编程语言字幕组 进入小组

组员：183 待译：182 完结：335

图解 | NumPy可视化指南

594 2021-01-18 10:36:49

分享 3 收藏

翻译评分： ★★★★★ 10.0 （2人评价）

发起：安德鲁•约翰 校对：Heidi 审核：佑而

参与翻译（1人）：季一帆

默认 只看中文 只看英文

英文原文：NumPy Illustrated: The Visual Guide to NumPy

标签： Python

01

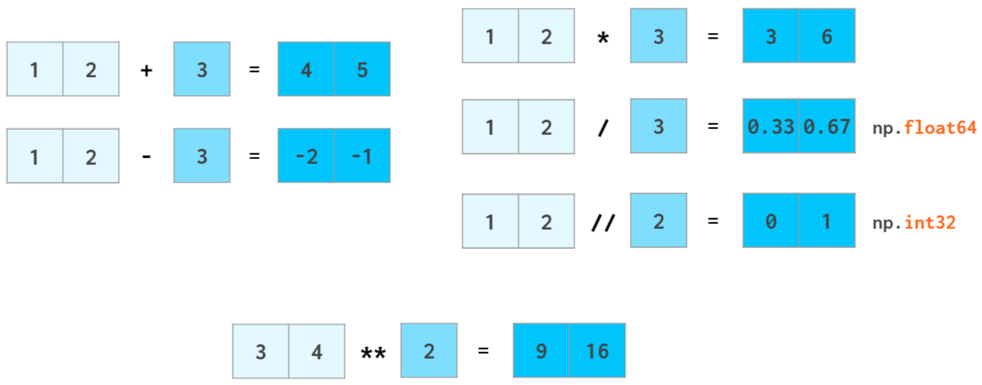


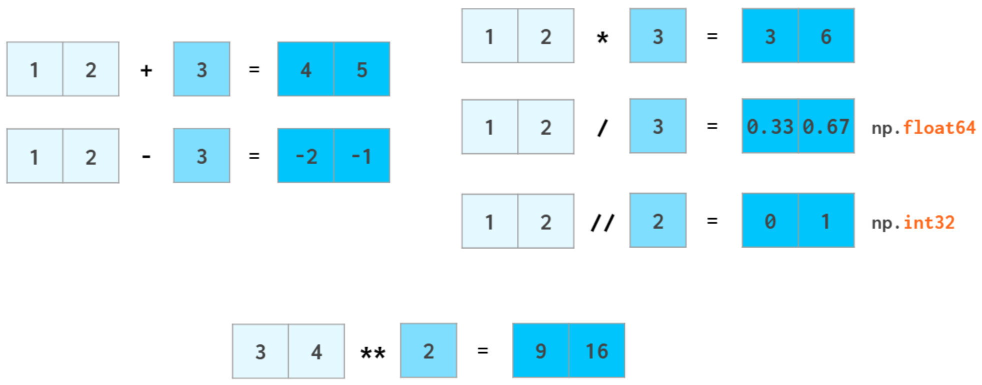
Image credit: Author

NumPy is a fundamental library that most of the widely used Python data processing libraries are built upon (pandas, OpenCV), inspired by (PyTorch), or can efficiently share data with (TensorFlow, Keras, etc). Understanding how NumPy works gives a boost to your skills in those libraries as well. It is also possible to run NumPy code with no or minimal changes on GPU[1].

The central concept of NumPy is an n-dimensional array. The beauty of it is that most operations look just the same, no matter how many dimensions an array has. But 1D and 2D cases are a bit special. The article consists of three parts:

1. Vectors, the 1D Arrays
2. Matrices, the 2D Arrays
3. 3D and above

I took a great article, “A Visual Intro to NumPy” by Jay Alammar[2], as a starting point, significantly expanded its coverage, and amended a pair of nuances.



NumPy是一个广泛适用的Python数据处理库，pandas, OpenCV等库都基于numpy。同时，在PyTorch、TensorFlow、Keras等深度许欸小框架中，了解numpy将显著提高数据共享和处理能力，甚至无需过多更改就可以在GPU运行计算。

n维数组是NumPy的核心概念，这样的好处，尽管一维和而为数组的处理方式有些差异，但多数不同维数组的操作是一样的。本文将对以下三个部分展开介绍：

1. 向量——一维数组
2. 矩阵——二维数组
3. 3维及更高维数组

本文受JayAlammar的文章 “ A Visual Intro to NumPy” 的启发，并对其做了更详细丰富的介绍。

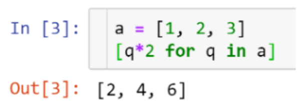
02

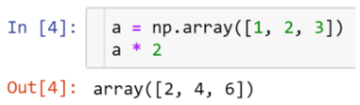
Numpy Array vs. Python List

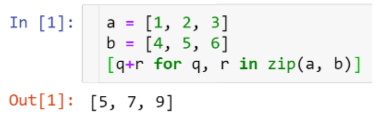
numpy数组 vs. Python 列表

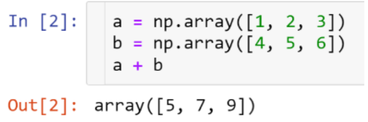
At first glance, NumPy arrays are similar to Python lists. They both serve as containers with fast item getting and setting and somewhat slower inserts and removals of elements.

The hands-down simplest example when NumPy arrays beat lists is arithmetic:


```
In [3]: a = [1, 2, 3]
        [q*2 for q in a]
Out[3]: [2, 4, 6]
```

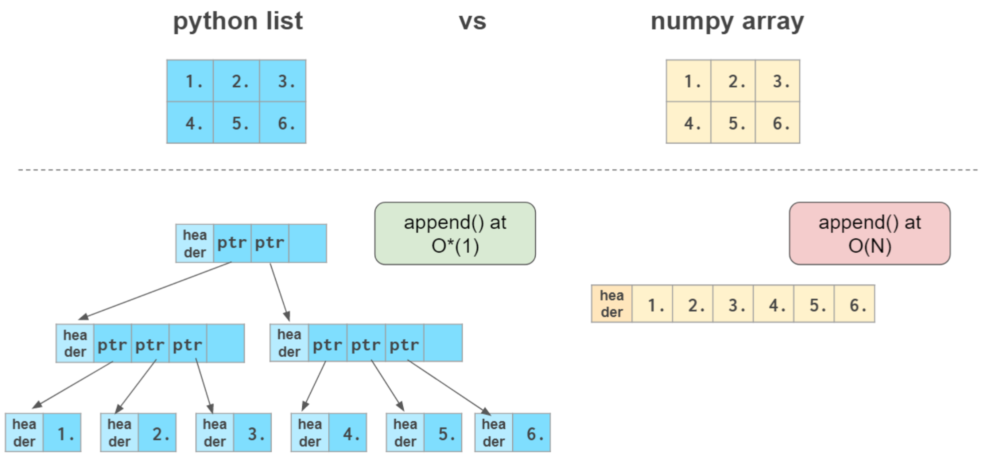

```
In [4]: a = np.array([1, 2, 3])
        a * 2
Out[4]: array([2, 4, 6])
```


```
In [1]: a = [1, 2, 3]
        b = [4, 5, 6]
        [q+r for q, r in zip(a, b)]
Out[1]: [5, 7, 9]
```


```
In [2]: a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])
        a + b
Out[2]: array([5, 7, 9])
```

Other than that, NumPy arrays are:

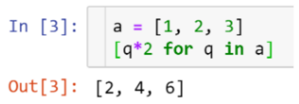
- more compact, especially when there’s more than one dimension
- faster than lists when the operation can be vectorized
- slower than lists when you append elements to the end
- usually homogeneous: can only work fast with elements of one type

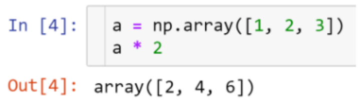


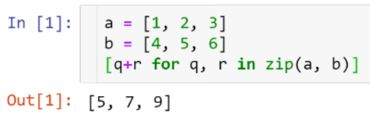
Here O(N) means that the time necessary to complete the operation is proportional to the size of the array (see Big-O Cheat Sheet[3] site), and O*(1) (the so-called “amortized” O(1)) means that the time does not generally depend on the size of the array (see Python Time Complexity[4] wiki page)

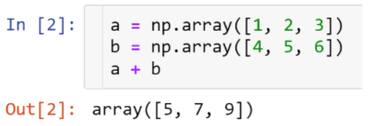
乍看上去，NumPy数组与Python列表极其相似。它们都用来装载数据，都能够快速添加或获取元素，插入和移除元素则比较慢。

当然相比python列表，numpy数组可以直接进行算术运算：


```
In [3]: a = [1, 2, 3]
        [q*2 for q in a]
Out[3]: [2, 4, 6]
```

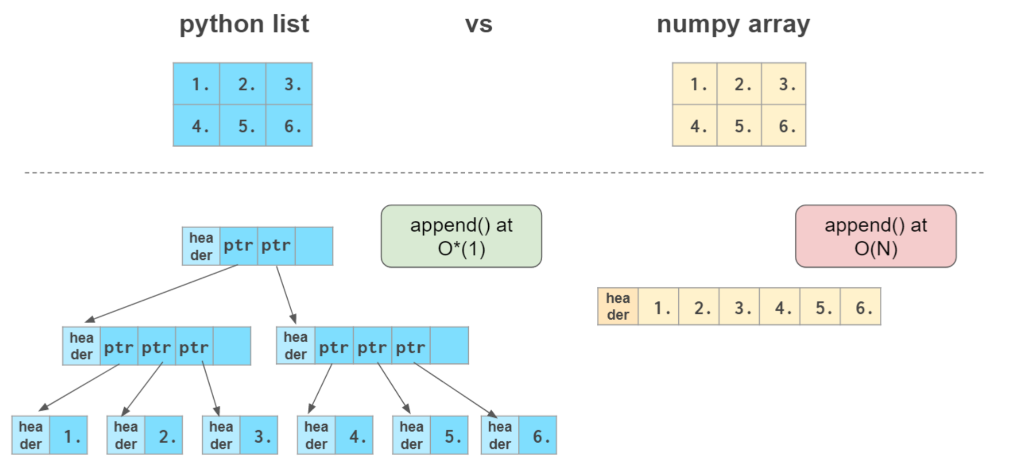

```
In [4]: a = np.array([1, 2, 3])
        a * 2
Out[4]: array([2, 4, 6])
```


```
In [1]: a = [1, 2, 3]
        b = [4, 5, 6]
        [q+r for q, r in zip(a, b)]
Out[1]: [5, 7, 9]
```


```
In [2]: a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])
        a + b
Out[2]: array([5, 7, 9])
```

除此之外，numpy数组还具有以下特点：

- 更紧凑，高维时尤为明显
- 向量化后运算速度比列表更快
- 在末尾添加元素时不如列表高效
- 元素类型一般比较固定



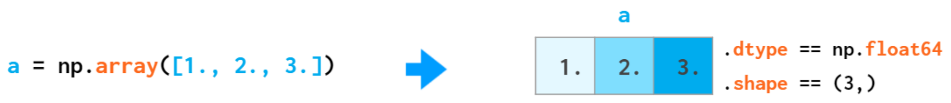
其中，O(N)表示完成操作所需的时间与数组大小成正比（请见 Big-O Cheat Sheet），O(1)表示操作时间与数组大小无关（详见Time Complexity）。

03

1. Vectors, the 1D Arrays

Vector initialization

One way to create a NumPy array is to convert a Python list. The type will be auto-deduced from the list element types:


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

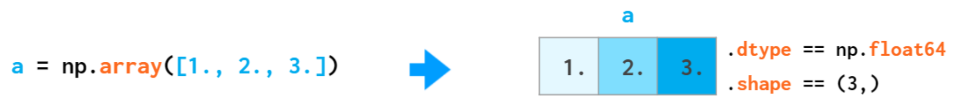
Be sure to feed in a homogeneous list, otherwise you’ll end up with `dtype=’object’`, which annihilates the speed and only leaves the syntactic sugar contained in NumPy.

NumPy arrays cannot grow the way a Python list does: No space is reserved at the end of the array to facilitate quick appends. So it is a common practice to either grow a Python list and convert it to a NumPy

1.向量与1维数组

向量初始化

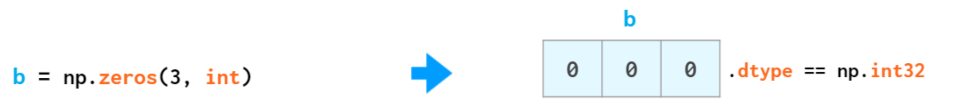
通过Python列表可以创建NumPy数组，如下将列表元素转化为一维数组：


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

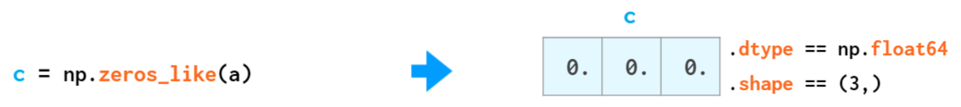
注意，确保列表元素类型相同，否则`dtype=’object'`，将影响运算甚至产生语法错误。

由于在数组末尾没有预留空间以快速添加新元素，NumPy数组无法像Python列表那样增长，因此，通常的做法是在变长Python列表中准备好数据，然后将其转换为NumPy数组，或是使用np.zeros或np.empty预先分配必要的空间：

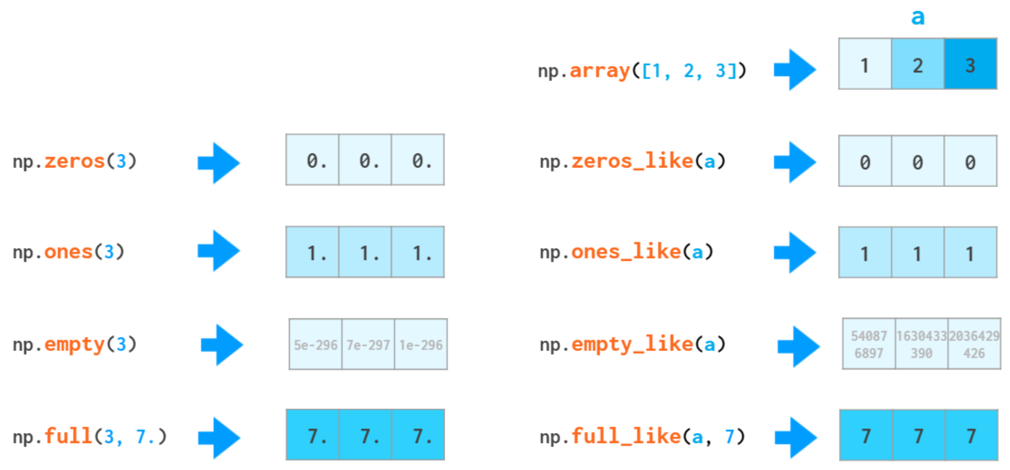
array when it is ready or to preallocate the necessary space with `np.zeros` or `np.empty` :



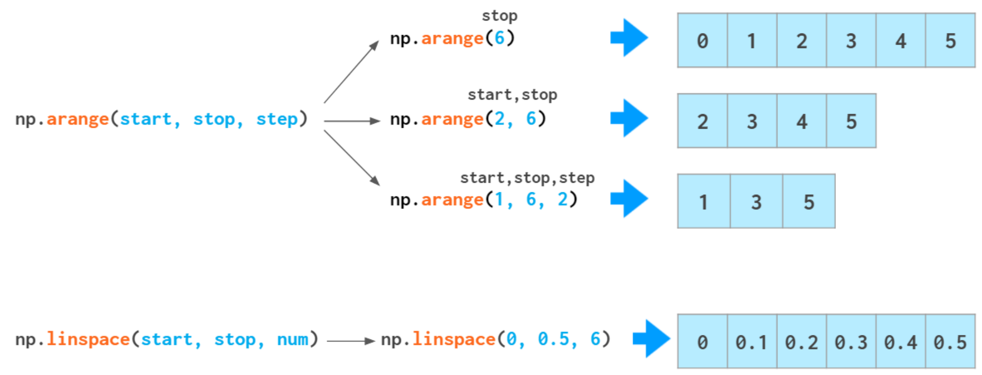
It is often necessary to create an empty array which matches the existing one by shape and elements type:



Actually, all the functions that create an array filled with a constant value have a `_like` counterpart:

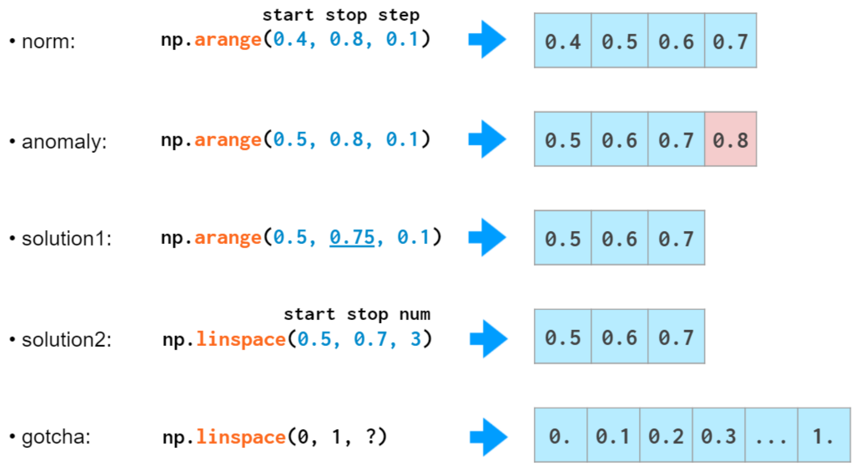


There are as many as two functions for array initialization with a monotonic sequence in NumPy:

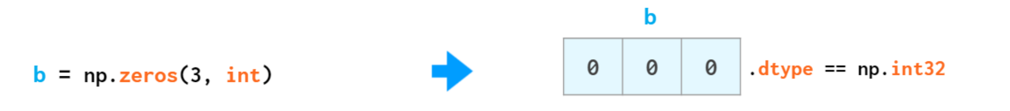


If you need a similar-looking array of floats, like `[0., 1., 2.]` , you can change the type of the `arange` output: `arange(3).astype(float)` , but there' s a better way. The `arange` function is type-sensitive: If you feed ints as arguments, it will generate ints, and if you feed floats (e.g., `arange(3.)`) it will generate floats.

