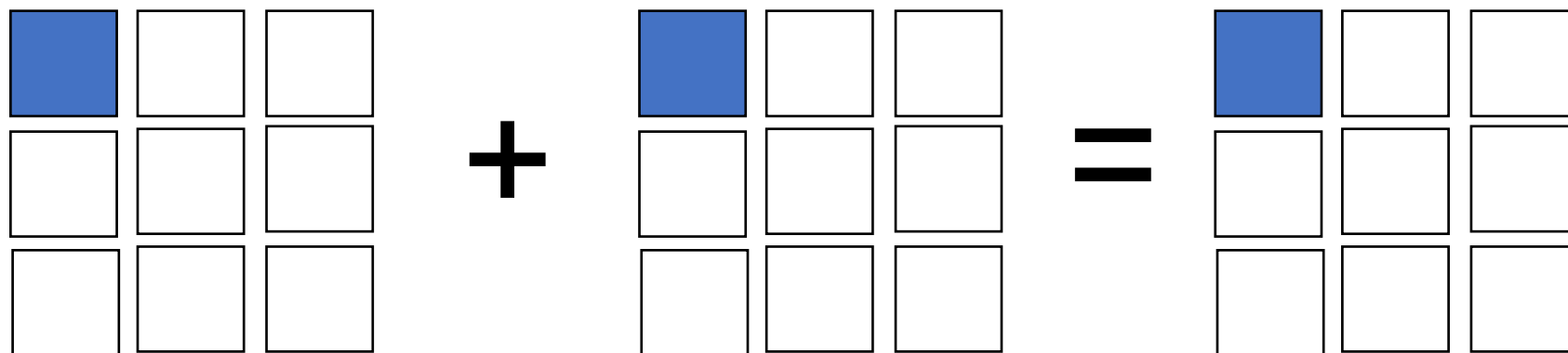
```
[4] ✓ 0.5s
```

```
... [[4, 4, 4], [10, 10, 10], [16, 16, 16]]
```

- 第一个显著特点是大量使用 (嵌套) for 循环
- 本质上是, 如果未经优化, 一次就只执行一条, 造成了极大的浪费



非常简单的优化思路

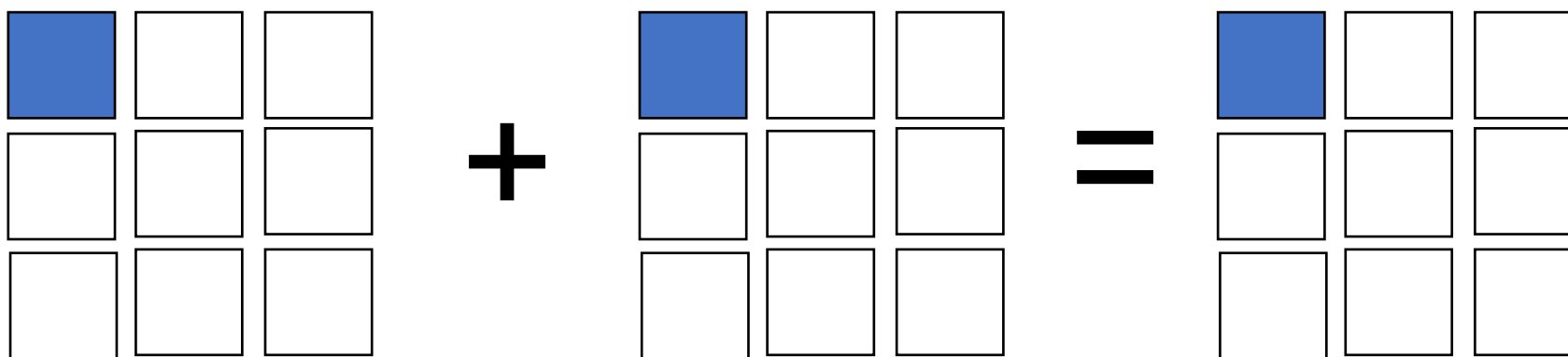

$$\begin{bmatrix} \text{blue} & & \\ & & \\ & & \end{bmatrix} + \begin{bmatrix} \text{blue} & & \\ & & \\ & & \end{bmatrix} = \begin{bmatrix} \text{blue} & & \\ & & \\ & & \end{bmatrix}$$

- 对于 3×3 的矩阵相加来说，答案的每个位置只和原来的两个矩阵相应坐标的值有关，和其他位置都无关
- 也就是说，一个位置的运算，和其他位置都没有关系
- 所以如果我有九个核心，那么一个核心处理一个位置就可以高效并行完成，甚至无需考虑原子操作，锁、进程间通信等等复杂琐碎的问题



非常简单的优化思路

- 对于 3×3 的矩阵相加来说，答案的每个位置只和原来的两个矩阵相应坐标的值有关，和其他位置都无关
- 如果我有九个核心，那么一个核心处理一个位置就可以高效并行完成，甚至无需考虑原子操作，锁、进程间通信等等复杂琐碎的问题





非常简单的优化思路

- 食堂做菜，请许多大厨，每个大厨各做一道菜





如果是乘法呢？

```
A = [[1,2,3],[4,5,6],[7,8,9]]  
B = [[3,2,1],[6,5,4],[9,8,7]]
```

[20] ✓ 0.4s

```
def matrix_mul(A,B):  
    matrix_product = [[0,0,0],[0,0,0],[0,0,0]]  
    for i in range(3):  
        for j in range(3):  
            matrix_product[i][j] = A[i][0] * B[0][j] + A[i][1] * B[1][j] + A[i][2] * B[2][j]  
    return matrix_product
```

[21] ✓ ✓ 0.4s

```
matrix_mul(A,B)
```

[22] ✓ 0.1s

```
... [[42, 36, 30], [96, 81, 66], [150, 126, 102]]
```



如果是乘法呢？

- 请几个大厨来同样可以提高效率
- 每一个值是两个一维的向量相乘，彼此独立
- 读的时候可能一个值会同时被多次读，但是没有写入，所以简单的分核心仍然是线程安全的。



如果是乘法呢？

- 但是从矩阵退化到向量的时候
- 如果是特别长的两个1维向量相乘呢？还能优化吗？





如果是乘法呢？

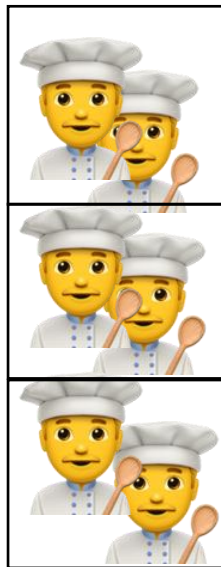
- 也可以提高效率，可以先让核心分头计算两个值的乘积
- 再将这些算好的加起来
- 当然也可以边算边加





如果是乘法呢？

- 也可以提高效率，可以先让核心分头计算两个值的乘积
- 再将这些算好的加起来
- 当然也可以边算边加





如果是乘法呢？

- 也可以提高效率，可以先让核心分头计算两个值的乘积
- 再将这些算好的加起来
- 当然也可以边算边加
- 暂不考虑如何优化使得效率最高
- 先理解可以同时工作提高效率，也就是**并行**





如果是乘法呢？

- 换句话说，请几个大厨来同样可以提高效率
- 只是需要考虑大厨间配合，不是每人独自做菜了





矩阵运算大抵皆如此

- 向量化核心思想：一次同时参与运算的不是一个值，
而是同时多个值一起算，即一个向量
- 多个值一起算，可以是逻辑上的，也可以是实际执行上的
- 很多时候，向量化是一种思维上的抽象

NumPy基础

GET STARTED



NumPy 基础

- 一个个介绍函数？
 - 时代变了！

ChatGPT on your desktop

Chat about email, screenshots, files, and anything on your screen.



NumPy 基础

- 创建一个矩阵
 - 将原有列表转为 NumPy 矩阵

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



a

1.	2.	3.
----	----	----

.dtype == np.float64
.shape == (3,)

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



a

1	2	3
4	5	6

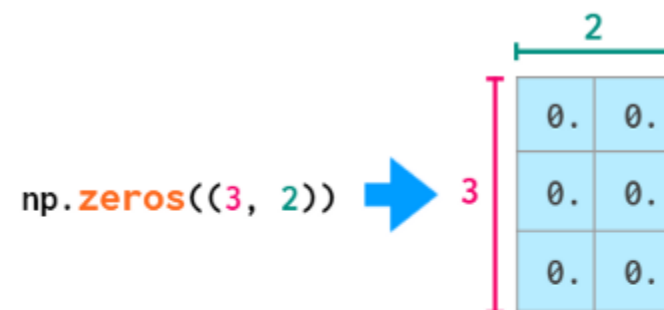
.dtype == np.int32
.shape == (2, 3)

↑
len(a) == a.shape[0]



NumPy 基础

- 创建一个矩阵
 - 创建一个全是0的矩阵，创建一个都是1的矩阵
 - 参数为矩阵的形状（也可以更高维）





NumPy 基础

- 创建一个矩阵
 - 创建都是自定义值的矩阵，创建单位矩阵，创建空矩阵

`np.full((3, 2), 7)`



7	7
7	7
7	7

`np.empty((3, 2))`



`np.eye(3, 3)`

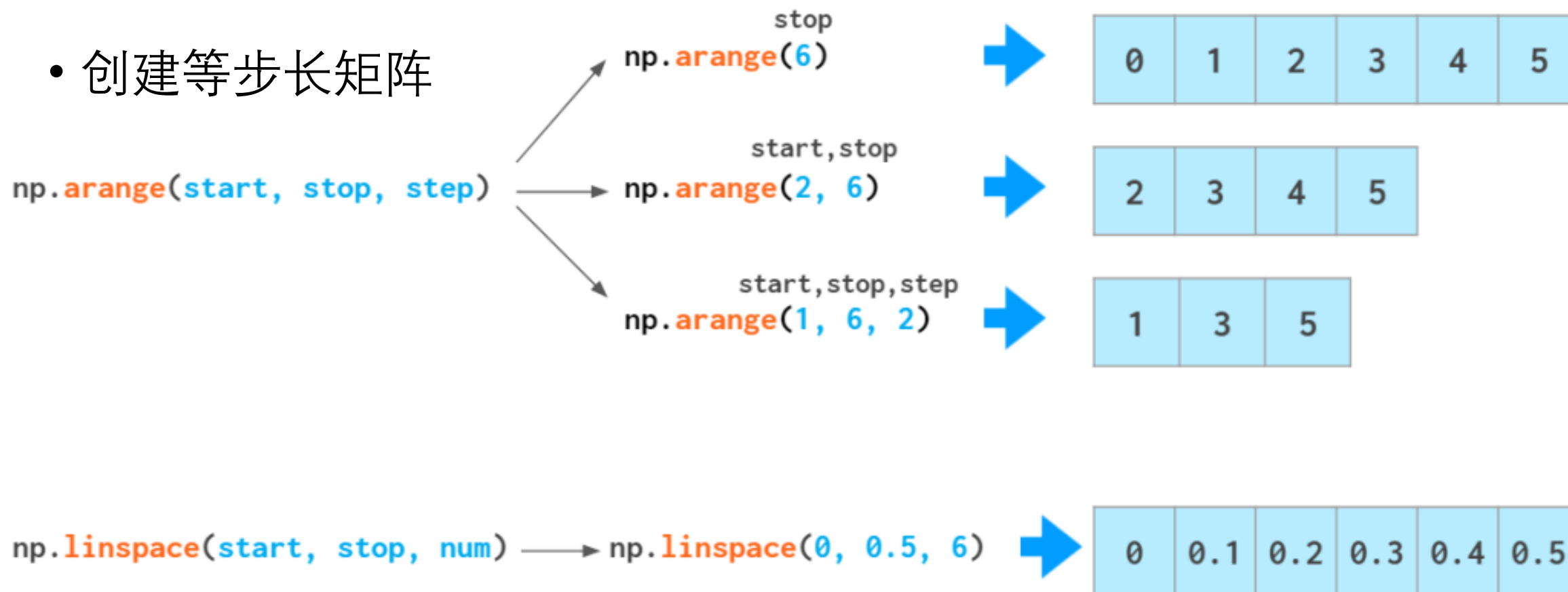
1.	0.	0.
0.	1.	0.
0.	0.	1.