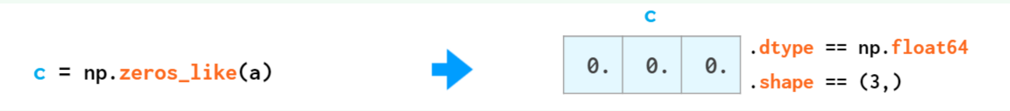
But `arange` is not especially good at handling floats:



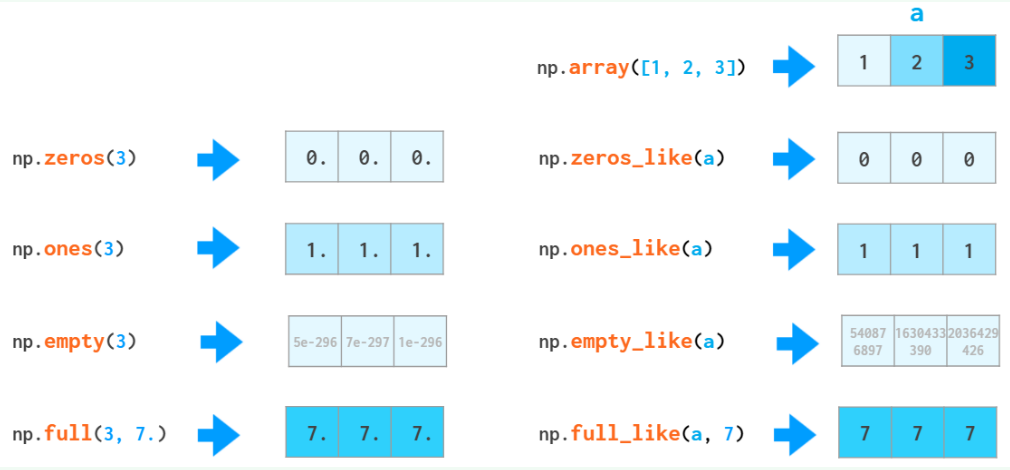
This `0.1` looks like a finite decimal number to us but not to the computer: In binary, it is an infinite fraction and has to be rounded somewhere thus an error. That' s why feeding a step with fractional part to `arange` is generally a bad idea: You might run into an off-by-one error. You can make an end of the interval fall into a non-integer number of steps (solution1) but that reduces readability and maintainability. This is where `linspace` might come in handy. It is immune to rounding errors and always generates the number of elements you ask for. There' s a common gotcha with `linspace` , though. It counts points, not intervals, thus the last argument is always plus one to what you would normally think of. So it is 11, not 10 in the example above.



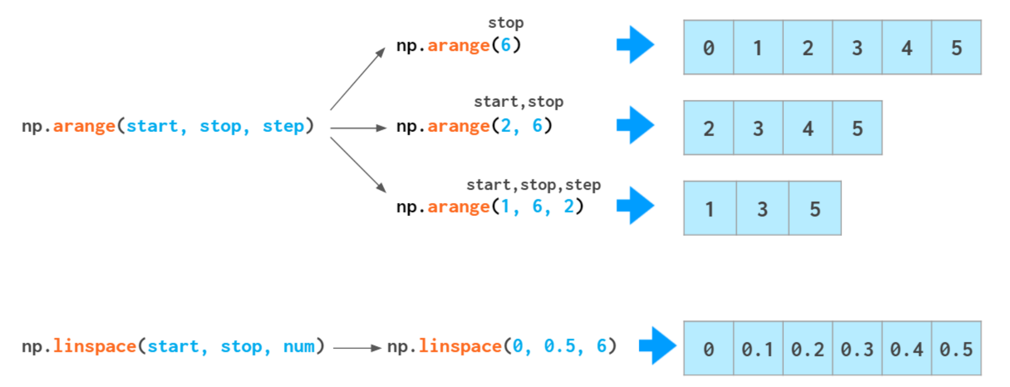
通过以下方法可以创建一个与某一变量形状一致的空数组：



不止是空数组，通过上述方法还可以将数组填充为特定值：

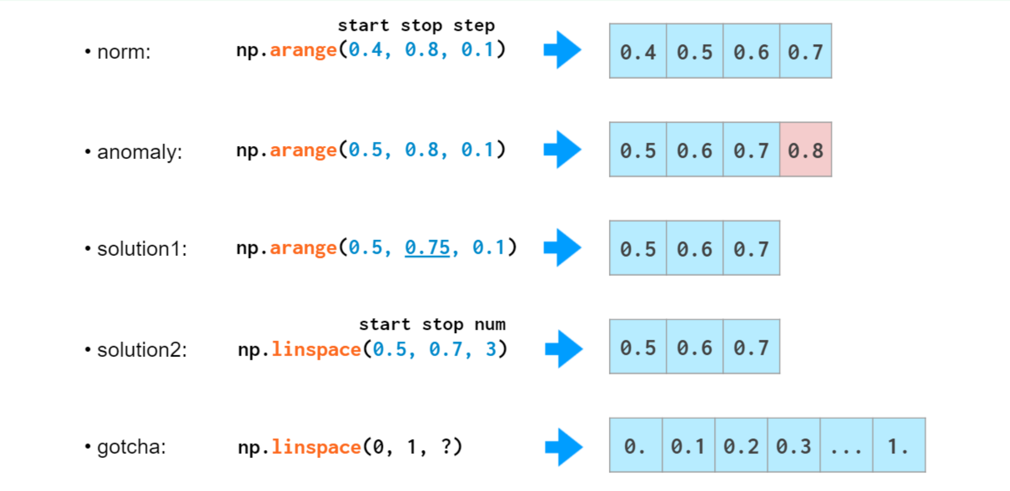


在NumPy中，还可以通过单调序列初始化数组：



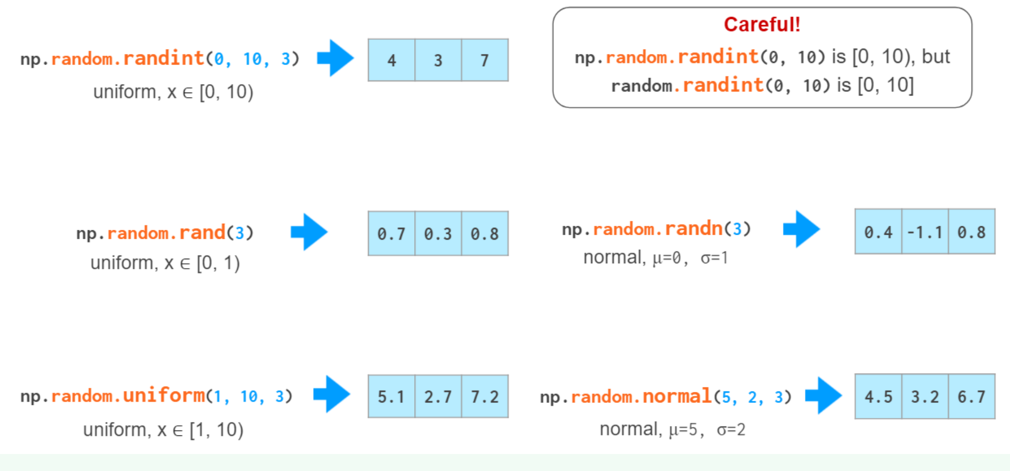
如果您需要[0., 1., 2.]这样的浮点数组，可以更改arange输出的类型，即arange(3).astype(float)，但有更好的方法：由于arange函数对类型敏感，因此参数为整数类型，它生成的也是整数类型，如果输入float类型arange(3.)，则会生成浮点数。

arange浮点类型数据不是非常友好：

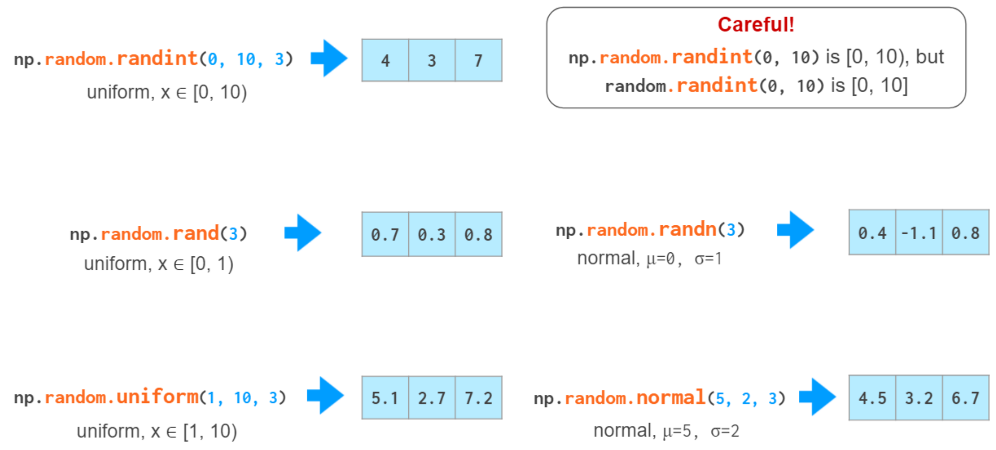


上图中，0.1对我们来说是一个有限的十进制数，但对计算机而言，它是一个二进制无穷小数，必须四舍五入为一个近似值。因此，将小数作为arange的步长可能导致一些错误。可以通过以下两种方式避免如上错误：一是使间隔末尾落入非整数步数，但这会降低可读性和可维护性；二是使用linspace，这样可以避免四舍五入的错误影响，并始终生成要求数量的元素。但使用linspace时尤其需要注意最后一个的数量参数设置，由于它计算点数量，而不是间隔数量，因此上图中数量参数是11，而不是10。

随机数组的生成如下：



For testing purposes it is often necessary to generate random arrays:

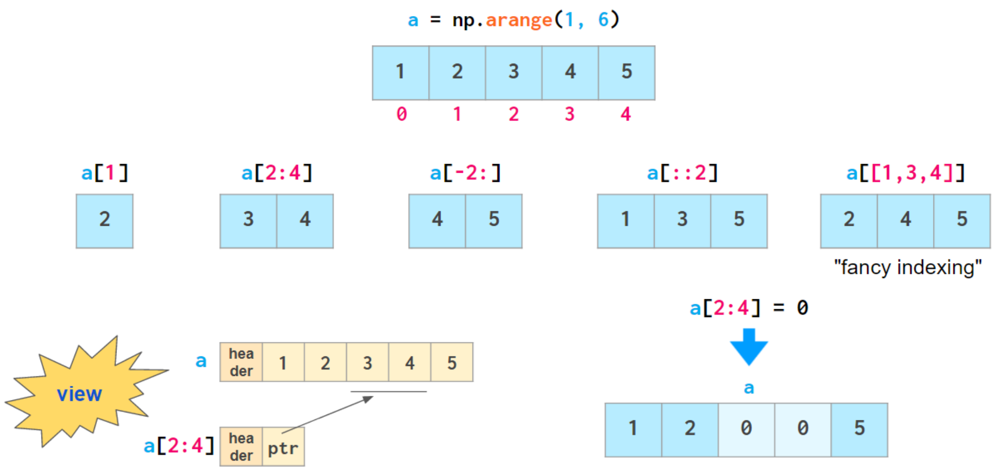


Careful!
np.random.randint(0, 10) is [0, 10), but
random.randint(0, 10) is [0, 10]

04

Vector indexing

Once you have your data in the array, NumPy is brilliant at providing easy ways of giving it back:

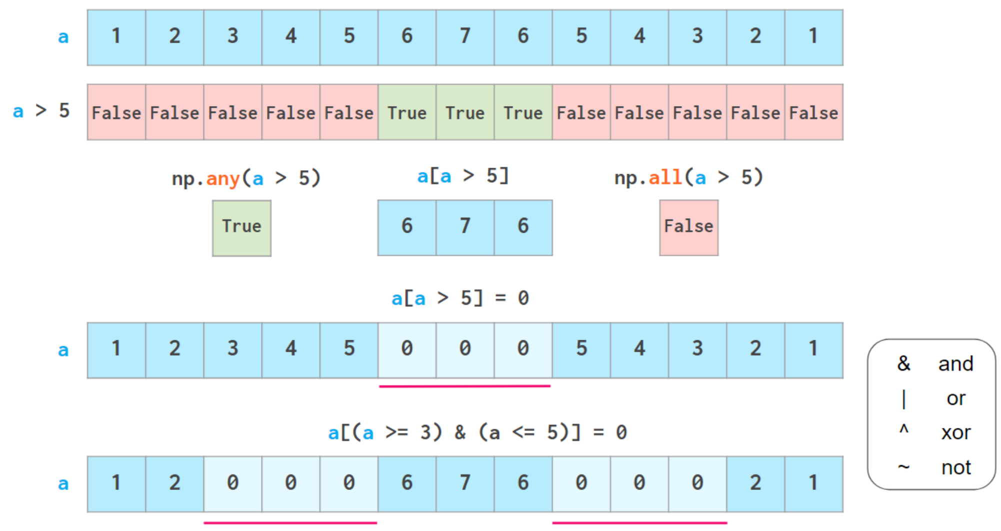


All of the indexing methods presented above except fancy indexing are actually so-called “views”: They don’t store the data and reflect the changes in the original array if it happens to get changed after being indexed.