`= np.eye(3)`



NumPy 基础

- 创建等步长矩阵





NumPy 基础

• 创建 随机数 矩阵

`np.random.randint(0, 10, (3, 2))`
uniform, $x \in [0, 10)$

4	8
3	7
5	2

Careful!
`random.randint(0, 10)`
 $x \in [0, 10]$

`np.random.rand(3, 2)`
uniform, $x \in [0, 1)$

0.7	0.5
0.3	0.8
0.4	0.1

`np.random.randn(3, 2)`
normal, $\mu=0, \sigma=1$

0.4	-1.2
1.4	-0.3
0.8	0.7

`np.random.uniform(1, 10, (3, 2))`
uniform, $x \in [1, 10)$

9.6	8.7
3.8	2.6
6.0	9.4

`np.random.normal(10, 2, (3, 2))`
normal, $\mu=10, \sigma=2$

8.7	12.3
10.5	10.8
14.3	11.2



NumPy 基础

- 创建

随机数

矩阵

这是新方法

更推荐这种

```
rng = np.random.default_rng()
```

```
rng.integers(0, 10, 3)
```

uniform, $x \in [0, 10)$



4	3	7
---	---	---



```
rng.integers(0, 10, 3, endpoint=True)
```

uniform, $x \in [0, 10]$

```
rng.random(3)
```

uniform, $x \in [0, 1)$



0.7	0.3	0.8
-----	-----	-----

```
rng.standard_normal(3)
```

normal, $\mu=0, \sigma=1$



0.4	-1.1	0.8
-----	------	-----

```
rng.uniform(1, 10, 3)
```

uniform, $x \in [1, 10)$



5.1	2.7	7.2
-----	-----	-----

```
rng.normal(5, 2, 3)
```

normal, $\mu=5, \sigma=2$



4.5	3.2	6.7
-----	-----	-----



NumPy 基础

- 更改矩阵的形状
- 使用 `reshape()`
- 也可以用 `shape` 查看形状
- 也可以用 `ndim` 查看维数

```
a = np.arange(9).reshape(3,3)
a
[44] ✓ 0.6s
... array([[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]])

a.ndim
[45] ✓ 0.4s
... 2

a.shape
[46] ✓ 0.4s
... (2, 3)
```



NumPy 基础

- 数据类型
- 当创建一个 NumPy 的 array 的时候，实际上创建了一个 ndarray 类型的对象，含义是 n-dimension array （n维数组）

```
▶ type(np.ones(3))  
[34] ✓ 0.5s  
... numpy.ndarray
```




NumPy 数据类型

- 而这个数组中存的数据也有自身的数据类型
- 回忆 Python 中的 list, 其可以装任何数据, 每个数据长度可以不一致, 如 `[1,2,"123",[4,5,6]]`
- 而 C 语言当中的数组, 其长度必须一致, 如 `int a[5];`
- **思考**: Python 的 list 在内存中是如何实现的?



答案是指针!

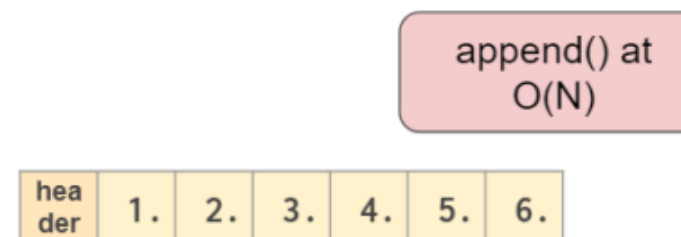
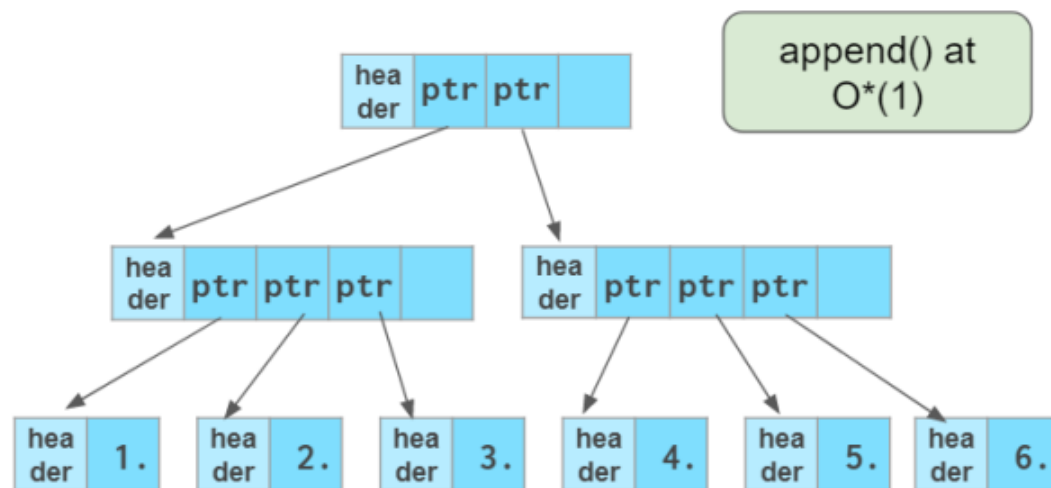
python list

vs

numpy array

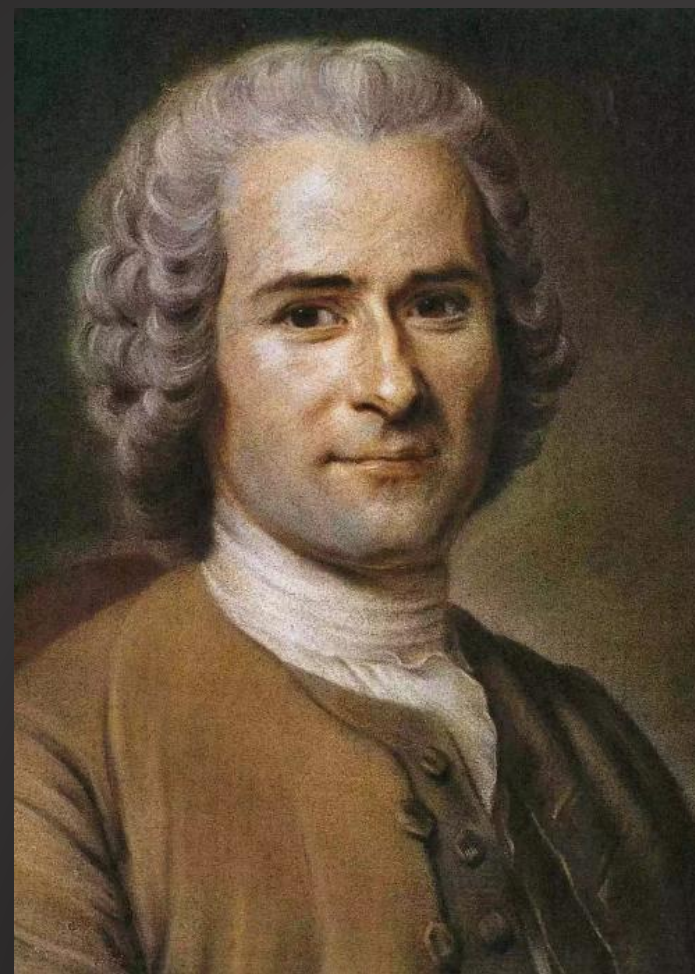
1.	2.	3.
4.	5.	6.

1.	2.	3.
4.	5.	6.



人生而自由， 却无往不在枷锁之中。

—— 卢梭





自由的代价

- Python 中的 list 可以装任何数据，原因是保存的实际上是指针，而指针可以指向任何数据
- 首先造成了**深浅拷贝**的问题（直接等号是一个数组的引用，使用`.copy`是浅拷贝，复制了原数组所有值的指针）。
- 对科学计算来说，更关键的是运算效率大大降低了，因为每次都要多一次寻址操作



自由的代价

- 思考：为什么两次寻址会更慢？



自由的代价

- **思考：**为什么两次寻址会更慢？
- 因为访问内存对于分秒必争的科学运算来说已经是很慢的了。
- 一次去内存找指针，一次按指针找真实的值
- 同时，如果数组按真实值连续放在内存中，缓存控制器会根据局部性定理一次性取多个值放到缓存中，又节省了时间。



NumPy 中的 dtype

- 从这个角度来讲，NumPy 和 C 更像一点
- 其数组是固定的数据类型
- 每种创建数组的函数都可以加一个 dtype 的参数
- 常见的 dtype 有 int32, float64, object 等等



NumPy 中的 dtype

- 需要知道的是类型转换使用 `.astype()`

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

```
✓ 0.8s
```

```
[40] a.dtype
```

```
✓ 0.5s
```

```
... dtype('int32')
```

```
[43] a = a.astype('int64')
```

```
a.dtype
```

```
✓ 0.7s
```

```
... dtype('int64')
```




NumPy 索引

- Python list 本身就已经提供了比较强大的索引功能。
- 需要注意的是维度多了以后，Python list 的索引和 C array 更像，而 Numpy 使用逗号来分割维数，更方便。

```
[47] a
... array([[1, 2, 3],
         [4, 5, 6]], dtype=int64)

[48] a[0,2]
... 3

[49] b = [[1,2,3],[4,5,6]]
      a[0][2]
... 3
```



NumPy 索引

- Start:Stop:Step
- NumPy 在此基础上还提供了 fancy indexing 的方式

```
a = np.arange(1, 6)
```

1	2	3	4	5
0	1	2	3	4

`a[1]`

2

`a[2:4]`

3	4
---	---

`a[-2:]`

4	5
---	---

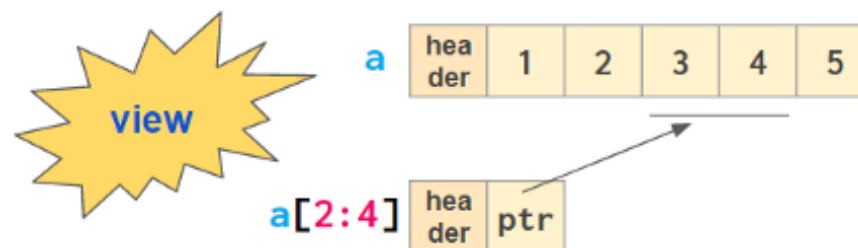
`a[::2]`

1	3	5
---	---	---

`a[[1,3,4]]`

2	4	5
---	---	---

"fancy indexing"



`a[2:4] = 0`



1	2	0	0	5
---	---	---	---	---



NumPy 索引

- NumPy 二维花式索引举例

```
▶ a
[50] ✓ 0.4s
... array([[1, 2, 3],
          [4, 5, 6]], dtype=int64)

a[[0, 0], [1, 2]]
[52] ✓ 0.4s
... array([2, 3], dtype=int64)
```



NumPy 索引

- 更多索引的例子

a

1	2	3	4
5	6	7	8
9	10	11	12

a[1,2]

1	2	3	4
5	6	7	8
9	10	11	12

a[1,:]

1	2	3	4
5	6	7	8
9	10	11	12

= **a[1]**

a[:,2]

1	2	3	4
5	6	7	8
9	10	11	12



a[:,1:3]

1	2	3	4
5	6	7	8
9	10	11	12

a[-2:,-2:]

1	2	3	4
5	6	7	8
9	10	11	12

a[:,2,1::2]

1	2	3	4
5	6	7	8
9	10	11	12



拷贝与视图

- 非常字面意思，视图不复制原有数据，而拷贝则需要
- NumPy 有些索引会返回拷贝(花式索引)，有些是视图

python list

```
a = [1, 2, 3]
b = a          # no copy
c = a[:]       # copy
d = a.copy()   # copy
```

vs

numpy array

```
a = np.array([1, 2, 3])
b = a          # no copy
c = a[:]       # no copy!!!
d = a.copy()   # copy
```



NumPy 布尔索引

- 类似于条件筛选
- `any()`
- `all()`

a	1	2	3	4	5	6	7	6	5	4	3	2	1
a > 5	False	False	False	False	False	True	True	True	False	False	False	False	False

`np.any(a > 5)`

True

`a[a > 5]`

6 7 6

`np.all(a > 5)`

False

`a[a > 5] = 0`

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

`a[(a >= 3) & (a <= 5)] = 0`

a	1	2	0	0	0	6	7	6	0	0	0	2	1
----------	---	---	---	---	---	---	---	---	---	---	---	---	---

& and
| or
^ xor
~ not



NumPy 条件筛选下标

a

1	2	3	4	5	6	7	6	5	4	3	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12

`np.where(a > 5)`

5	6	7
---	---	---

`= np.nonzero(a > 5)`

`a[a < 5] = 0; a[a >= 5] = 1`

a

0	0	0	0	1	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---

`= np.where(a >= 5, 1, 0)`

`a[a < 3] = 3; a[a > 5] = 5`

a

3	3	3	4	5	5	5	5	5	4	3	3	3
---	---	---	---	---	---	---	---	---	---	---	---	---

`= np.clip(a, 3, 5)`



NumPy 条件筛选下标

- 二维需要考虑轴的问题

$$b = \exists i, j \mid a_{ij} > 5 \quad \rightarrow \quad \text{np.any}(\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} > 5) \quad \rightarrow \quad \text{True}$$

$$b_j = \exists i \mid a_{ij} > 5 \quad \rightarrow \quad \text{np.any}(\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} > 5, \text{axis}=0) \quad \rightarrow \quad \begin{array}{|c|c|c|} \hline \text{False} & \text{False} & \text{True} \\ \hline \end{array}$$

$$b_i = \exists j \mid a_{ij} > 5 \quad \rightarrow \quad \text{np.any}(\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} > 5, \text{axis}=1) \quad \rightarrow \quad \begin{array}{|c|} \hline \text{False} \\ \hline \text{True} \\ \hline \end{array}$$



NumPy 运算

- 两个向量就当作两个数一样，自动向量化了，非常简单
- Element-wise

$$\begin{bmatrix} 1 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} - \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} -3 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} * \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 40 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} / \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.6 \end{bmatrix} \text{ np.float64}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} // \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 1 \end{bmatrix} \text{ np.int32}$$

$$\begin{bmatrix} 3 & 4 \end{bmatrix} ** \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 9 & 64 \end{bmatrix}$$



NumPy 运算

- 矩阵也一样

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 2 \\ \hline 3 & 5 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} - \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 2 \\ \hline 3 & 3 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 2 & 0 \\ \hline 0 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 0 \\ \hline 0 & 8 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} @ \begin{array}{|c|c|} \hline 2 & 0 \\ \hline 0 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 4 \\ \hline 6 & 8 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} / \begin{array}{|c|c|} \hline 2 & 1 \\ \hline 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0.5 & 2. \\ \hline 3. & 2. \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} ** \begin{array}{|c|c|} \hline 2 & 1 \\ \hline 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 16 \\ \hline \end{array}$$



NumPy 运算

- NumPy 有广播机制，当尺寸不匹配的时候可以将“小”的广播到“大”的尺寸上

<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2	+	<table border="1"><tr><td>3</td></tr></table>	3	=	<table border="1"><tr><td>4</td><td>5</td></tr></table>	4	5		<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2	*	<table border="1"><tr><td>3</td></tr></table>	3	=	<table border="1"><tr><td>3</td><td>6</td></tr></table>	3	6		
1	2																					
3																						
4	5																					
1	2																					
3																						
3	6																					
<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2	-	<table border="1"><tr><td>3</td></tr></table>	3	=	<table border="1"><tr><td>-2</td><td>-1</td></tr></table>	-2	-1		<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2	/	<table border="1"><tr><td>3</td></tr></table>	3	=	<table border="1"><tr><td>0.33</td><td>0.67</td></tr></table>	0.33	0.67	np.float64	
1	2																					
3																						
-2	-1																					
1	2																					
3																						
0.33	0.67																					
						<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2	//	<table border="1"><tr><td>2</td></tr></table>	2	=	<table border="1"><tr><td>0</td><td>1</td></tr></table>	0	1	np.int32						
1	2																					
2																						
0	1																					
						<table border="1"><tr><td>3</td><td>4</td></tr></table>	3	4	**	<table border="1"><tr><td>2</td></tr></table>	2	=	<table border="1"><tr><td>9</td><td>16</td></tr></table>	9	16							
3	4																					
2																						
9	16																					



NumPy 运算

- 广播机制
- 对矩阵也有效
- 突出一个自然
- 注意直接相乘不是点积
- 要使用@或 `.dot()` 或 `np.dot()`

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} / \begin{bmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix} = \begin{bmatrix} .1 & .2 & .3 \\ .4 & .5 & .7 \\ .8 & .9 & 1. \end{bmatrix}$$

normalization

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 3 \\ -4 & 0 & 6 \\ -7 & 0 & 9 \end{bmatrix}$$

multiplying several columns at once

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} / \begin{bmatrix} 3 & 3 & 3 \\ 6 & 6 & 6 \\ 9 & 9 & 9 \end{bmatrix} = \begin{bmatrix} .3 & .7 & 1. \\ .6 & .8 & 1. \\ .8 & .9 & 1. \end{bmatrix}$$

row-wise normalization

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

outer product



NumPy 运算

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

outer product

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} @ \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$


outer product

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} @ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \end{bmatrix}$$

inner (or dot) product



NumPy 运算

- 广播机制的 
- 广播的条件
 - 两个向量维度相同
 - Or 某个维度一个向量有，一个无
 - Or 某个维度一个向量有，一个有但为1

```
X=np.empty((5,3,4,1))  
Y=np.empty((3,1,1))
```



```
X=np.empty((5,3,4,1))  
Y=np.empty((1,3,1,1))
```



```
X=np.empty((5,3,4,1))  
Y=np.empty((5,3,4,1))
```



NumPy 运算

- 转置操作

a

1	2	3
4	5	6

(2, 3)

$a_{ij} \rightarrow a_{ji}$

a.T

1	4
2	5
3	6

(3, 2)

b

1	2	3
---	---	---

(1, 3)

$b_{ij} \rightarrow b_{ji}$

b.T

1
2
3

(3, 1)

c

1	2	3
---	---	---

(3,)

$c_i \rightarrow c_i$

c.T

1	2	3
---	---	---

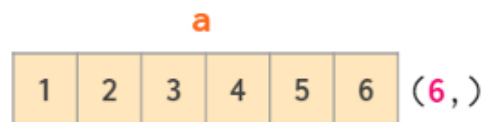
(3,)



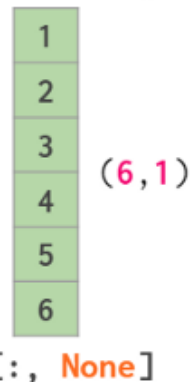


NumPy 运算

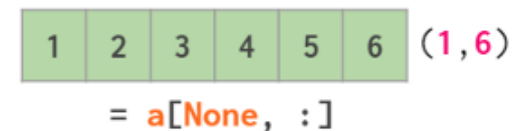
- 再看 reshape 操作
- -1 表示让 NumPy 自己去算
- 因为元素数量定了, 只要 n-1 维确定了, 最后一维度就定了



a.reshape(-1,1)



a.reshape(1,-1)



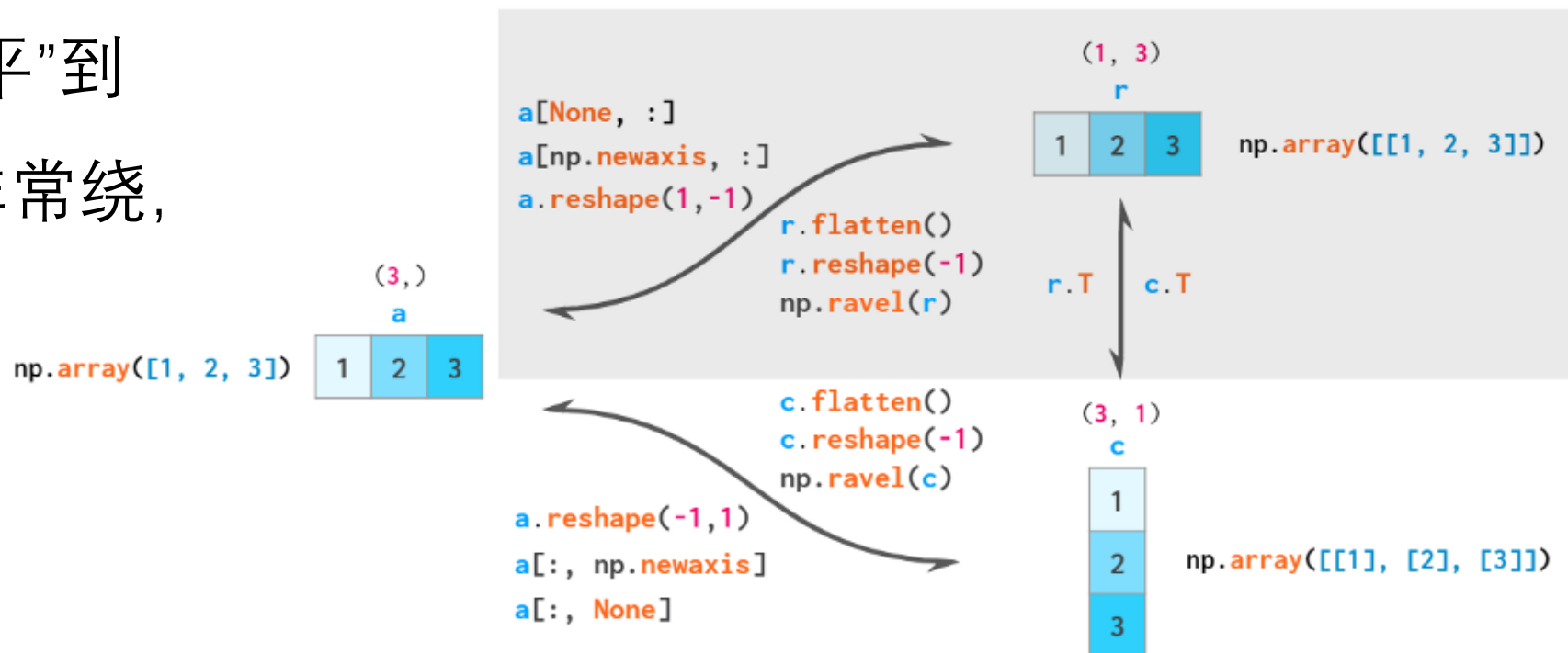
a.reshape(2,3)