All of those methods including fancy indexing are mutable: They allow modification of the original array contents through assignment, as shown above. This feature breaks the habit of copying arrays by slicing them:

```
            python list                    vs               numpy array

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

Another super-useful way of getting data from NumPy arrays is boolean indexing, which allows using all kinds of logical operators:

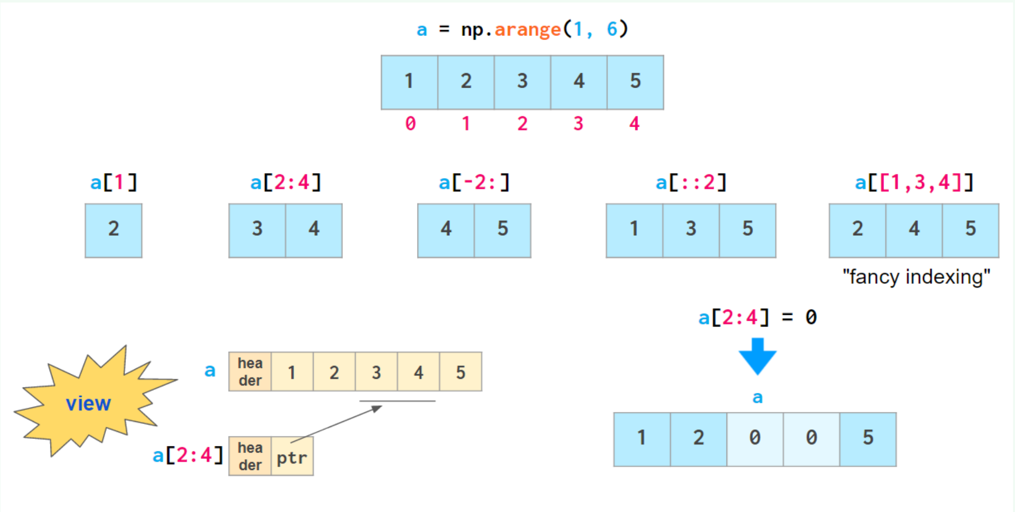


any and all act just like their python peers, but don’t short-circuit

Careful though; Python “ternary” comparisons like 3<=a<=5 don’t work here.

向量索引

对于数组数据的访问，numpy提供了便捷的访问方式：

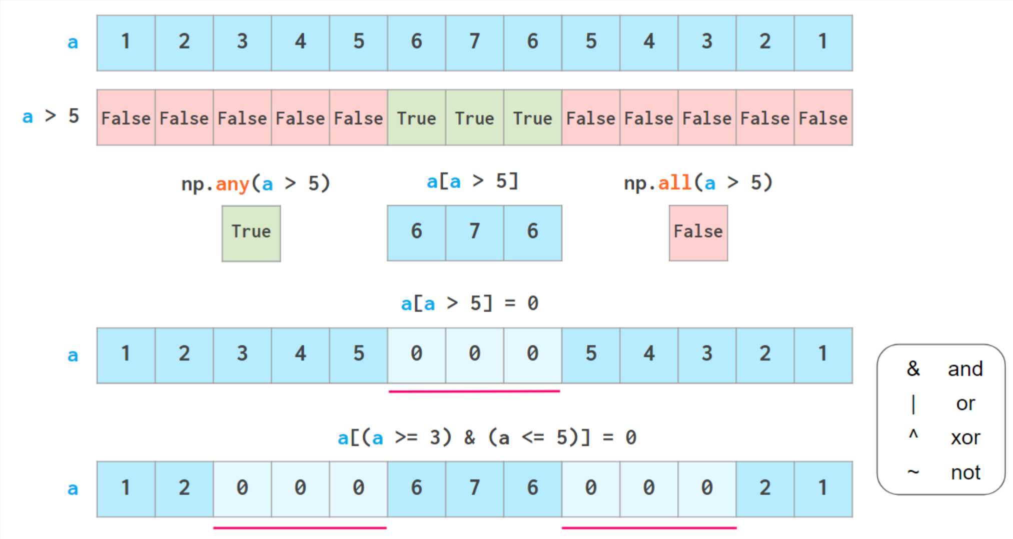


上图中，除“fancy indexing”外，其他所有索引方法本质上都是`views`：它们并不存储数据，如果原数组在被索引后发生更改，则会反映出原始数组中的更改。

上述所有这些方法都可以改变原始数组，即允许通过分配新值改变原数组的内容。这导致无法通过切片来复制数组：

```
          python list                 vs               numpy array

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

此外，还可以通过布尔索引从NumPy数组中获取数据，这意味着可以使用各种逻辑运算符：

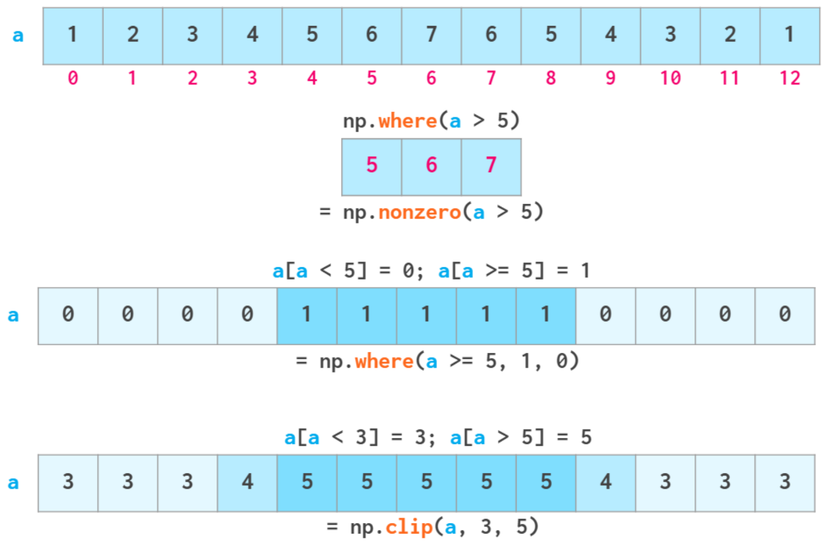


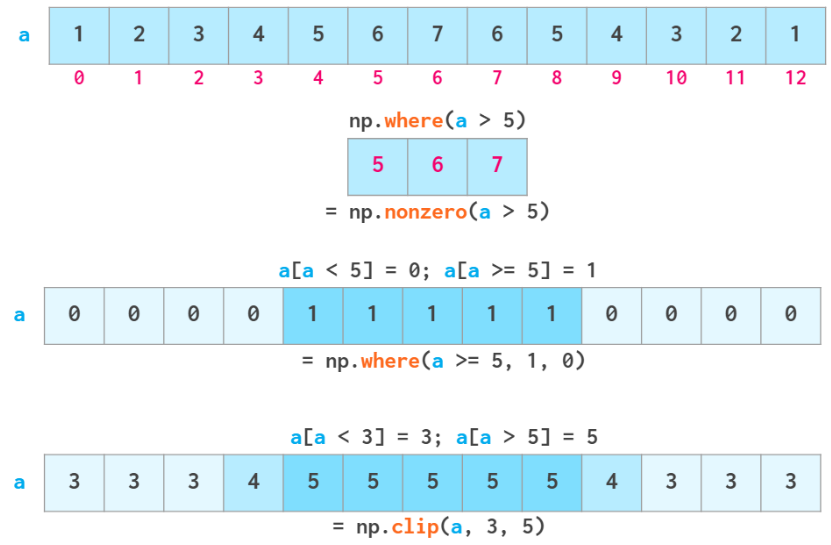
any和all与其他Python使用类似

注意，不可以使用`3 <= a <= 5`这样的Python “三元”比较。

如上所述，布尔索引是可写的。如下图np.where和np.clip两个专有函数。

As seen above, boolean indexing is also writable. Two common use cases of it spun off as dedicated functions: the excessively overloaded `np.where` function (see both meanings below) and `np.clip` .

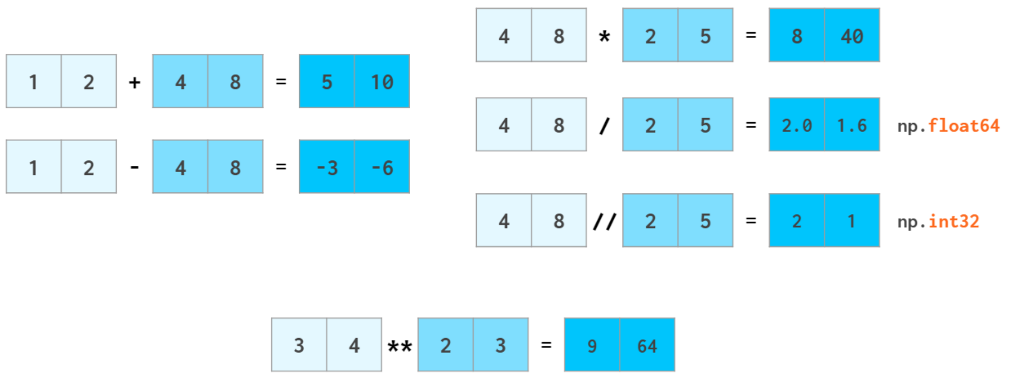




05

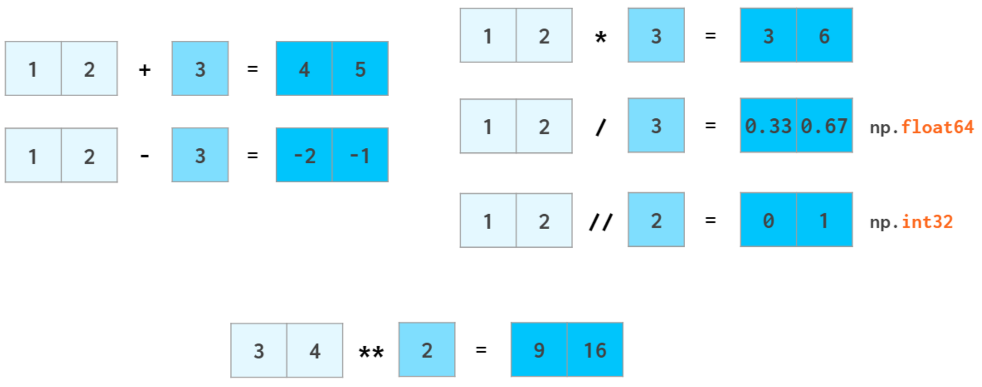
Vector operations

Arithmetic is one of the places where NumPy speed shines most. Vector operators are shifted to the c++ level and allow us to avoid the costs of slow Python loops. NumPy allows the manipulation of whole arrays just like ordinary numbers:

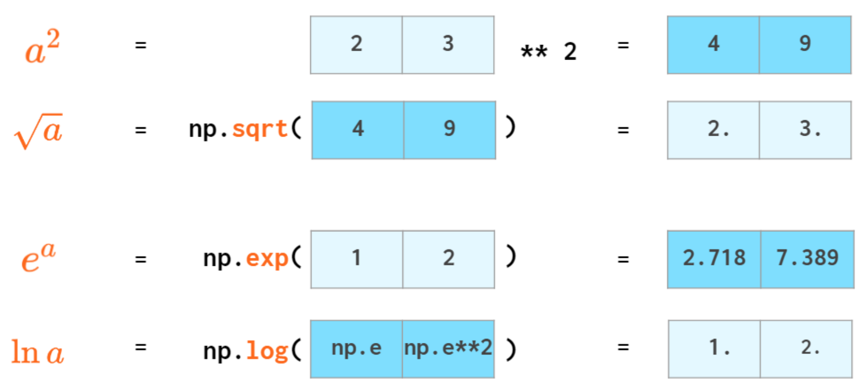


As usual in Python, a//b means a div b (quotient from division), x**n means x^n

The same way ints are promoted to floats when adding or subtracting, scalars are promoted (aka broadcasted) to arrays:



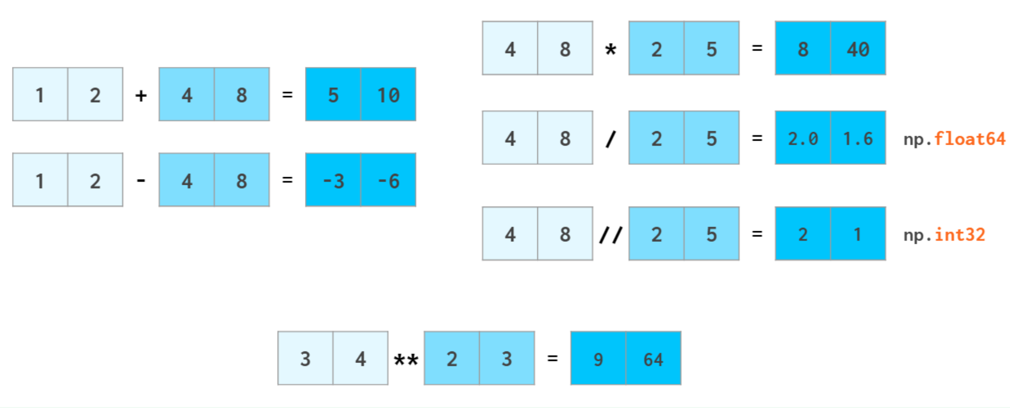
Most of the math functions have NumPy counterparts that can handle vectors:


a^2 = 2 3 ** 2 = 4 9
$\sqrt{a}$ = np.sqrt(4 9) = 2. 3.
e^a = np.exp(1 2) = 2.718 7.389
$\ln a$ = np.log(np.e np.e**2) = 1. 2.

Scalar product has an operator of its own:

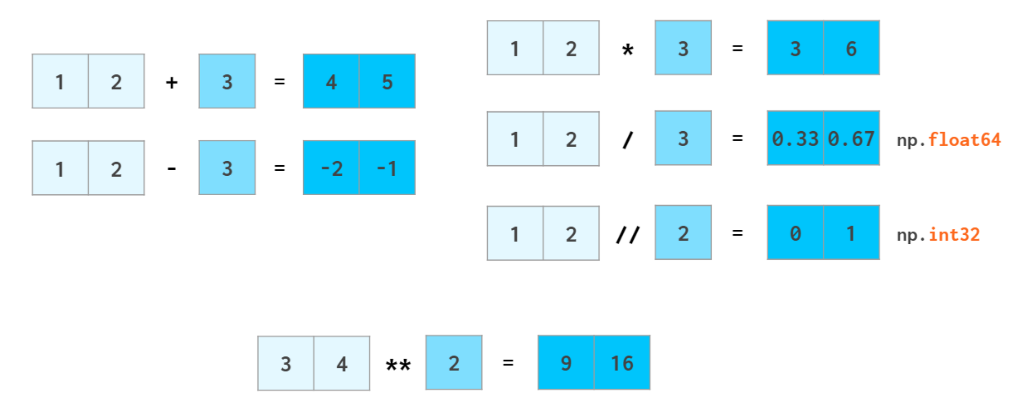
向量操作

NumPy的计算速度是其亮点之一，其向量运算操作接近C++级别，避免了Python循环耗时较多的问题。NumPy允许像普通数字一样操作整个数组：

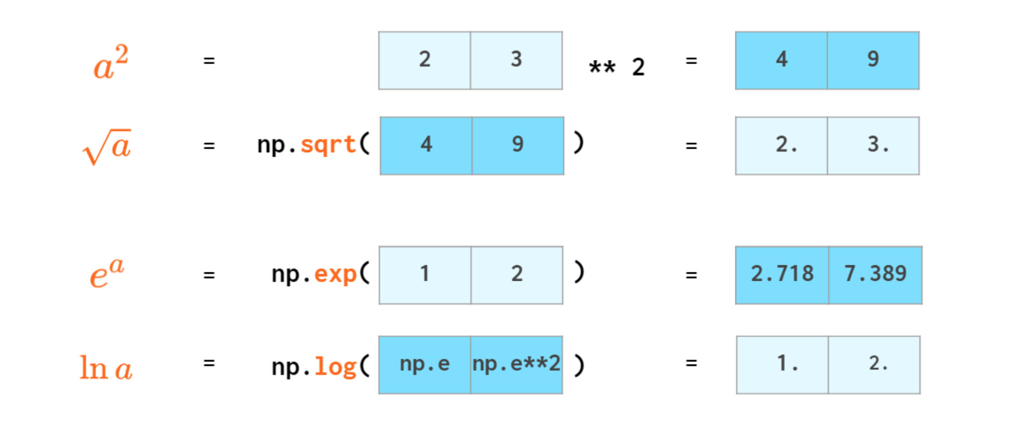


在python中，a//b表示a div b（除法的商），x**n表示 x^n

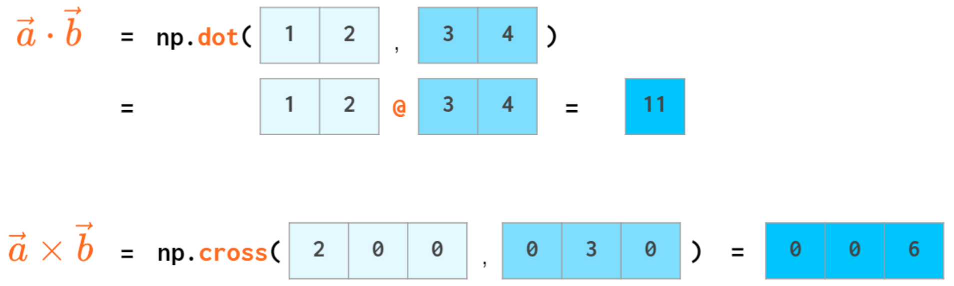
浮点数的计算也是如此，numpy能够将标量广播到数组：



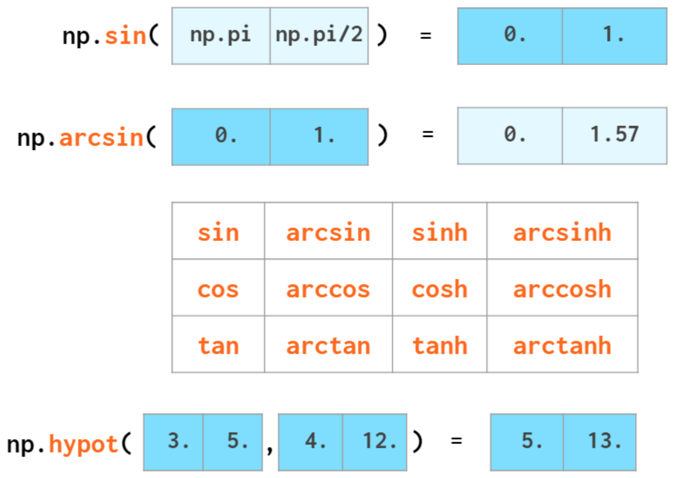
numpy提供了许多数学函数来处理矢量：


a^2 = 2 3 ** 2 = 4 9
$\sqrt{a}$ = np.sqrt(4 9) = 2. 3.
e^a = np.exp(1 2) = 2.718 7.389
$\ln a$ = np.log(np.e np.e**2) = 1. 2.

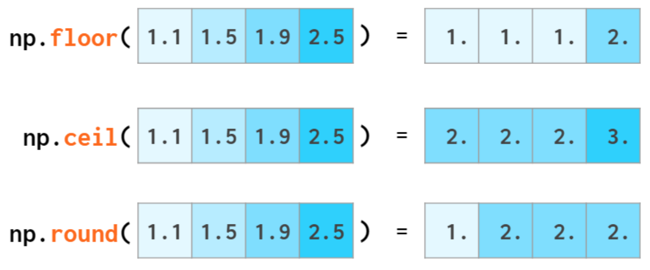
向量点乘（内积）和叉乘（外积、向量积）如下：


$\vec{a}\cdot\vec{b}$ = np.dot(1 2 , 3 4)
= 1 2 @ 3 4 = 11
$\vec{a}\times\vec{b}$ = np.cross(2 0 0 , 0 3 0) = 0 0 6

You don’ t need loops for trigonometry either:


np.sin(np.pi np.pi/2) = 0. 1.
np.arcsin(0. 1.) = 0. 1.57
sin arcsin sinh arcsinh
cos arccos cosh arccosh
tan arctan tanh arctanh
np.hypot(3. 5. , 4. 12.) = 5. 13.

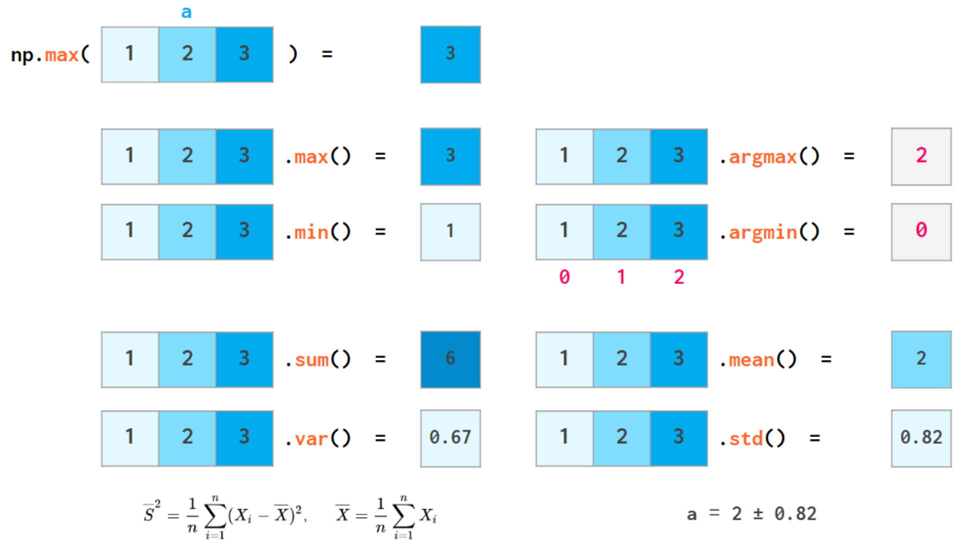
Arrays can be rounded as a whole:


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.

floor rounds to -∞, ceil to +∞ and around — to the nearest integer (.5 to even)

The name np.around is just an alias to np.round introduced to avoid shadowing Python round when you write from numpy import * (as opposed to a more common import numpy as np). You can use a.round() as well.

NumPy is also capable of doing the basic stats:

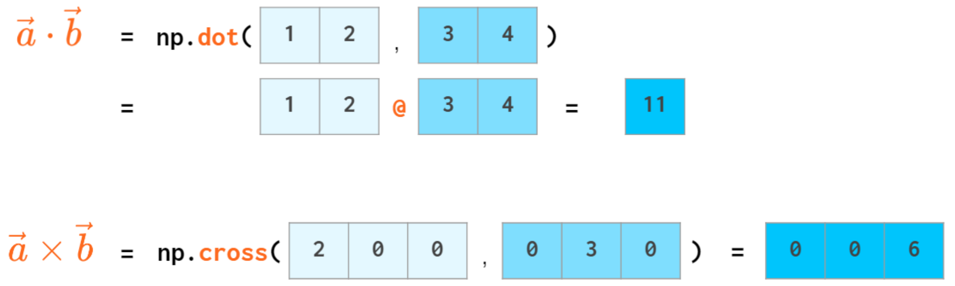


Each of these functions has a nan-resistant variant: eg nansum, nanmax, etc

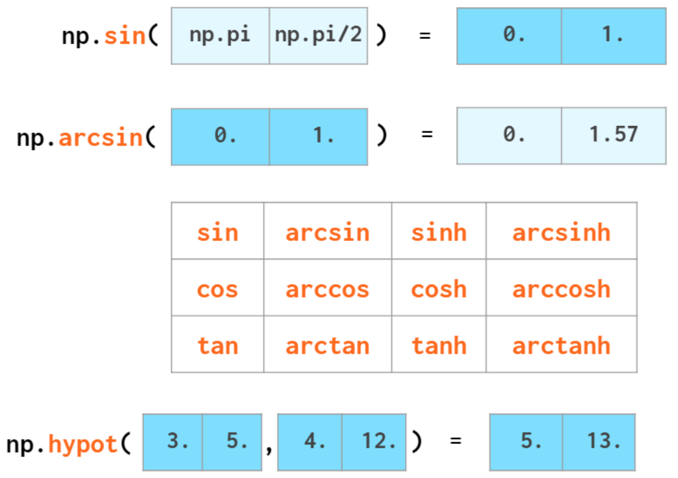
Sorting functions have fewer capabilities than their Python counterparts:

python lists	numpy arrays	
a.sort()	a.sort()	sorts in-place
sorted(a)	np.sort(a)	returns new sorted array
a.sort(key=f)	–	key function
a.sort(reversed=False)	–	ascending/descending

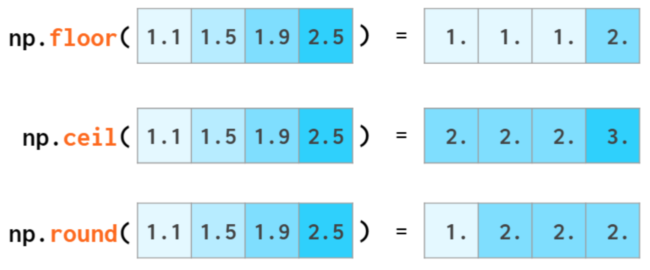
In the 1D case, the absence of the reversed keyword can be easily compensated for by reversing the result. In 2D it is somewhat trickier (feature request[5]).


$\vec{a}\cdot\vec{b}$ = np.dot(1 2 , 3 4)
= 1 2 @ 3 4 = 11
$\vec{a}\times\vec{b}$ = np.cross(2 0 0 , 0 3 0) = 0 0 6

numpy也提供了如下三角函数运算：


np.sin(np.pi np.pi/2) = 0. 1.
np.arcsin(0. 1.) = 0. 1.57
sin arcsin sinh arcsinh
cos arccos cosh arccosh
tan arctan tanh arctanh
np.hypot(3. 5. , 4. 12.) = 5. 13.

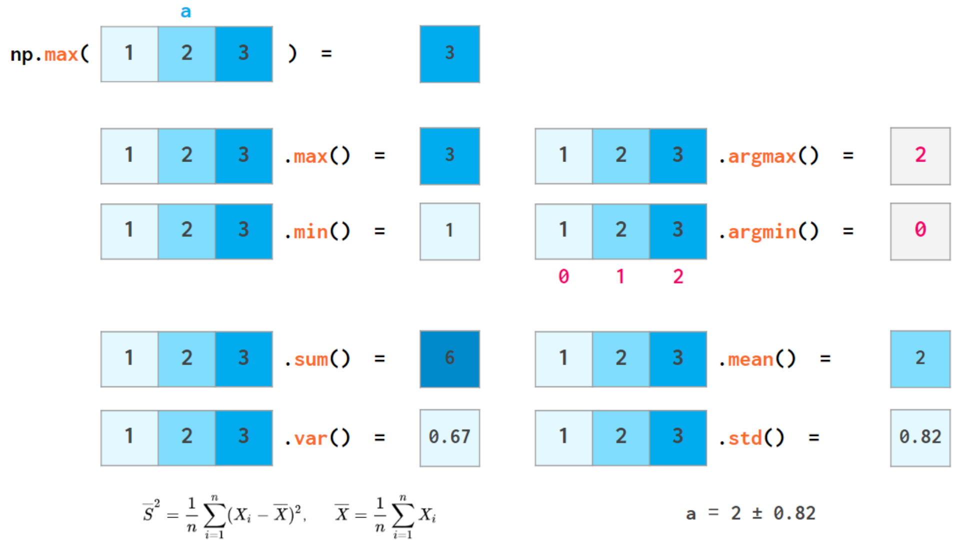
数组整体进行四舍五入：


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.

floor向上取整，ceil向下取整，round四舍五入

np.around与np.round是等效的，这样做只是为了避免 from numpy import * 时与Python aroun的冲突（但一般的使用方式是import numpy as np）。当然，你也可以使用a.round()。

numpy还可以实现以下功能：



以上功能都存在相应的nan-resistant变体：例如nansum，nanmax等

在numpy中，排序函数功能有所阉割：

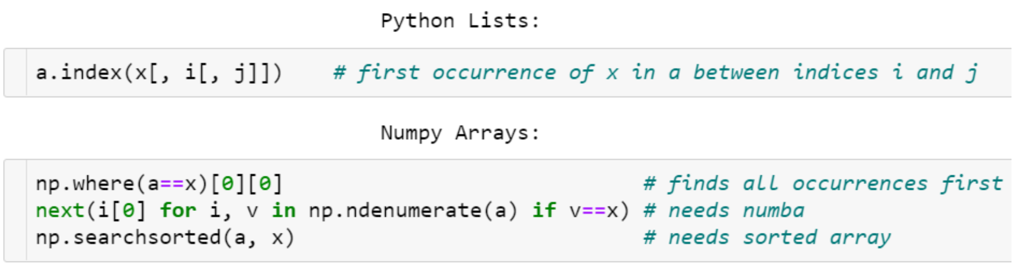
python lists	numpy arrays	
a.sort()	a.sort()	sorts in-place
sorted(a)	np.sort(a)	returns new sorted array
a.sort(key=f)	–	key function
a.sort(reversed=False)	–	ascending/descending

对于一维数组，可以通过反转结果来解决reversed函数缺失的不足，但在2维数组中该问题变得棘手。

06

Searching for an element in a vector

As opposed to Python lists, a NumPy array does not have an index method. The corresponding feature request[6] has been hanging there for quite a while now.



```
Python Lists:
a.index(x[, i[, j]])    # first occurrence of x in a between indices i and j

Numpy Arrays:
np.where(a==x)[0][0]                            # finds all occurrences first
next(i[0] for i, v in np.ndenumerate(a) if v==x) # needs numba
np.searchsorted(a, x)                           # needs sorted array
```

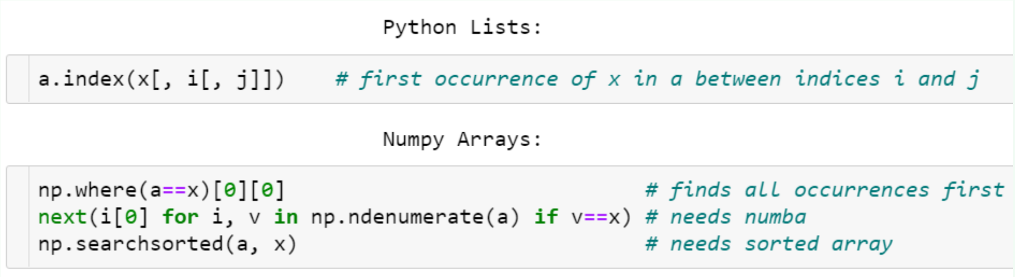
The square brackets in the definition of index() mean that either j or both i and j can be omitted

- One way of finding an element is np.where(a==x)[0][0] , which is neither elegant nor fast as it needs to look through all elements of the array even if the item to find is in the very beginning.
- A faster way to do it is via accelerating next((i[0] for i, v in np.ndenumerate(a) if v==x), -1) with Numba[7] (otherwise it' s way slower in the worst case than where).
- Once the array is sorted though, the situation gets better: v = np.searchsorted(a, x); return v if a[v]==x else -1 is really fast with O(log N) complexity, but it requires O(N log N) time to sort first.

Actually, it is not a problem to speed up searching by implementing it in C. The problem is float comparisons. This is a task that simply does not work out of the box for arbitrary data.

查找向量中的元素

不同于Python列表，NumPy数组没有索引方法。



```
Python Lists:
a.index(x[, i[, j]])    # first occurrence of x in a between indices i and j

Numpy Arrays:
np.where(a==x)[0][0]                            # finds all occurrences first
next(i[0] for i, v in np.ndenumerate(a) if v==x) # needs numba
np.searchsorted(a, x)                           # needs sorted array
```

index()中的方括号表示j或i&j可以省略

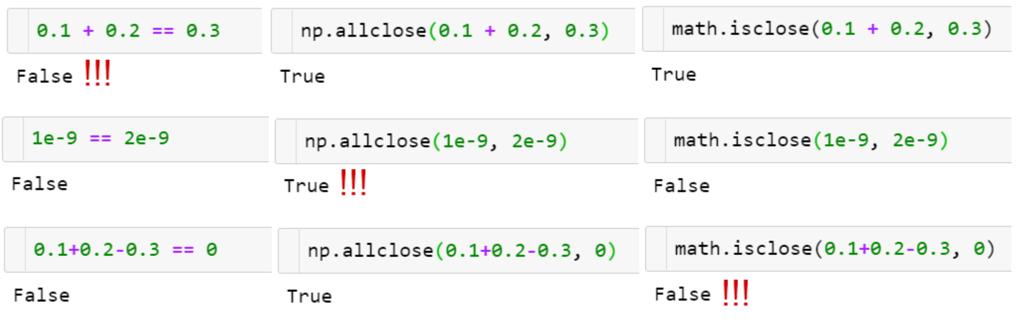
- 可以通过np.where(a==x)[0] [0]查找元素，但这种方法很不pythonic，哪怕需要查找的项在数组开头，该方法也需要遍历整个数组。
- 使用Numba实现加速查找，next((i[0] for i, v in np.ndenumerate(a) if v==x), -1)，在最坏的情况下，它的速度要比where慢。
- 如果数组是排好序的，使用v = np.searchsorted(a, x); return v if a[v]==x else -1时间复杂度为O(log N)，但在这之前，排序的时间复杂度为O(N log N)。

实际上，通过C实现加速搜索并不是困难，问题是浮点数据比较。

07

Comparing floats

The function np.allclose(a, b) compares arrays of floats with a given tolerance



```
0.1 + 0.2 == 0.3
False !!!

np.allclose(0.1 + 0.2, 0.3)
True

math.isclose(0.1 + 0.2, 0.3)
True

1e-9 == 2e-9
False

np.allclose(1e-9, 2e-9)
True !!!

math.isclose(1e-9, 2e-9)
False

0.1+0.2-0.3 == 0
False

np.allclose(0.1+0.2-0.3, 0)
True

math.isclose(0.1+0.2-0.3, 0)
False !!!
```

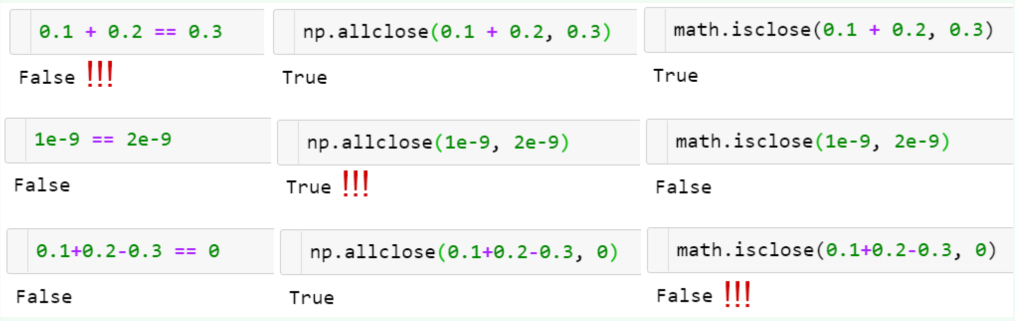
There is no silver bullet!