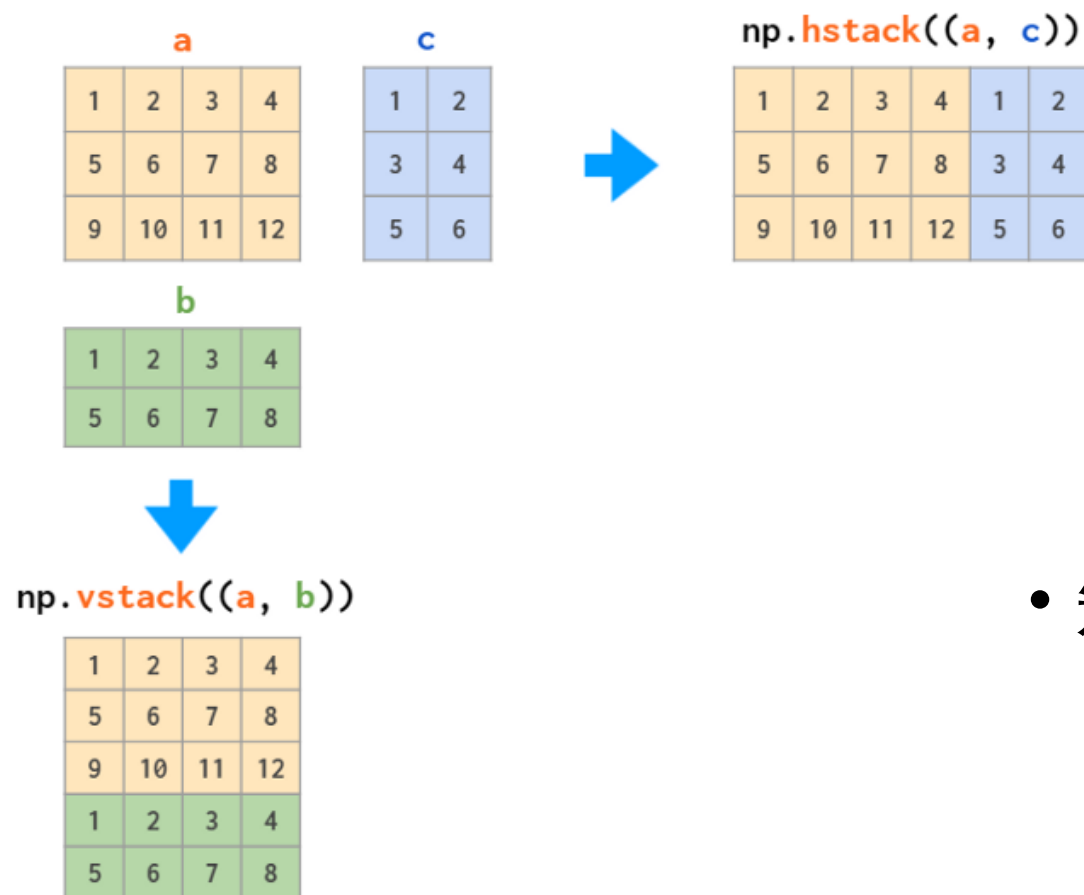
NumPy 运算

- 几种将矩阵“拍平”到一维的方法，非常绕，记不住没关系





NumPy 运算

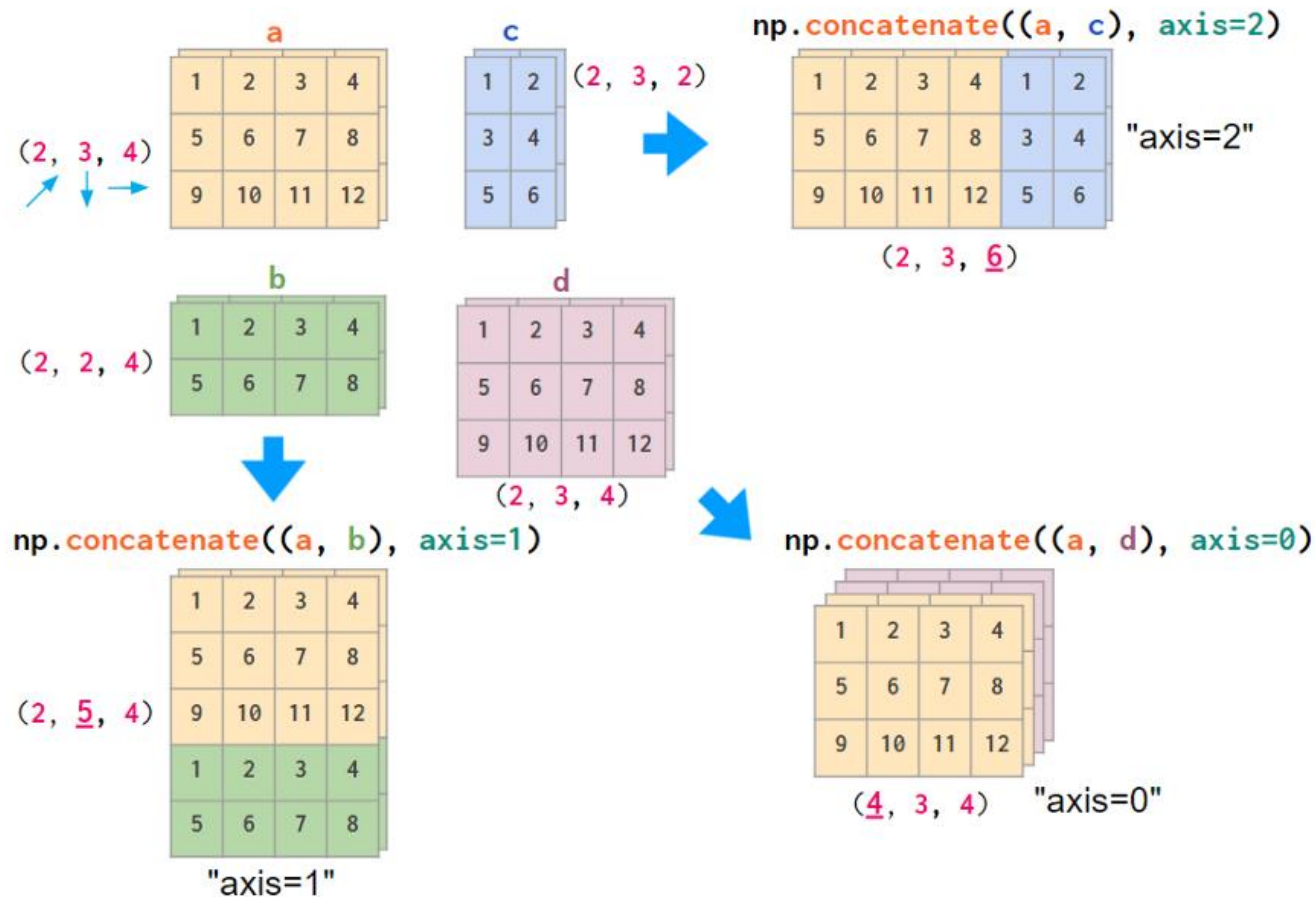


- 矩阵的堆叠（合并）



NumPy 运算

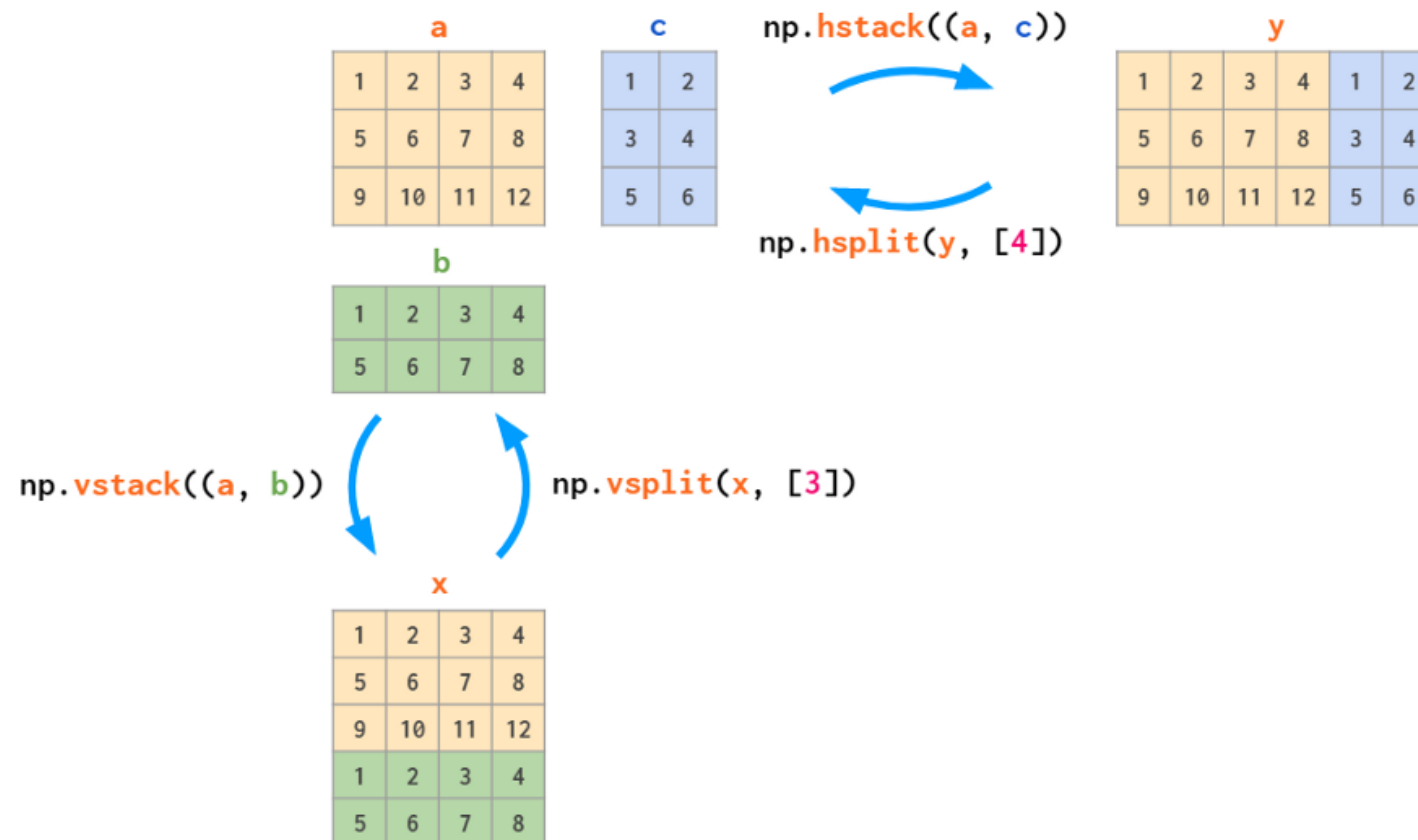
- 高维的时候使用 `concatenate()` 函数更方便





NumPy 运算

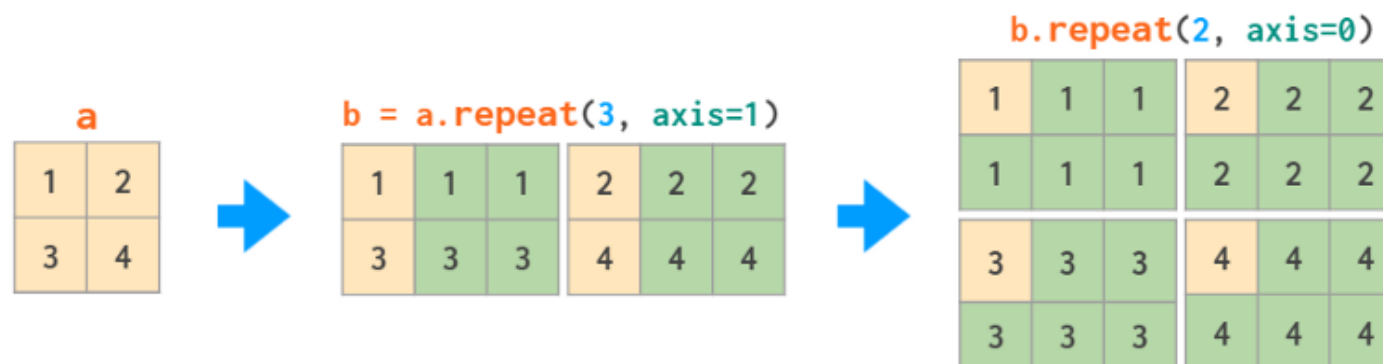
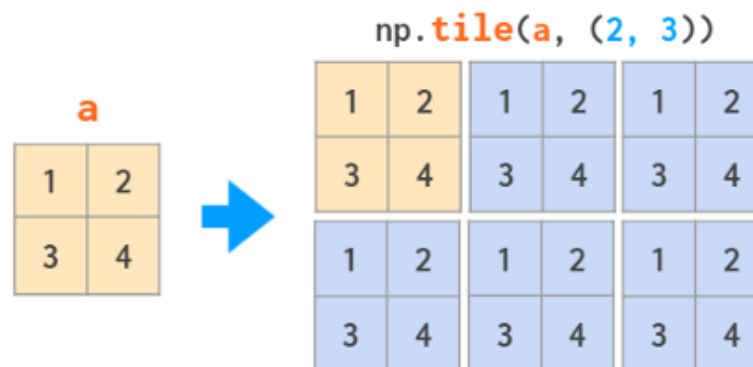
- 矩阵的切分





NumPy 运算

- 矩阵的自我重复





NumPy 运算

- 矩阵的部分删除

a

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

`np.delete(a, [1, 3], axis=1)`

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15



1	3	5
6	8	10
11	13	15

`np.delete(a, 1, axis=0)`

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15



1	2	3	4	5
11	12	13	14	15

`np.delete(a, np.s_[1:-1], axis=1)`

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15



1	5
6	10
11	15

`= np.delete(a, slice(1,-1), axis=1)`



NumPy 运算

- 矩阵的插入

`np.insert(h, [1, 2], 0, axis=1)`

1	0	3	0	5
6	0	8	0	10
11	0	13	0	15



h

1	3	5
6	8	10
11	13	15

0 1 2

`np.insert(v, 1, 7, axis=0)`

1	2	3	4	5
7	7	7	7	7
11	12	13	14	15



v

1	2	3	4	5
11	12	13	14	15

`np.insert(u, [1], w, axis=1)`

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15



u

1	5
6	10
11	15

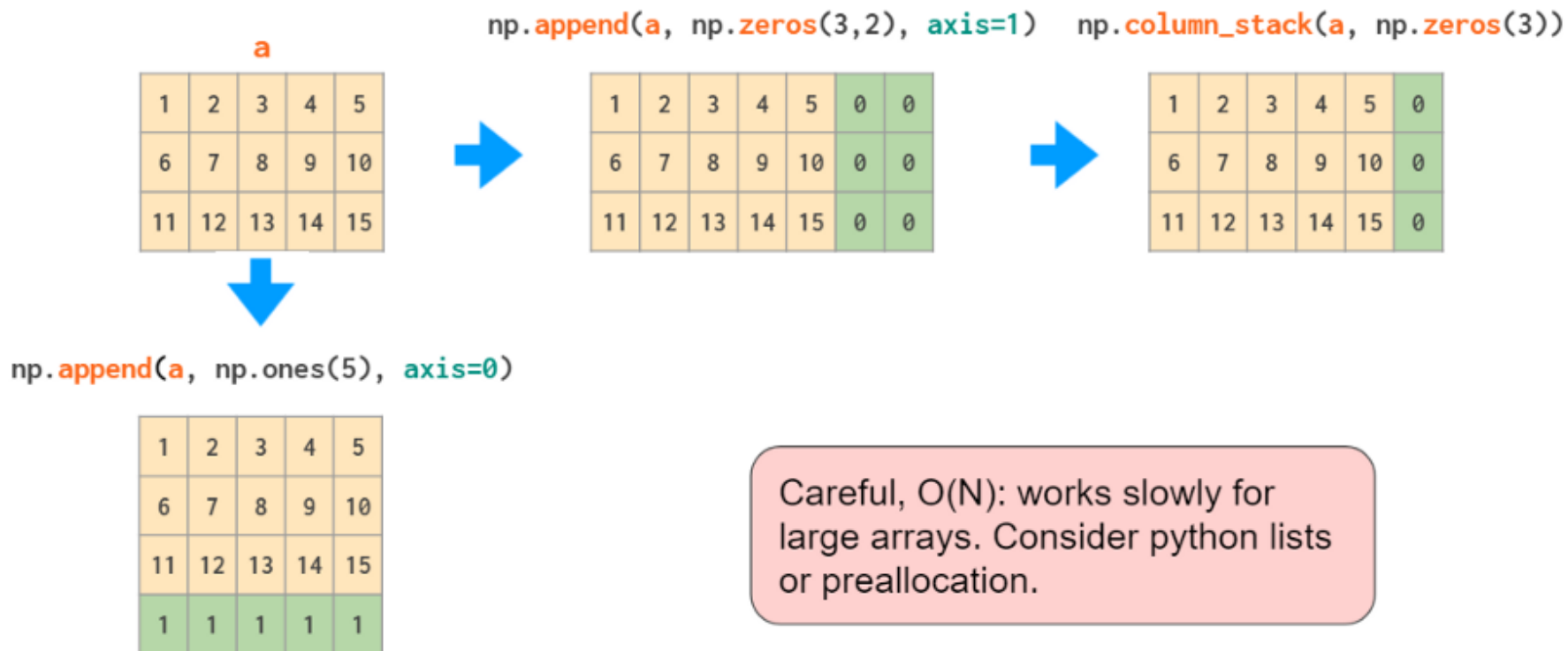
w

2	3	4
7	8	9
12	13	14



NumPy 运算

- 矩阵的append

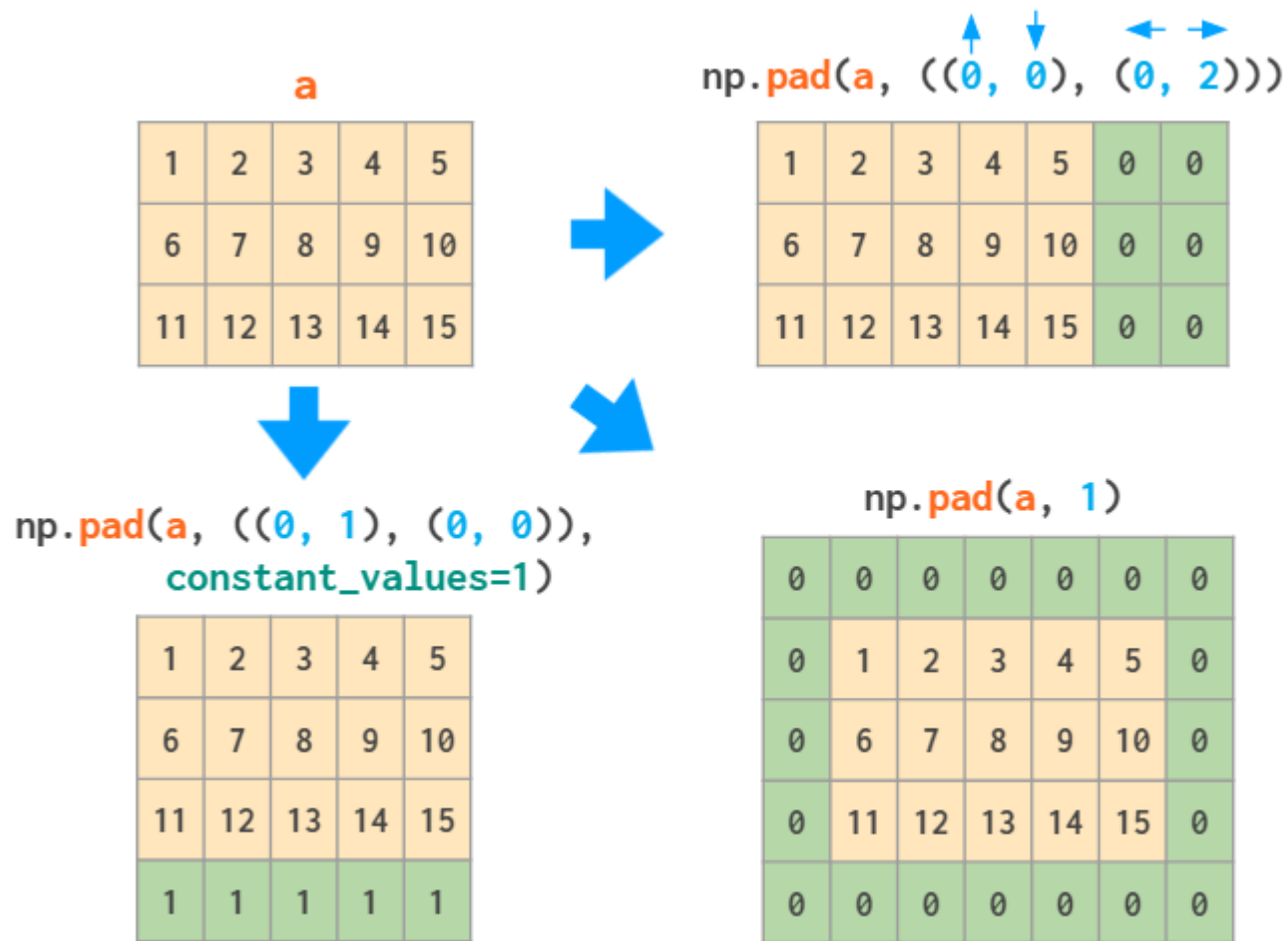


Careful, $O(N)$: works slowly for large arrays. Consider python lists or preallocation.



NumPy 运算

- 矩阵的外边框增加





NumPy 运算

- 乘法开方对数等

$$a^2 = \begin{bmatrix} 2 & 3 \end{bmatrix} ** 2 = \begin{bmatrix} 4 & 9 \end{bmatrix}$$

$$\sqrt{a} = \text{np.sqrt}(\begin{bmatrix} 4 & 9 \end{bmatrix}) = \begin{bmatrix} 2. & 3. \end{bmatrix}$$

$$e^a = \text{np.exp}(\begin{bmatrix} 1 & 2 \end{bmatrix}) = \begin{bmatrix} 2.718 & 7.389 \end{bmatrix}$$

$$\ln a = \text{np.log}(\begin{bmatrix} \text{np.e} & \text{np.e**2} \end{bmatrix}) = \begin{bmatrix} 1. & 2. \end{bmatrix}$$



NumPy 运算

- 点积与叉积

$$\vec{a} \cdot \vec{b} = \text{np.dot}\left(\begin{bmatrix} 1 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 4 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 1 & 2 \end{bmatrix} @ \begin{bmatrix} 3 & 4 \end{bmatrix} = \begin{bmatrix} 11 \end{bmatrix}$$

$$\vec{a} \times \vec{b} = \text{np.cross}\left(\begin{bmatrix} 2 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & 0 & 6 \end{bmatrix}$$