- np.allclose assumes all the compared numbers to be of a typical scale of 1. For example, if you work with nanoseconds, you need to divide the default atol argument value by 1e9: np.allclose(1e-9, 2e-9, atol=1e-17) == False .
- math.isclose makes no assumptions about the numbers to be compared but relies on a user to give a reasonable abs_tol value instead (taking the default np.allclose atol value of 1e-8 is good enough for numbers with a typical scale of 1): math.isclose(0.1+0.2–0.3, abs_tol=1e-8)==True .

Aside from that, np.allclose has some minor issues in a formula for absolute and relative tolerances, for example, for certain a,b allclose(a, b) != allclose(b, a) . Those issues were resolved in the (scalar)

浮点数比较

np.allclose(a, b) 用于容忍误差之内的浮点数比较。



```
0.1 + 0.2 == 0.3
False !!!

np.allclose(0.1 + 0.2, 0.3)
True

math.isclose(0.1 + 0.2, 0.3)
True

1e-9 == 2e-9
False

np.allclose(1e-9, 2e-9)
True !!!

math.isclose(1e-9, 2e-9)
False

0.1+0.2-0.3 == 0
False

np.allclose(0.1+0.2-0.3, 0)
True

math.isclose(0.1+0.2-0.3, 0)
False !!!
```

- np.allclose假定所有比较数字的尺度为1。如果在纳秒级别上，则需要将默认atol参数除以1e9：np.allclose(1e-9,2e-9, atol=1e-17)==False。
- math.isclose不对要比较的数字做任何假设，而是需要用户提供一个合理的abs_tol值（np.allclose默认的atol值1e-8足以满足小数位数为1的浮点数比较，即math.isclose(0.1+0.2–0.3, abs_tol=1e-8)==True。

此外，对于绝队偏差和相对偏差，np.allclose依然存在一些问题。例如，对于某些值a、b， allclose(a,b)!=allclose(b,a)，而在math.isclose中则不存在这些问题。查看GitHub上的浮点数据指南和相应的NumPy问题了解更多信息。

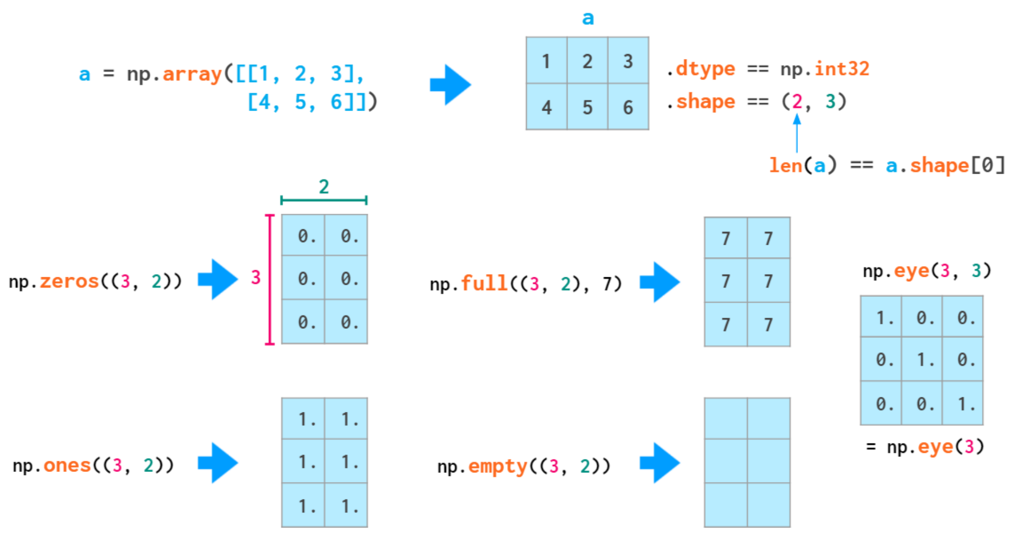
function math.isclose (which was introduced later). To learn more on that, take a look at the excellent floating-point guide[8] and the corresponding NumPy issue[9] on GitHub.

08

2. Matrices, the 2D Arrays

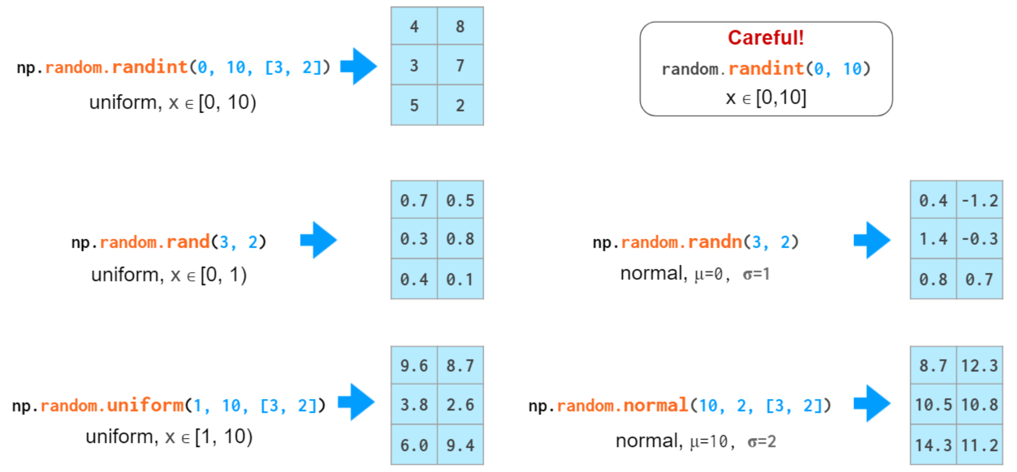
There used to be a dedicated matrix class in NumPy, but it is deprecated now, so I' ll use the words matrix and 2D array interchangeably.

Matrix initialization syntax is similar to that of vectors:

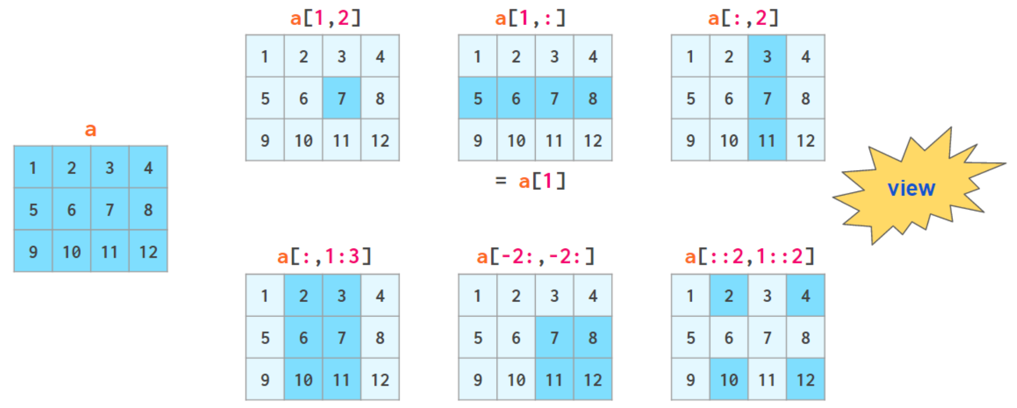


Double parentheses are necessary here because the second positional parameter is reserved for the (optional) dtype (which also accepts integers).

Random matrix generation is also similar to that of vectors:



Two-dimensional indexing syntax is more convenient than that of nested lists:

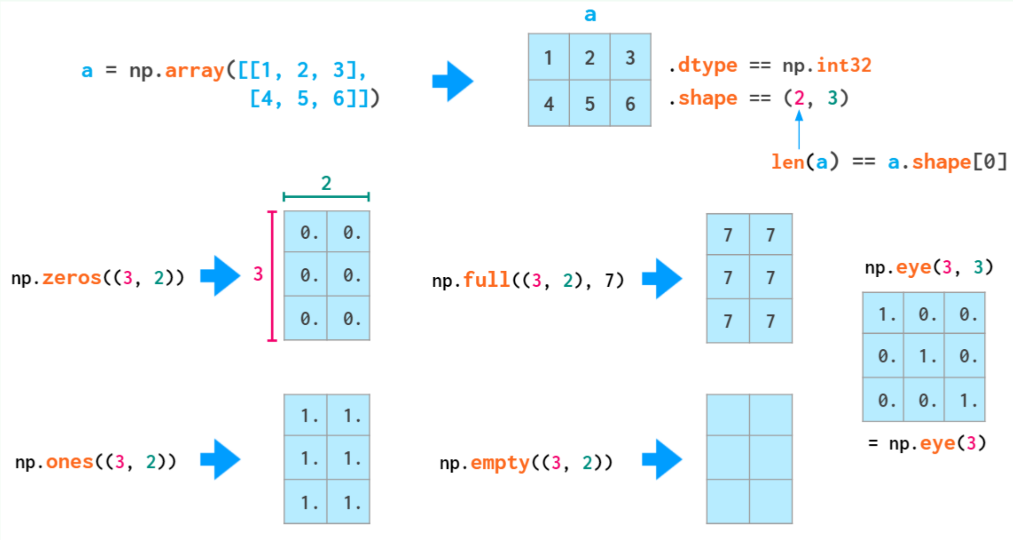


The "view" sign means that no copying is actually done when slicing an array. When the array is modified, the changes are reflected in the slice as well.

2.矩阵和二维数组

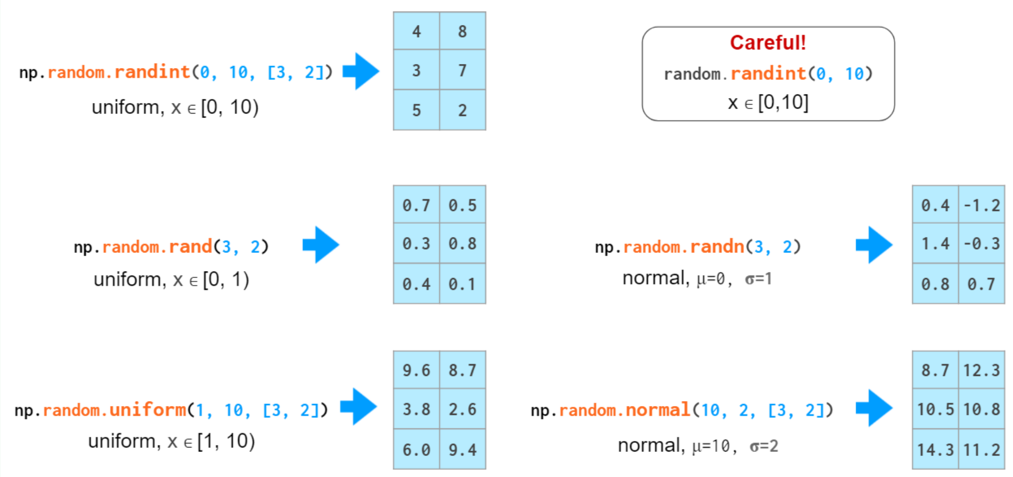
过去，NumPy中曾有一个专用的matrix类，但现在已被弃用，因此在下文中矩阵和2维数组表示同一含义。

矩阵的初始化语法与向量类似：

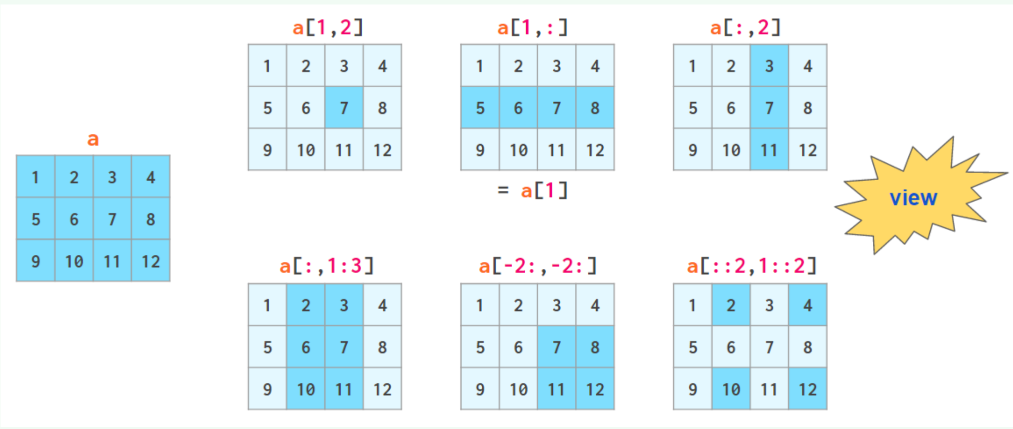


如上要使用双括号，因为第二个位置参数（可选）是为dtype（也接受整数）保留的。

随机矩阵的生成也与向量类似：



二维数组的索引语法要比嵌套列表更方便：



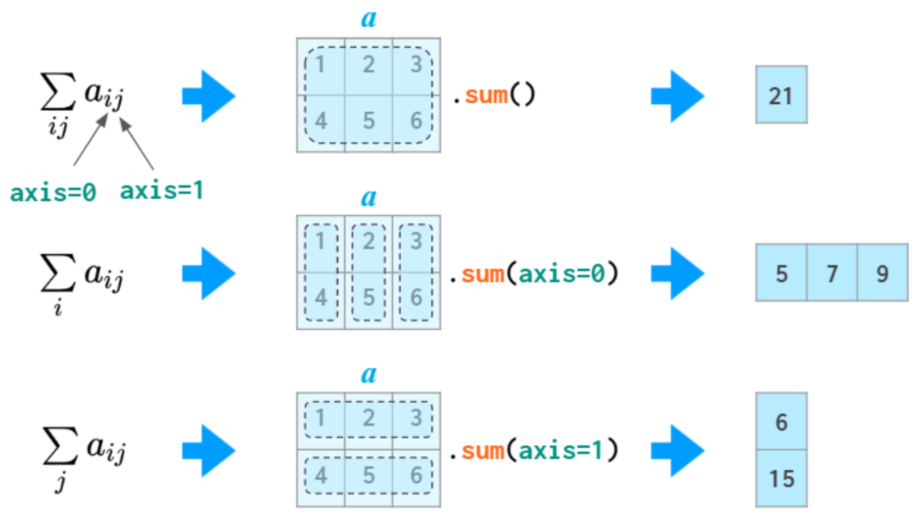
"view"表示数组切片时并未进行任何复制，在修改数组后，相应更改也将反映在切片中。

09

The axis argument

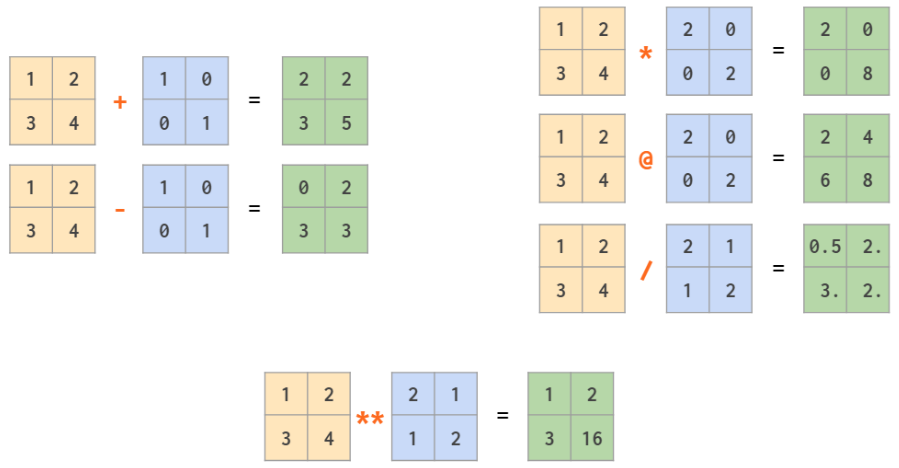
轴参数

In many operations (e.g., `sum`) you need to tell NumPy if you want to operate across rows or columns. To have a universal notation that works for an arbitrary number of dimensions, NumPy introduces a notion of axis: The value of the `axis` argument is, as a matter of fact, the number of the index in question: The first index is `axis=0`, the second one is `axis=1`, and so on. So in 2D `axis=0` is column-wise and `axis=1` means row-wise.