NumPy 运算

- 三角函数与欧几里得范数

$$\text{np.sin}(\text{np.pi}, \text{np.pi}/2) = 0. \quad 1.$$

$$\text{np.arcsin}(0., 1.) = 0. \quad 1.57$$

sin	arcsin	sinh	arcsinh
cos	arccos	cosh	arccosh
tan	arctan	tanh	arctanh

$$\text{np.hypot}(3., 5., 4., 12.) = 5. \quad 13.$$



NumPy 运算

- 取整函数

`np.floor(`

1.1	1.5	1.9	2.5
-----	-----	-----	-----

`) =`

1.	1.	1.	2.
----	----	----	----

`np.ceil(`

1.1	1.5	1.9	2.5
-----	-----	-----	-----

`) =`

2.	2.	2.	3.
----	----	----	----

`np.round(`

1.1	1.5	1.9	2.5
-----	-----	-----	-----

`) =`

1.	2.	2.	2.
----	----	----	----



NumPy 运算

- 简单统计
- 和、最值（以及下标）、均值、标准差

`np.max(`

1	2	3
---	---	---

`) =`

3

1	2	3
---	---	---

`.max()` =

3

1	2	3
---	---	---

`.argmax()` =

2

1	2	3
---	---	---

`.min()` =

1

1	2	3
---	---	---

`.argmin()` =

0

0 1 2

1	2	3
---	---	---

`.sum()` =

6

1	2	3
---	---	---

`.mean()` =

2

1	2	3
---	---	---

`.var()` =

0.67

1	2	3
---	---	---

`.std()` =

0.82

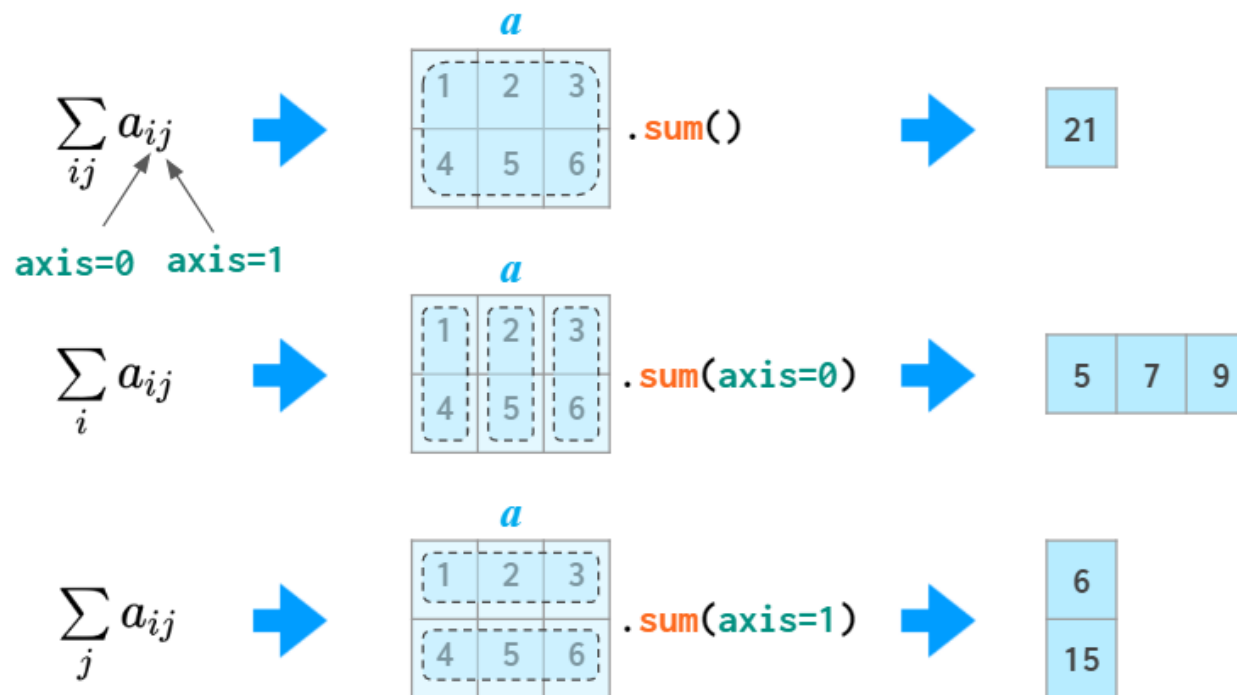
$$\bar{S}^2 = \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2, \quad \bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

$$a = 2 \pm 0.82$$



NumPy 运算

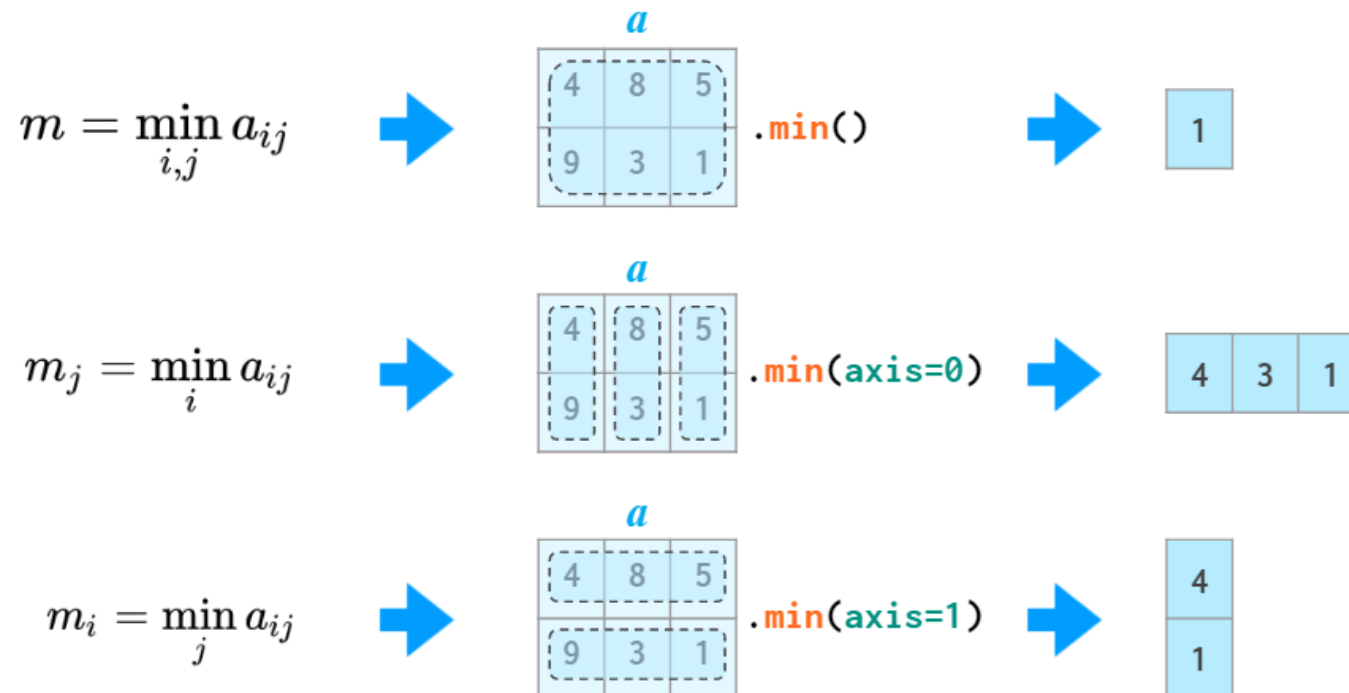
- 到二维的时候就有轴的问题了
- 通常来说是记不住哪个轴是行和列的...
- 可以每个都试一试





NumPy 运算

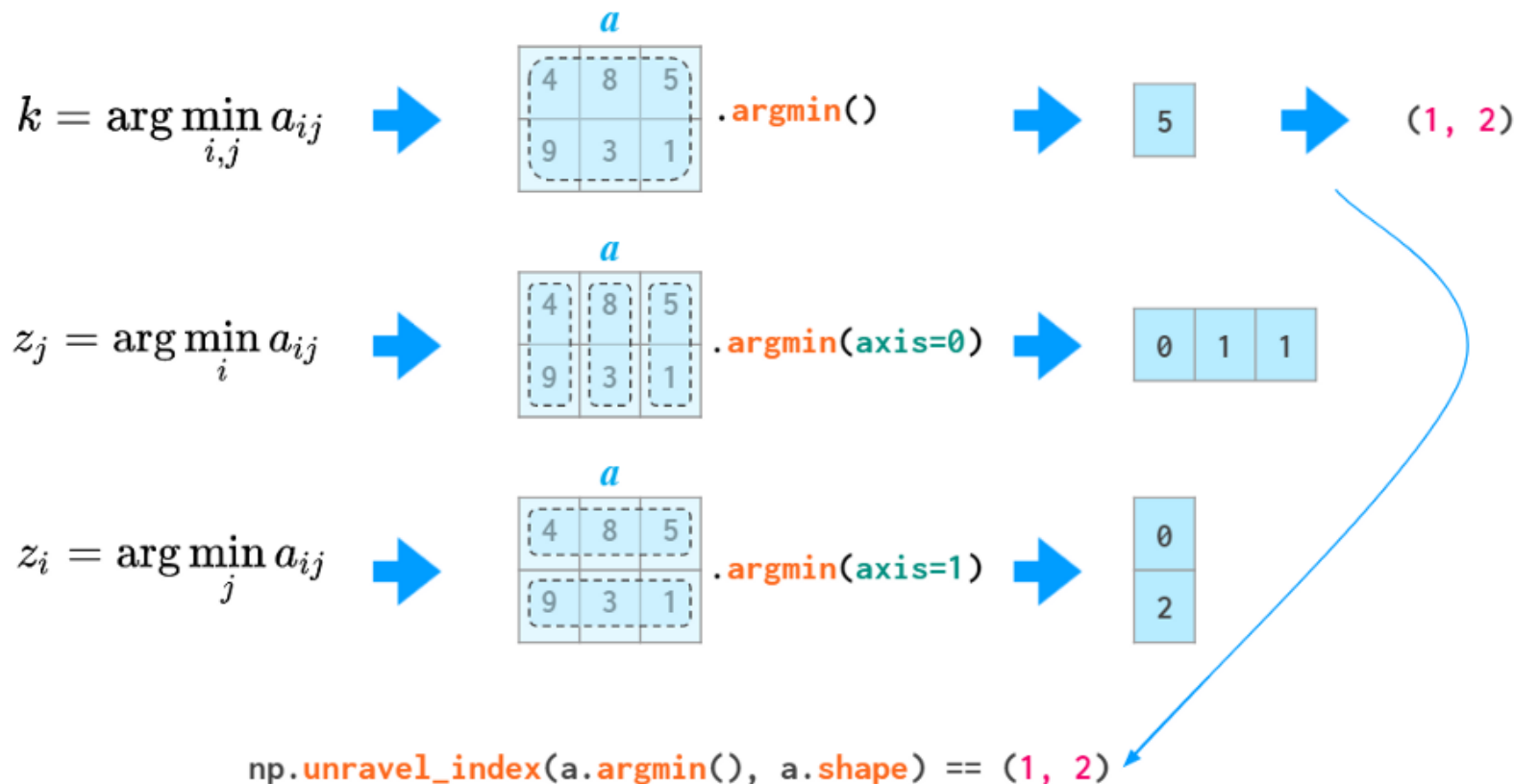
- 到二维的时候就有轴的问题了
- 通常来说是记不住哪个轴是行和列的...
- 可以每个都试一试





NumPy 运算

- 加了 arg 是找下标





NumPy 运算

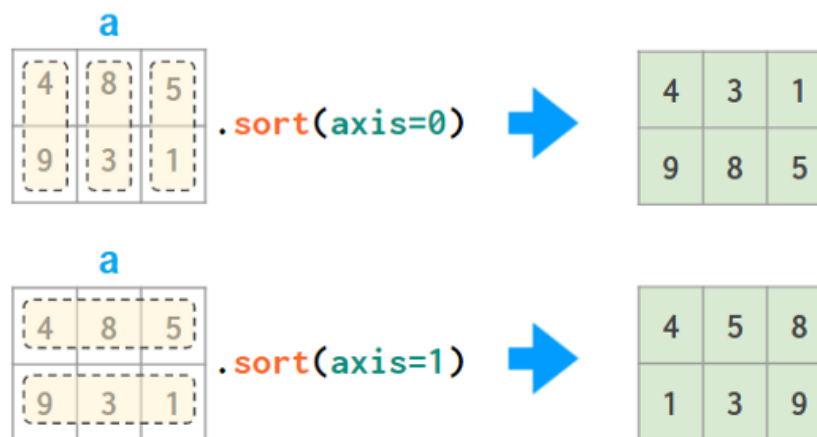
• 矩阵排序

python lists

```
a.sort()
sorted(a)
a.sort(key=f)
a.sort(reversed=False)
-
```

numpy arrays

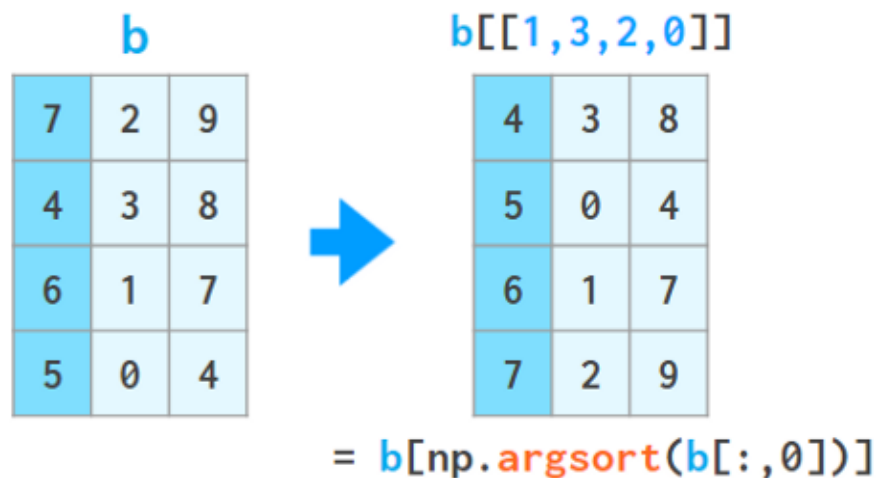
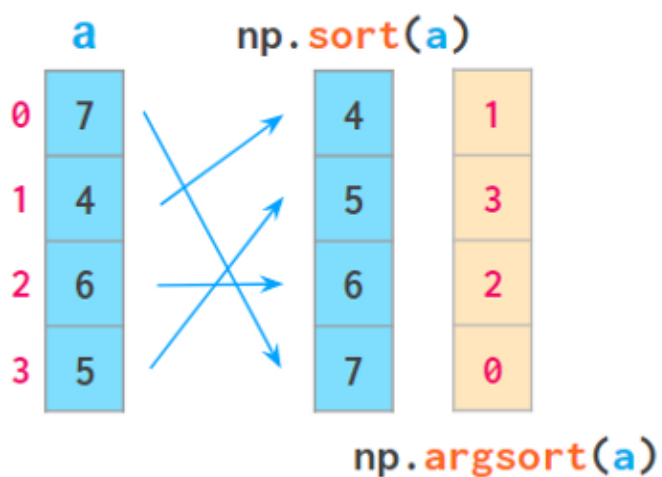
```
a.sort()           sorts in-place
np.sort(a)         returns new sorted array
-                 key function
-                 ascending/descending
a.sort(axis=-1)    which axis to sort along
```





NumPy 运算

- 矩阵排序





休息一下



可读性与速度的平衡

- Python 为了可读性牺牲了速度
- NumPy 为了速度牺牲了可读性
- 我全都要？





来看看这段代码

```
def function_2(seq, sub):  
    target = np.dot(sub, sub)  
    candidates = np.where(np.correlate(seq, sub, mode='valid') == target)[0]  
    check = candidates[:, np.newaxis] + np.arange(len(sub))  
    mask = np.all((np.take(seq, check) == sub), axis=-1)  
    return candidates[mask]
```



再来看看这段代码

```
def function_2(seq, sub):  
    target = np.dot(sub, sub)  
    candidates = np.where(np.correlate(seq, sub, mode='valid') == target)[0]  
    check = candidates[:, np.newaxis] + np.arange(len(sub))  
    mask = np.all((np.take(seq, check) == sub), axis=-1)  
    return candidates[mask]
```

```
def function_1(seq, sub):  
    return [i for i in range(len(seq) - len(sub)) if seq[i:i+len(sub)] == sub]
```

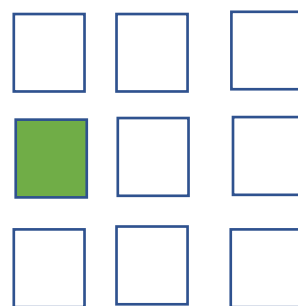
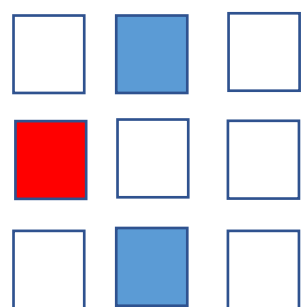
翻译翻译？



就这?

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

- 来看个小题目：给定一个矩阵，
- 每个位置的新值是原始值加上它右上角和右下角的值
- 没有右上角右下角的当作那个地方为0



绿 = 红 + 蓝



思路

算法优化 – 空间换时间

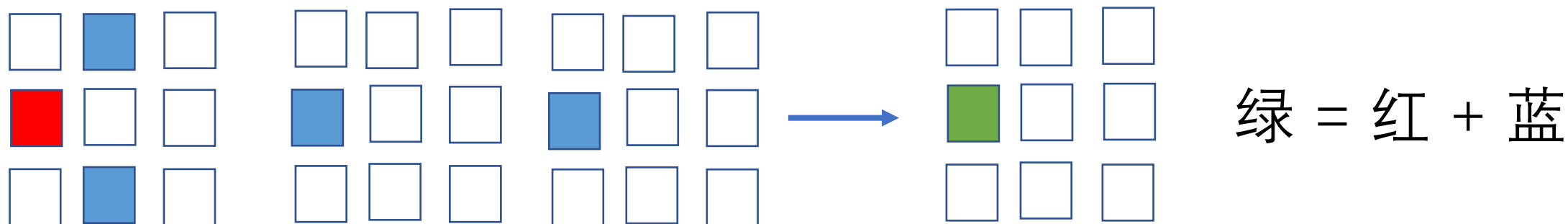
- 也就是俗称的“打表”





思路

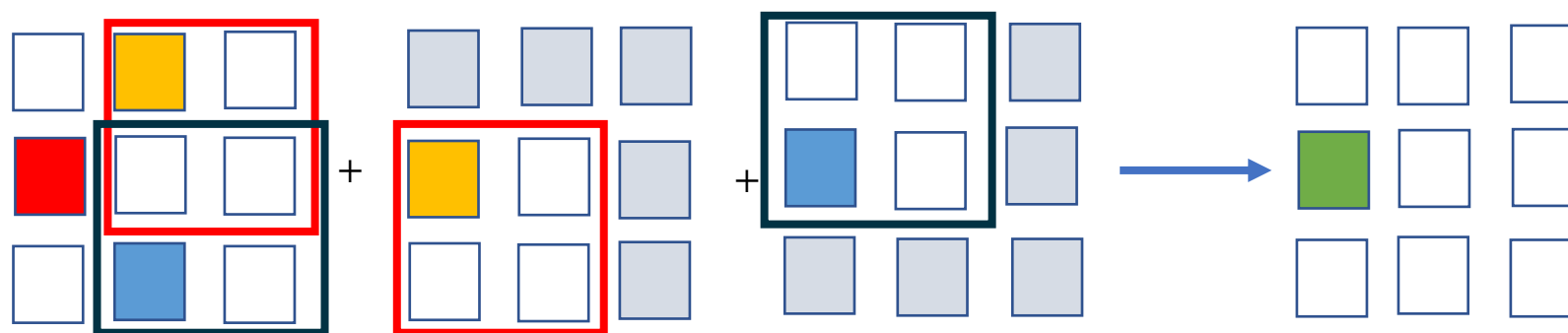
- 空间换时间
- 对齐矩阵的格子





思路

- 空间换时间
- 对齐矩阵的格子
- 使用索引创建新的矩阵，在外面使用 pad 垫上0
- 三个矩阵相加



绿 = 红 + 蓝



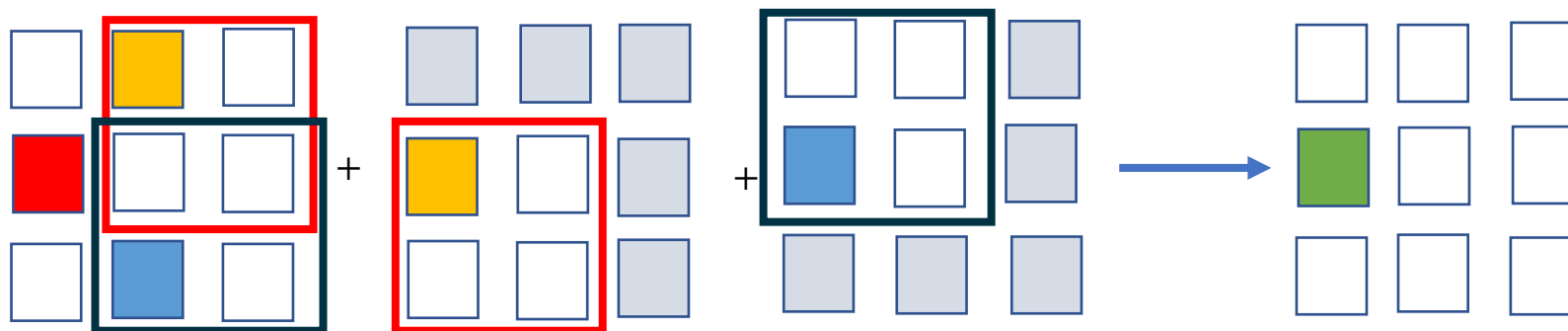
Code

```
[1] a = np.arange(9).reshape(3,3)
[2] ✓ 0.4s
[3] ✓ 0.6s
... array([[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]])
```

```
b = np.pad(a[:-1,1:],((1,0),(0,1)))
[10] ✓ 0.6s
... array([[0, 0, 0],
          [1, 2, 0],
          [4, 5, 0]])

c = np.pad(a[1:,1:],((0,1),(0,1)))
[12] ✓ 0.6s
... array([[4, 5, 0],
          [7, 8, 0],
          [0, 0, 0]])
```

```
[13] ✓ 0.5s
... a + b + c
... array([[ 4,  6,  2],
          [11, 14,  5],
          [10, 12,  8]])
```



绿 = 红 + 蓝

Part-2 手写SIMD向量化



SIMD 是什么?

- Single Instruction Multiple Data, 单指令多数据流
- 在x86架构下, SIMD一般和SSE和AVX等指令集联系在一起
- SSE和AVX指令集中提供了大量可以单指令操作多个数据单元的指令



数据个数=加速倍数?

- 很自然的会认为，SIMD指令同时操作2个数据，那加速比就应该是2
- 真的是这样吗？🤔



数据个数=加速倍数？

- 很自然的会认为，SIMD指令同时操作2个数据，那加速比就应该是2
- 事实上很复杂：内存带宽使用，解码消耗减小等等，你很难说清楚，具体问题具体分析，但可以使用倍数估算
- 当作为整体代码一部分时，情况就更加复杂了
- 甚至可能提供的AVX2指令实际上是由2条AVX指令模拟出来的



越长越好？

- 一条AVX2指令能处理256位的数据，一条AVX512指令一次能处理512位数据，那为什么不再长亿点？
- 一条指令干翻它10GB的数据量不香吗 🌟
- 显然这不现实 🤡