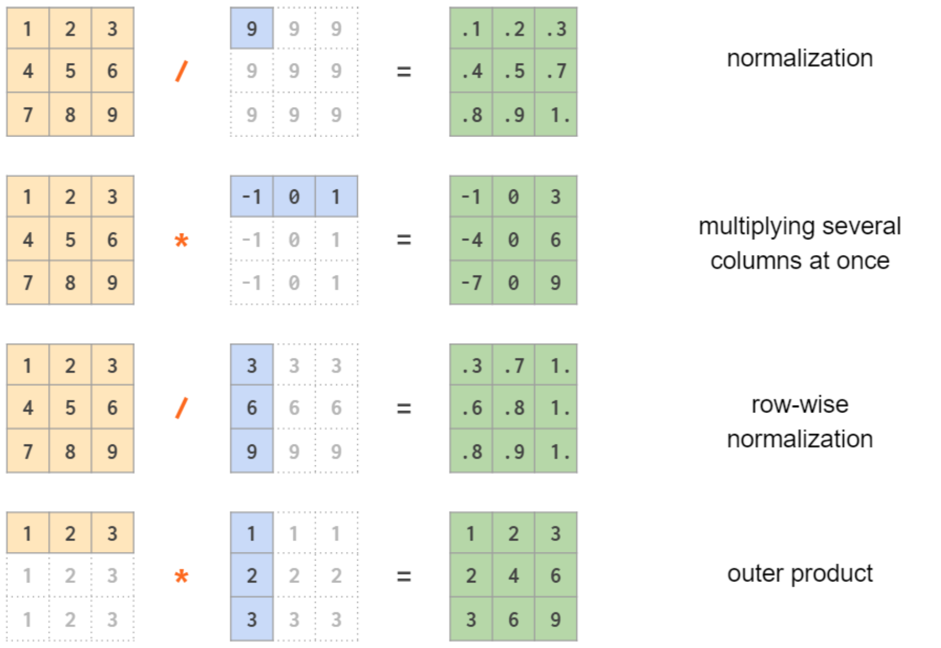


Matrix arithmetic

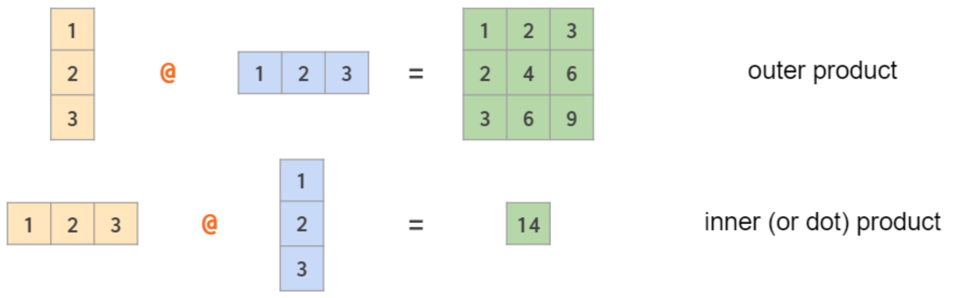
In addition to ordinary operators (like +,-,*,/,// and **) which work element-wise, there' s a @ operator that calculates a matrix product:



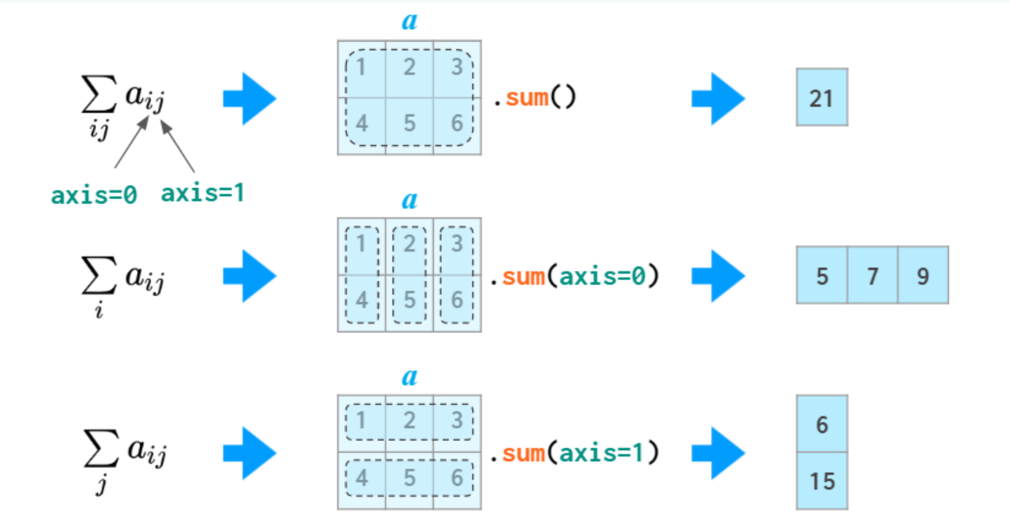
As a generalization of broadcasting from scalar that we' ve seen already in the first part, NumPy allows mixed operations between a vector and a matrix, and even between two vectors:



Note that in the last example it is a symmetric per-element multiplication. To calculate the outer product using an asymmetric linear algebra matrix multiplication the order of the operands should be reversed:

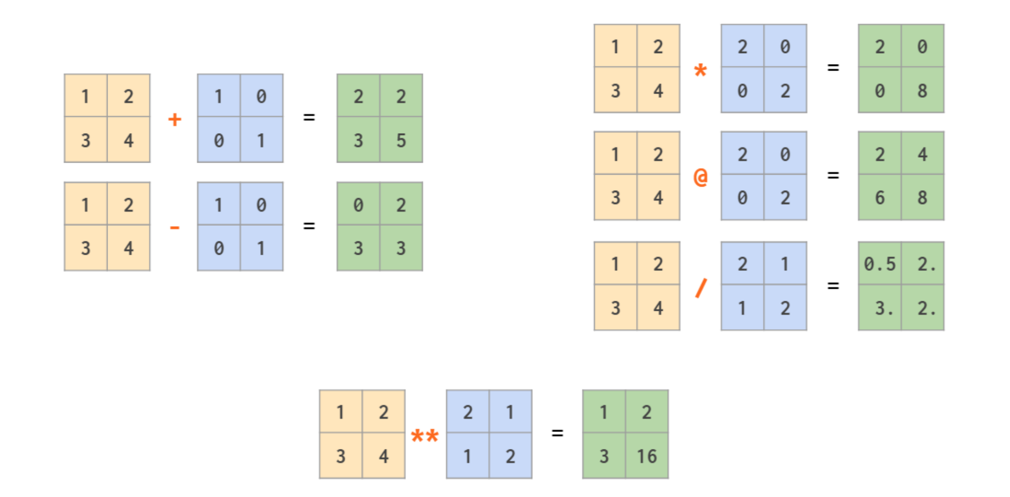


在求和等操作中，NumPy可以实现跨行或跨列的操作。为了适用任意维数的数组，NumPy引入了axis的概念。axis参数的值实际上就是维度数量，如第一个维是axis=0，第二维是axis=1，依此类推。因此，在2维数组中，axis=0指列方向，axis=1指行方向。

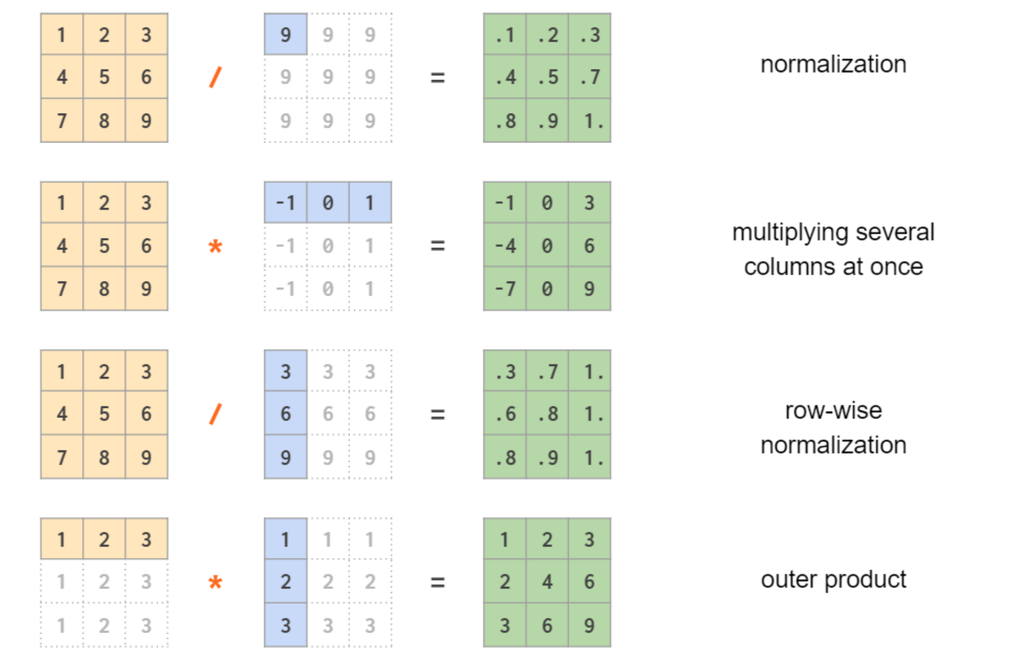


矩阵运算

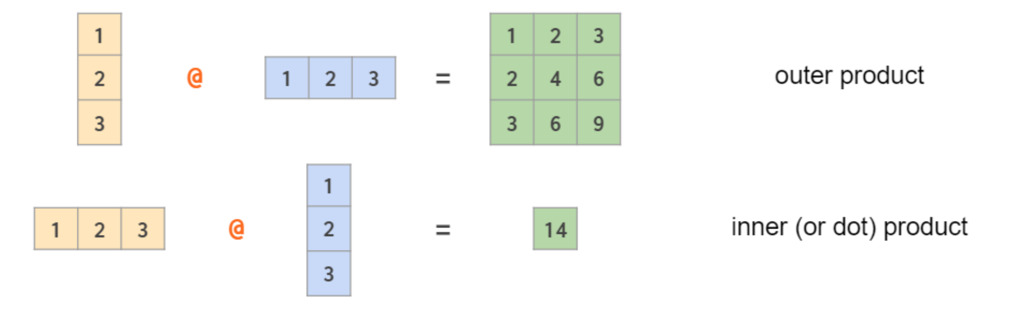
除了+，-，*，/，//和**等数组元素的运算符外，numpy提供了@运算符计算矩阵乘积：



类似前文介绍的标量广播机制，numpy同样可以通过广播机制实现向量与矩阵，或两个向量之间的混合运算：

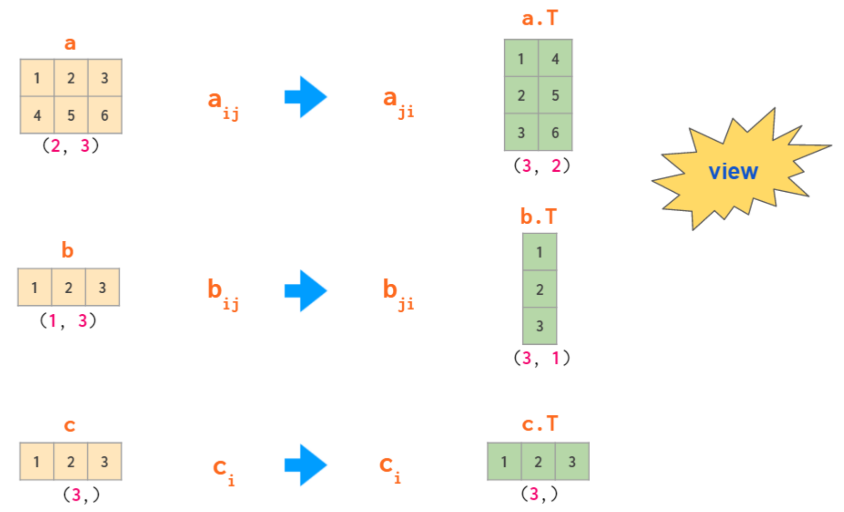


注意，上图最后一个示例是对称的逐元素乘法。使用矩阵乘法@可以计算非对称线性代数外积，两个矩阵互换位置后计算内积：

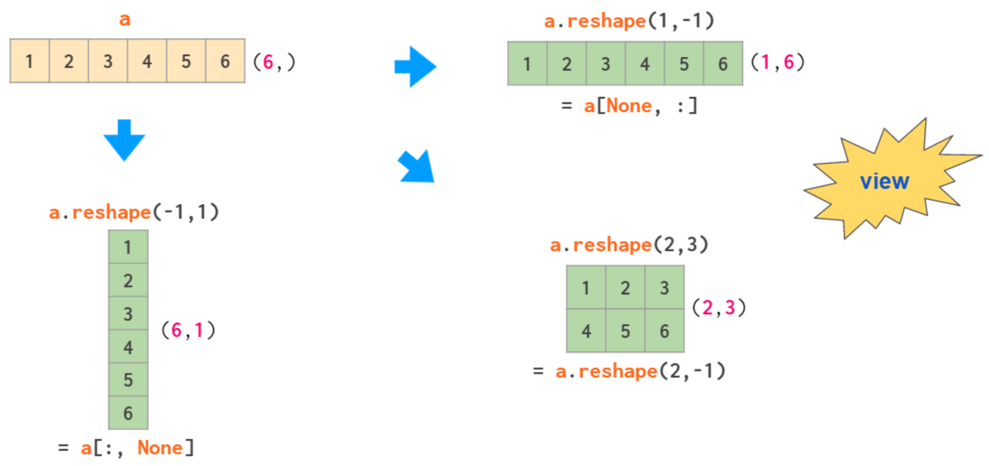


Row vectors and column vectors

As seen from the example above, in the 2D context, the row and column vectors are treated differently. This contrasts with the usual NumPy practice of having one type of 1D arrays wherever possible (e.g., `a[:, j]` — the j-th column of a 2D array `a` — is a 1D array). By default 1D arrays are treated as row vectors in 2D operations, so when multiplying a matrix by a row vector, you can use either shape (n,) or (1, n) — the result will be the same. If you need a column vector, there are a couple of ways to cook it from a 1D array, but surprisingly `transpose` is not one of them:

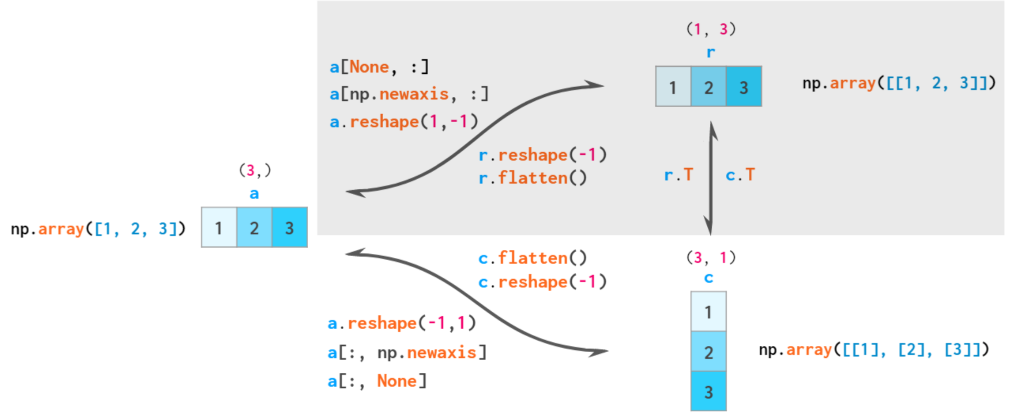


Two operations that are capable of making a 2D column vector out of a 1D array are reshaping and indexing with `newaxis`:



Here the -1 argument tells `reshape` to calculate one of the dimension sizes automatically and `None` in the square brackets serves as a shortcut for `np.newaxis`, which adds an empty axis at the designated place.