越长越好？

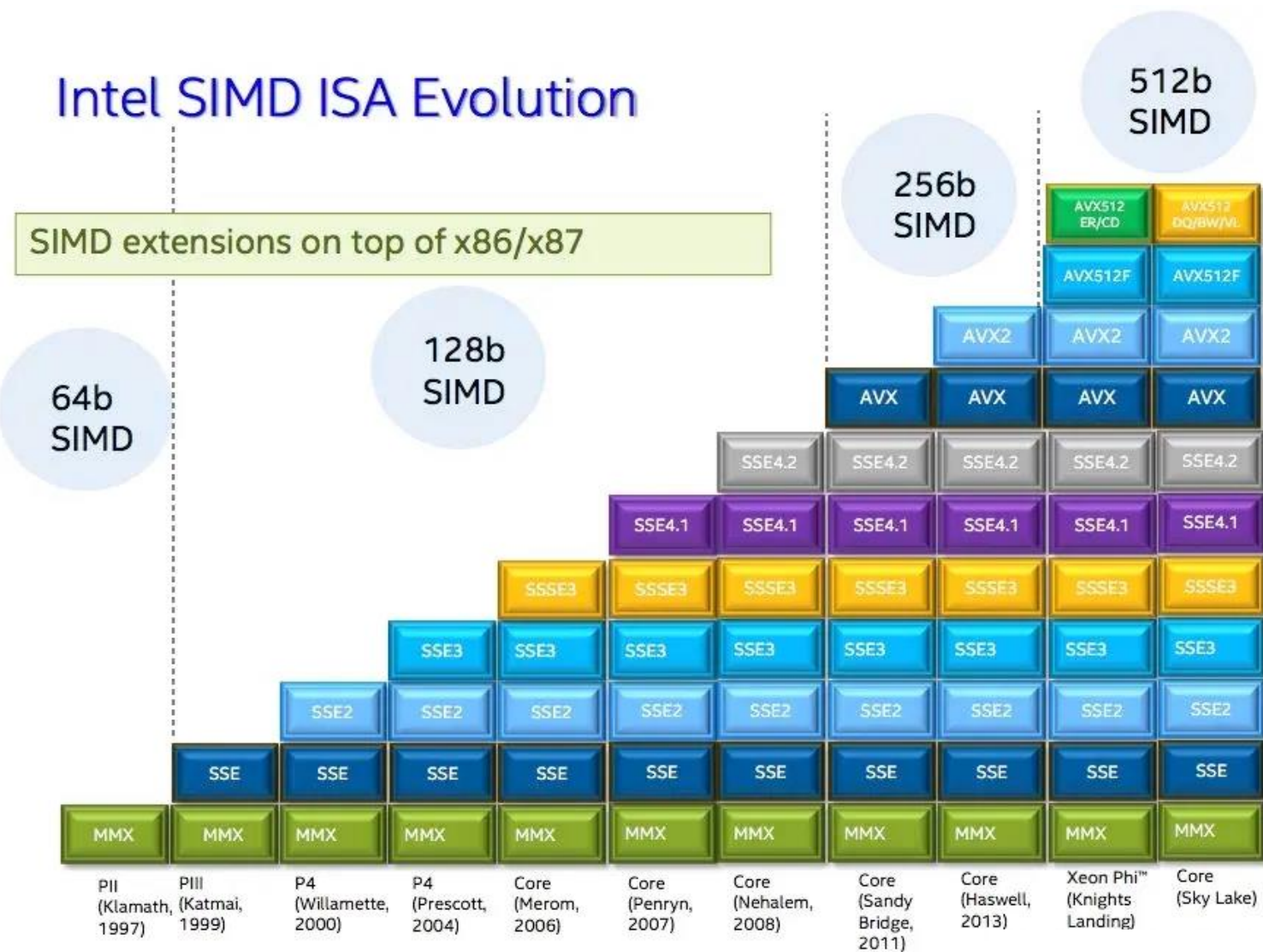
- 指令的功能是需要硬件实现的，一条指令处理8个double就意味至少需要8个double的运算单元，这意味着更大的面积，更高的成本，更多的发热
- AVX512之前被戏称为大火炉，对CPU负载高频率跑高容易不稳定
- 发热严重，过热导致降频，导致了AVX512表现不如AVX2的奇景



Intel?

- MMX
- SSE
- SSE2
- AVX
- AVX2
- AVX512

Intel SIMD ISA Evolution





ARM?

- NEON
- 你们有人现在用的M系列芯片就是ARM架构的



为什么要手写? 🤯

- 简单的情况（如矩阵乘法）下其实没有必要，你只要 `-O3 -mavx2`，编译器就能做的很好
- 但如果你的代码结构复杂，循环难以界定边界，甚至还有分支，那就只能靠你自己手写了



我该怎么写？

<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#>

- 看文档！！！！
- 你需要什么
- 再看有什么

Intel® Intrinsics Guide

Updated
04/22/2022

Version
3.6.2

Instruction Set

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVMML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask
- ☐ Miscellaneous
- ☐ Move
- ☐ OS-Targeted
- ☐ Probability/Statistics
- ☐ Random
- ☐ C++

The Intel® Intrinsics Guide contains reference information for Intel intrinsics, which provide access to Intel instructions such as Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX), and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- For information about how Intel compilers handle intrinsics, view the [Intel® C++ Compiler Classic Developer Guide and Reference](#).
- For questions about Intel intrinsics, visit the [Intel® C++ Compiler board](#).

_mm_search

```
void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)                vp2intersectd
void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)            vp2intersectd
void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2)          vp2intersectd
void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)                vp2intersectq
void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)            vp2intersectq
void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)            vp2intersectq
__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssd
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssd
__m512i _mm512_maskz_4dpwssd_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssd
__m512i _mm512_4dpwssds_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssds
__m512i _mm512_mask_4dpwssds_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssds
__m512i _mm512_maskz_4dpwssds_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)    rp4dpwssds
__m512 _mm512_4fmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fmaddps
__m512 _mm512_mask_4fmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fmaddps
__m512 _mm512_maskz_4fmadd_ps (__mmask16 k, __m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fmaddps
__m128 _mm_4fmadd_ss (__m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)      vf4fmaddss
__m128 _mm_mask_4fmadd_ss (__m128 src, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)    vf4fmaddss
__m128 _mm_maskz_4fmadd_ss (__mmask8 k, __m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)    vf4fmaddss
__m512 _mm512_4fnmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fnmaddps
__m512 _mm512_mask_4fnmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fnmaddps
__m512 _mm512_maskz_4fnmadd_ps (__mmask16 k, __m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)    vf4fnmaddps
__m128 _mm_4fnmadd_ss (__m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)      vf4fnmaddss
__m128 _mm_mask_4fnmadd_ss (__m128 src, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)    vf4fnmaddss
__m128 _mm_maskz_4fnmadd_ss (__mmask8 k, __m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)    vf4fnmaddss
__m128i _mm_abs_epi16 (__m128i a)                                                            pabsw
__m128i _mm_mask_abe_epi16 (__m128i src, __mmask8 k, __m128i a)                             vtabsw
```

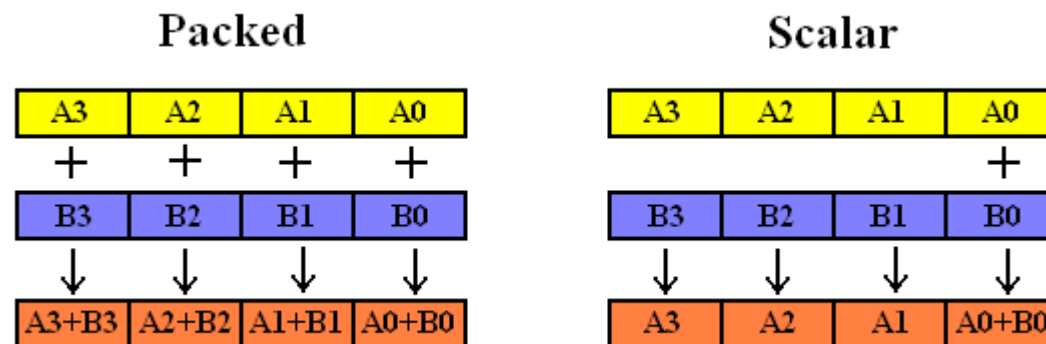
 Give Feedback



怎么看文档

- 命名有规则
- 看返回类型和输入类型筛选
- 指令类型_操作_单元类型
- 查找关键字
- 上Google, 知乎学习 (我猜你们做Bonus第一步就是这个)

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```





怎么看文档

- 需要的头文件
- 实际汇编指令
- Iscpu可以看处理器flags
- 伪代码等效操作
- 实际架构下的性能

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```

Synopsis

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
#include <immintrin.h>
Instruction: vaddpd ymm, ymm, ymm
CPUID Flags: AVX
```

Description

Add packed double-precision (64-bit) floating-point elements in `a` and `b`, and store the results in `dst`.

Operation

```
FOR j := 0 to 3
    i := j*64
    dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Broadwell	3	1
Haswell	3	1
Ivy Bridge	3	1



向量化前后

- 分支可以向量化
你没看错
- 向量化完
可读性喂了🐼!
- 此处向量化的含义是
真正的同时操作多个数据组成的向量

```
double l = m_lvec[i];  
double a = m_avec[i];  
double b = m_bvec[i];  
  
double _distlab = (1 - kseeds1[n])*(1 - kseeds1[n]) +  
                  (a - kseedsa[n])*(a - kseedsa[n]) +  
                  (b - kseedsb[n])*(b - kseedsb[n]);  
  
double _distxy = (x - kseedsx[n])*(x - kseedsx[n]) +  
                 (y - kseedsy[n])*(y - kseedsy[n]);  
  
//double dist = _distlab/maxlab[n] + _distxy*invxywt; //only varying m, prettier superpixels  
//double dist = _distlab/maxlab[n] + _distxy*invxywt; //varying both m and S  
//-----  
  
distlab[i] = _distlab;  
distxy[i] = _distxy;  
  
if( dist < distvec[i] )  
{  
    distvec[i] = dist;  
    klabels[i] = n;  
}
```

```
_mm256d l_vec = _mm256_load_pd(&m_lvec[i]);  
_mm256d a_vec = _mm256_load_pd(&m_avec[i]);  
_mm256d b_vec = _mm256_load_pd(&m_bvec[i]);  
  
_mm256d x_vec =  
    _mm256_set_pd((double)(x + 3), (double)(x + 2),  
                  (double)(x + 1), (double)(x));  
_mm256d y_vec = _mm256_set1_pd((double)y);  
  
_mm256d l_vec_t1 = _mm256_sub_pd(l_vec, _kseeds1_vec);  
l_vec_t1 = _mm256_mul_pd(l_vec_t1, l_vec_t1);  
_mm256d a_vec_t1 = _mm256_sub_pd(a_vec, _kseedsa_vec);  
a_vec_t1 = _mm256_mul_pd(a_vec_t1, a_vec_t1);  
_mm256d b_vec_t1 = _mm256_sub_pd(b_vec, _kseedsb_vec);  
b_vec_t1 = _mm256_mul_pd(b_vec_t1, b_vec_t1);  
_mm256d _distlab_vec =  
    _mm256_add_pd(l_vec_t1, a_vec_t1);  
distlab_vec = _mm256_add_pd(_distlab_vec, b_vec_t1);  
  
_mm256d x_vec_t1 = _mm256_sub_pd(x_vec, _kseedsx_vec);  
x_vec_t1 = _mm256_mul_pd(x_vec_t1, x_vec_t1);  
_mm256d y_vec_t1 = _mm256_sub_pd(y_vec, _kseedsy_vec);  
y_vec_t1 = _mm256_mul_pd(y_vec_t1, y_vec_t1);  
_mm256d distxy_vec = _mm256_add_pd(x_vec_t1, y_vec_t1);  
  
_mm256d dist_vec_t1 =  
    _mm256_div_pd(_distlab_vec, maxlab_vec);  
_mm256d dist_vec_t2 =  
    _mm256_mul_pd(distxy_vec, invxywt_vec);  
_mm256d dist_vec =  
    _mm256_add_pd(dist_vec_t1, dist_vec_t2);  
  
_mm256_store_pd(&distlab[i], _distlab_vec);  
  
_mm256d distvec_vec = _mm256_load_pd(&distvec[i]);  
_mm256d cmp_res_vec =  
    _mm256_cmp_pd(dist_vec, distvec_vec, _CMP_LT_OQ);  
distvec_vec = _mm256_blendv_pd(distvec_vec, dist_vec,  
                                cmp_res_vec);  
_mm256_store_pd(&distvec[i], distvec_vec);  
  
_mm256i permuted_vec = _mm256_permutevar8x32_epi32(  
    _mm256_castpd_si256(cmp_res_vec), K_PERM_VEC);  
_mm128i cmp_int_vec =  
    _mm256_castsi256_si128(permuted_vec);  
  
_mm128i n_vec = _mm_set1_epi32(n);  
_mm_maskstore_epi32(&klabels[i], cmp_int_vec, n_vec);  
  
x += 4;
```



基本流程

- Load需要计算的数据到向量寄存器
- 进行需要的向量化计算
- Store将向量寄存器的数据存回内存
- 和把大象塞进冰箱一样，简单吧 (X





常见问题

- 内存对齐
- 循环边界不确定——展开非整数边界
- 循环分支的开销掩盖了SIMD提升——循环展开
- 寄存器数量超限——一般默认16个256位寄存器进行考量



小结

- 如果想要熟练掌握手写SIMD向量化的优化技术
实践才是王道
- 大多数时候自动向量化就够了
- 一定一定要在**最后**再进行手写SIMD优化



谢谢大家！