So, there’ s a total of three types of vectors in NumPy: 1D arrays, 2D row vectors, and 2D column vectors. Here’ s a diagram of explicit conversions between those:

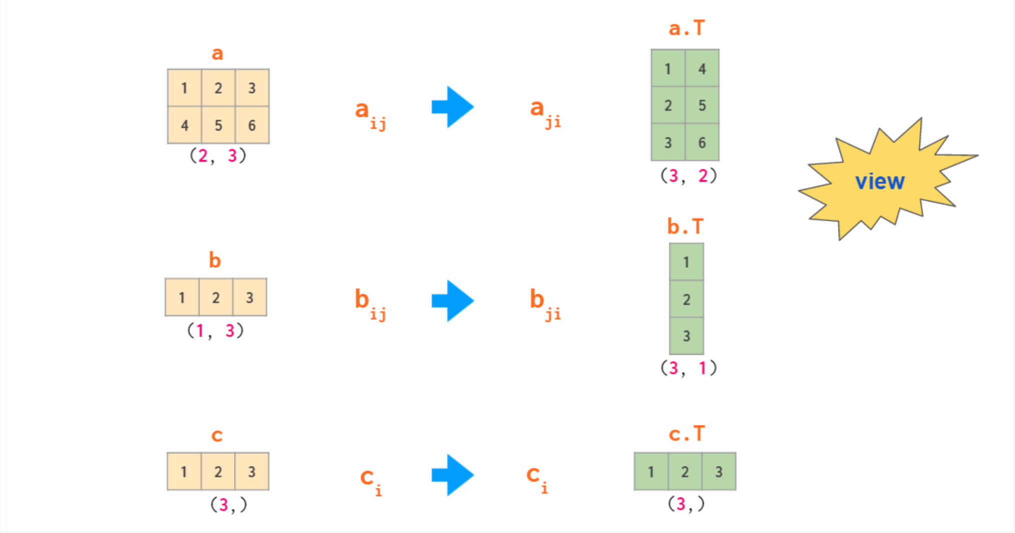


By the rules of broadcasting, 1D arrays are implicitly interpreted as 2D row vectors, so it is generally not necessary to convert between those two — thus the corresponding area is shaded.

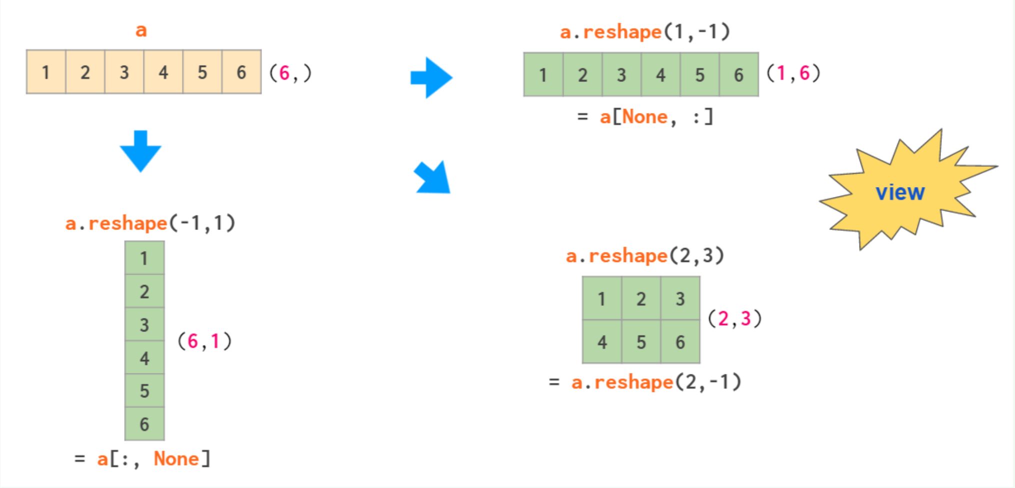
Strictly speaking, any array, all but one dimensions of which are single-sized, is a vector (eg. `a.shape==[1,1,1,5,1,1]`), so there’ s an infinite number of vector types in numpy, but only these three are commonly used. You can use `np.reshape` to convert a ‘normal’ 1D vector to this form and `np.squeeze` to get it back. Both functions act as views.

行向量与列向量

根据前文可知，在2维数组中，行向量和列向量被区别对待。通常NumPy会尽可能使用单一类型的1维数组（例如，2维数组a的第j列a[:, j]是1维数组）。默认情况下，一维数组在2维操作中被视为行向量，因此，将矩阵乘行向量时，使用形状(n,)或(1,n)的向量结果一致。有多种方法可以从一维数组中得到列向量，但并不包括transpose：

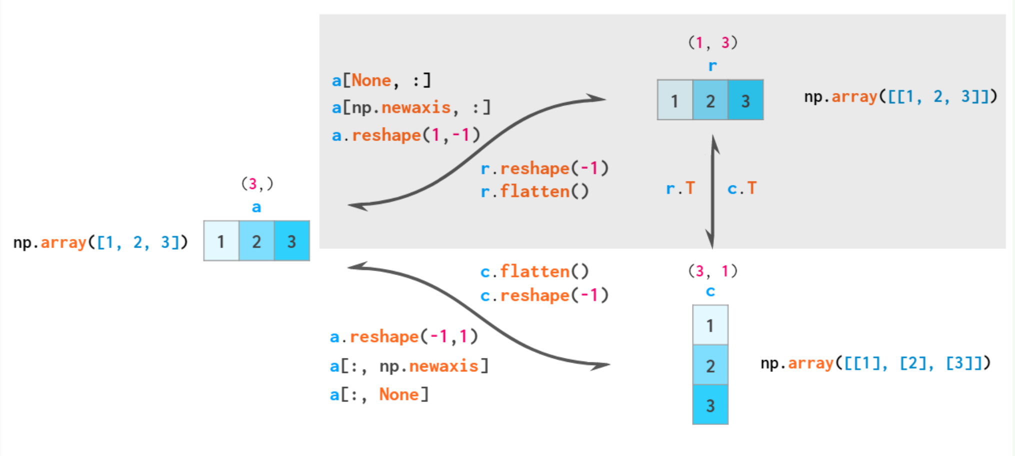


使用newaxis更新数组形状和索引可以将1维数组转化为2维列向量：



其中，-1表示在reshape是该维度自动决定，方括号中的None等同于np.newaxis，表示在指定位置添加一个空轴。

因此，NumPy中共有三种类型的向量：1维数组，2维行向量和2维列向量。以下是两两类型转换图：

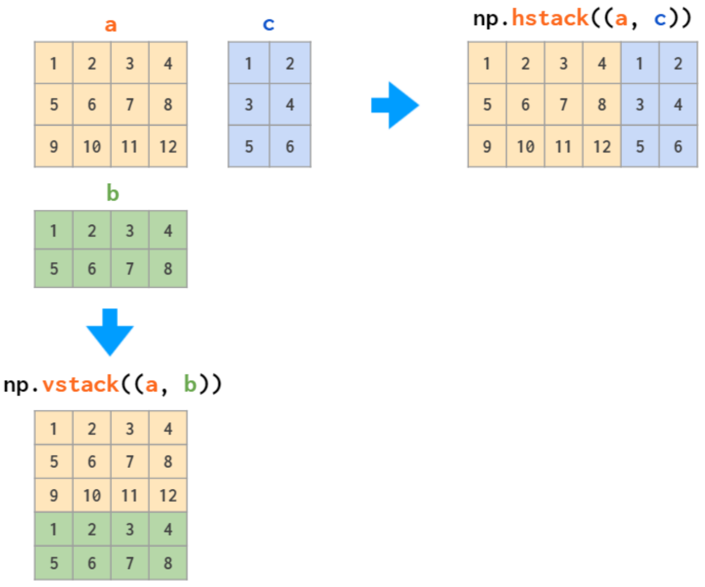


根据广播规则，一维数组被隐式解释为二维行向量，因此通常不必在这两个数组之间进行转换，对应图中阴影化区域。

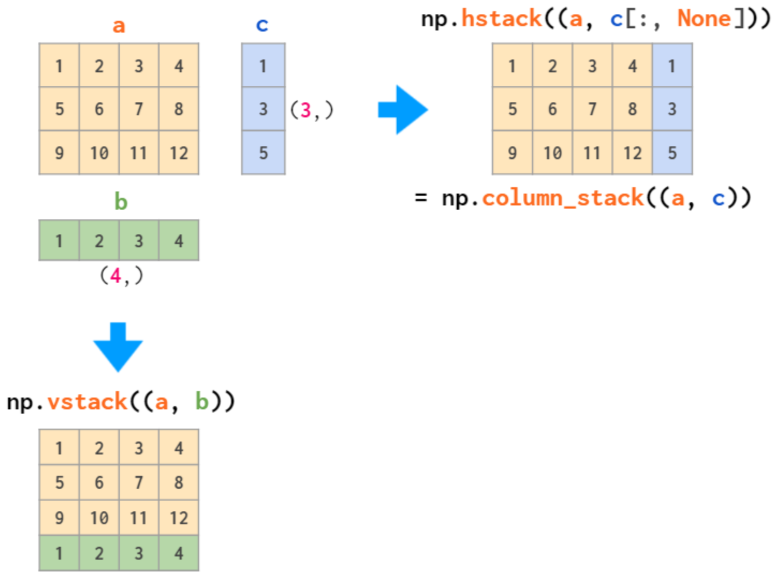
严格来说，除一维外的所有数组的大小都是一个向量（如a.shape == [1,1,1,5,1,1]），因此numpy的输入类型是任意的，但上述三种最为常用。可以使用np.reshape将一维矢量转换为这种形式，使用np.squeeze可将其恢复。这两个功能都通过view发挥作用。

Matrix manipulations

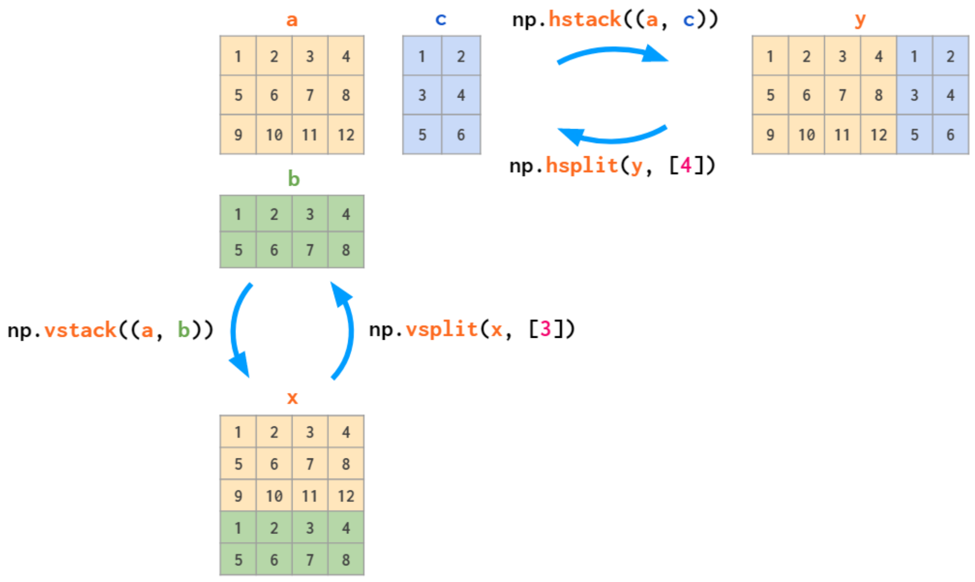
There are two main functions for joining the arrays:



Those two work fine with stacking matrices only or vectors only, but when it comes to mixed stacking of 1D arrays and matrices, only the `vstack` works as expected: The `hstack` generates a dimensions-mismatch error because as described above, the 1D array is interpreted as a row vector, not a column vector. The workaround is either to convert it to a row vector or to use a specialized `column_stack` function which does it automatically:



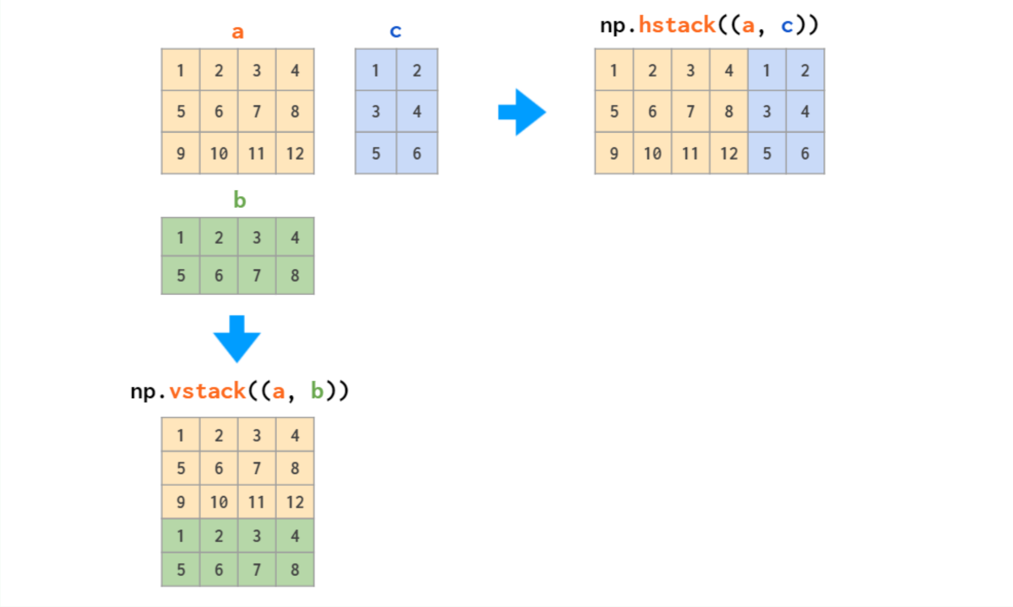
The inverse of stacking is splitting:



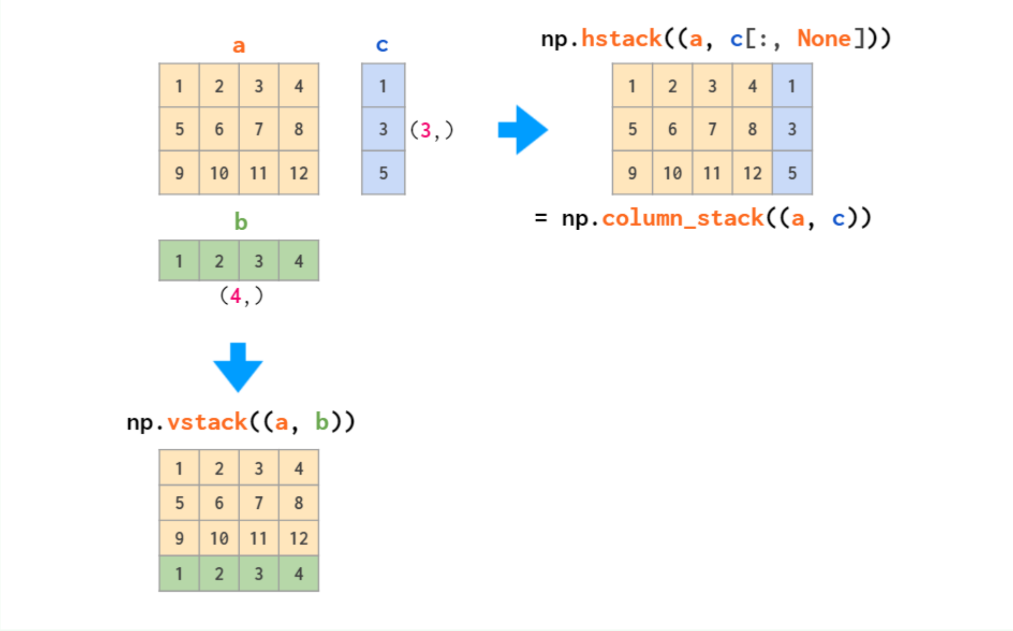
Matrix replication can be done in two ways: `tile` acts like copy-pasting and `repeat` like collated printing:

矩阵操作

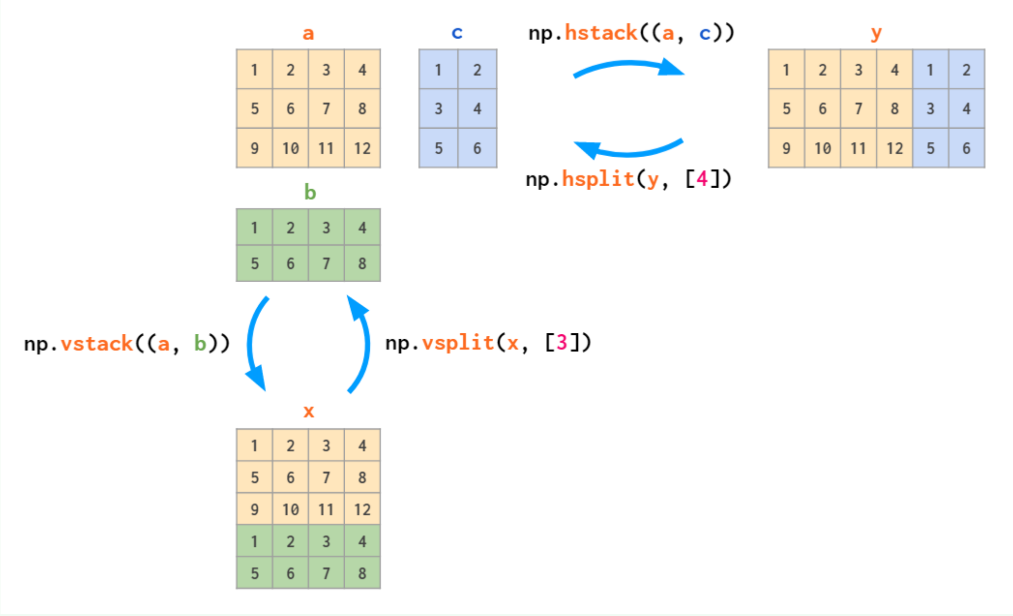
矩阵的拼接有以下两种方式：



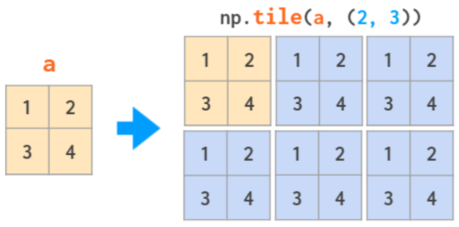
图示操作仅适用于矩阵堆叠或向量堆叠，而一维数组和矩阵的混合堆叠只有通过vstack才可实现，hstack会导致维度不匹配错误。因为前文提到将一维数组作为行向量，而不是列向量。为此，可以将其转换为行向量，或使用专门的column_stack函数执行此操作：

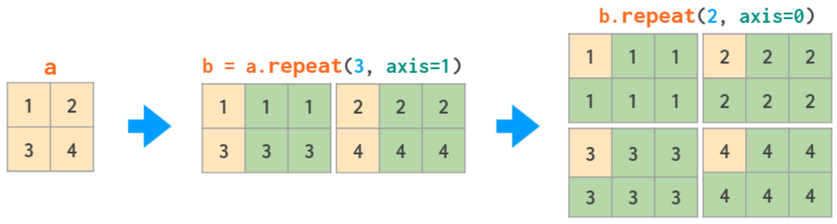


与stack对应的是split：

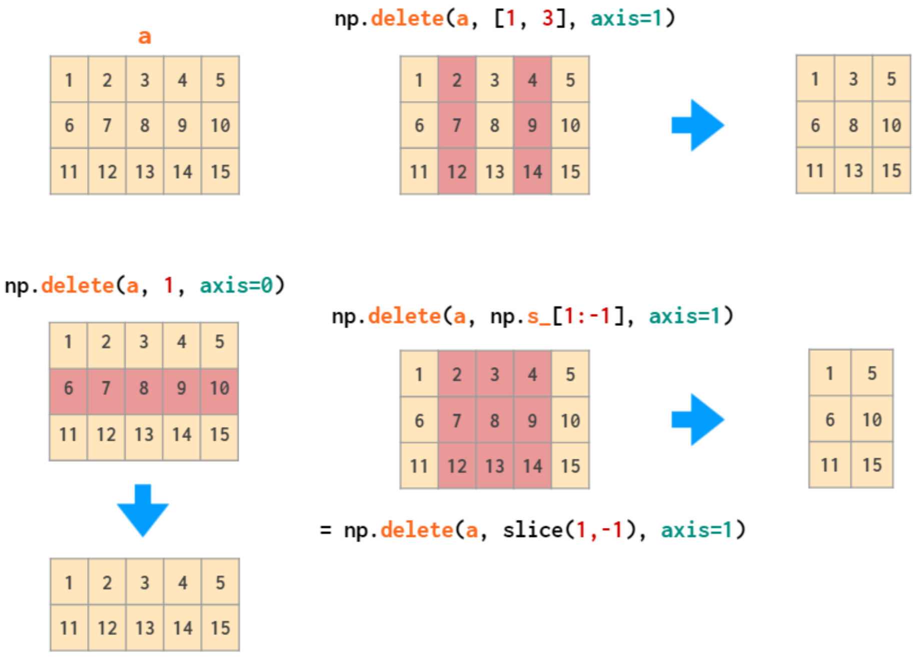


矩阵复制有两种方式：tile类似粘贴复制；repeat相当于分页打印。

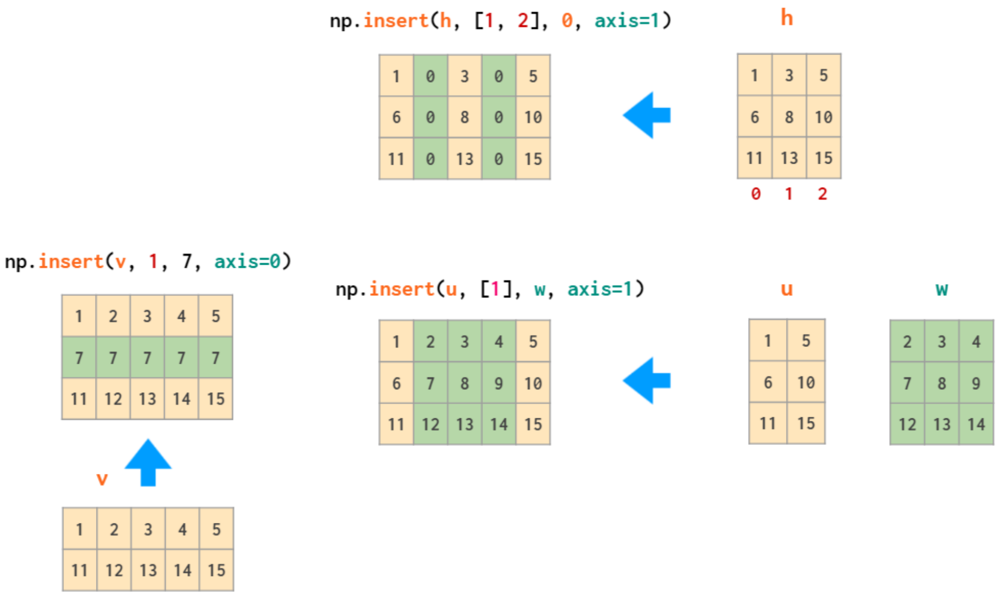




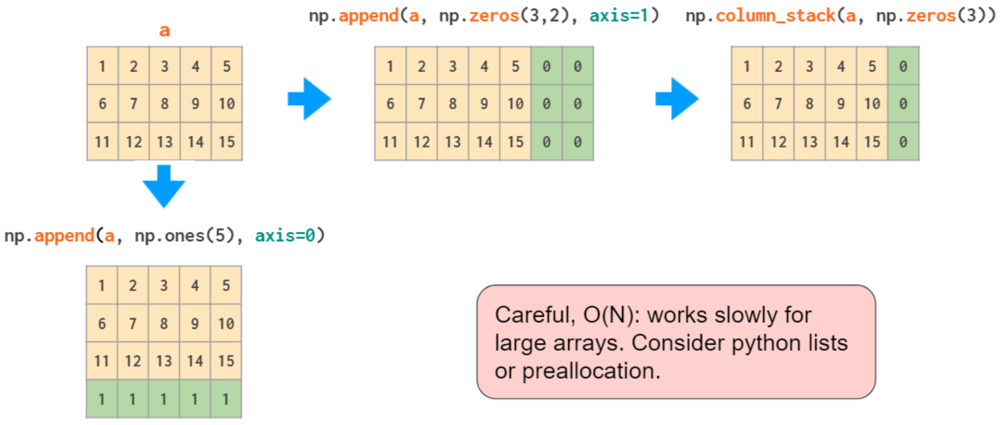
Specific columns and rows can be `delete`d like that:


a
np.delete(a, [1, 3], axis=1)
np.delete(a, 1, axis=0)
np.delete(a, np.s_[1:-1], axis=1)
= np.delete(a, slice(1,-1), axis=1)

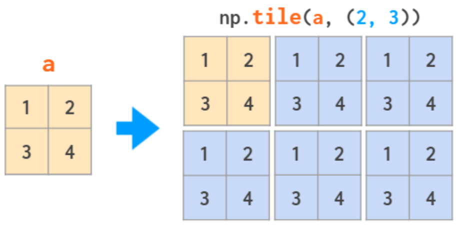
The inverse operation is `insert`:

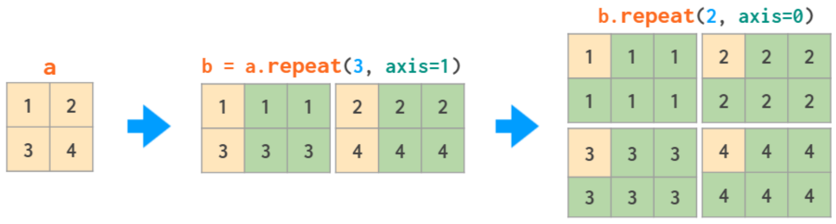

np.insert(h, [1, 2], 0, axis=1)
h
0 1 2
np.insert(v, 1, 7, axis=0)
v
np.insert(u, [1], w, axis=1)
u
w

The `append` function, just like `hstack`, is unable to automatically transpose 1D arrays, so once again, either the vector needs to be reshaped or a dimension added, or `column_stack` needs to be used instead:

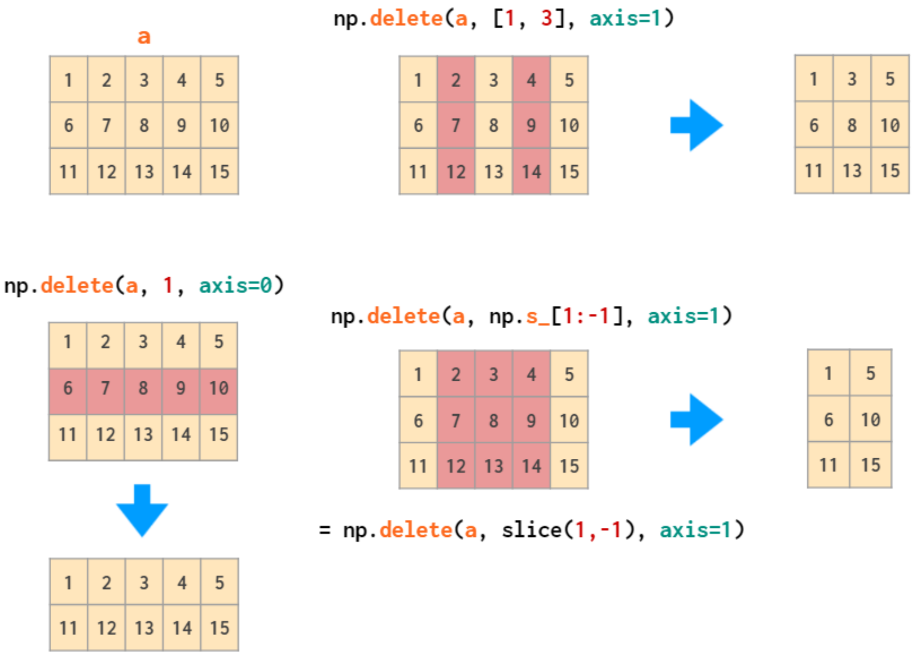

a
np.append(a, np.zeros(3,2), axis=1)
np.column_stack(a, np.zeros(3))
np.append(a, np.ones(5), axis=0)
Careful, O(N): works slowly for large arrays. Consider python lists or preallocation.

Actually, if all you need to do is add constant values to the border(s) of the array, the (slightly overcomplicated) `pad` function should suffice:

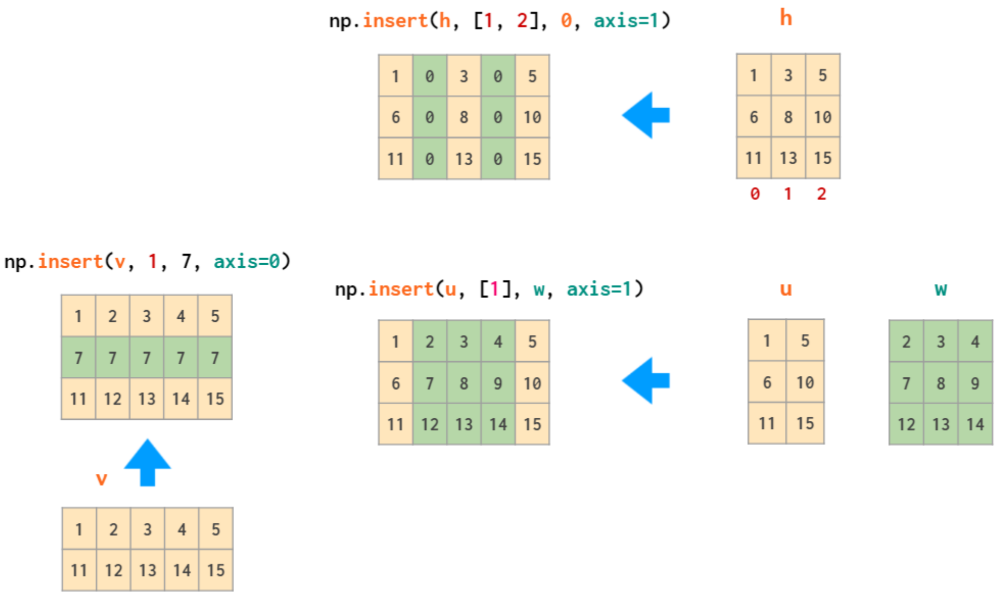




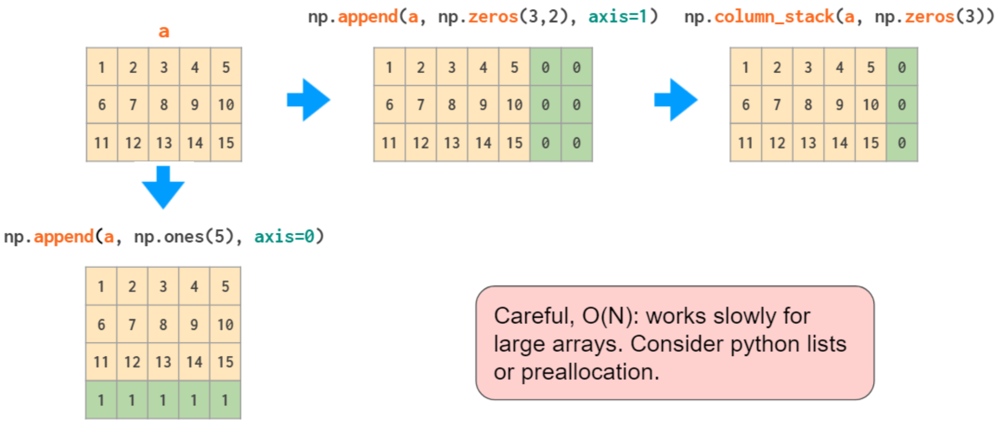
delete可以删除特定的行或列：


a
np.delete(a, [1, 3], axis=1)
np.delete(a, 1, axis=0)
np.delete(a, np.s_[1:-1], axis=1)
= np.delete(a, slice(1,-1), axis=1)

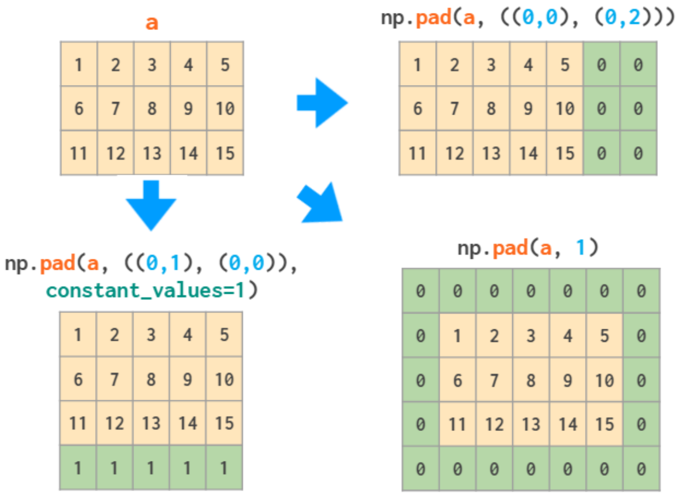
相应插入操作为insert：


np.insert(h, [1, 2], 0, axis=1)
h
0 1 2
np.insert(v, 1, 7, axis=0)
v
np.insert(u, [1], w, axis=1)
u
w

与hstack一样，append函数无法自动转置1D数组，因此需要重新调整向量形状或添加维数，或者使用column_stack：


a
np.append(a, np.zeros(3,2), axis=1)
np.column_stack(a, np.zeros(3))
np.append(a, np.ones(5), axis=0)
Careful, O(N): works slowly for large arrays. Consider python lists or preallocation.

如果仅仅是向数组的边界添加常量值，pad函数是足够的：



12

Meshgrids

The broadcasting rules make it simpler to work with meshgrids. Suppose, you need the following matrix (but of a very large size):

$$A_{ij} = j - i$$

0	1	2
-1	0	1

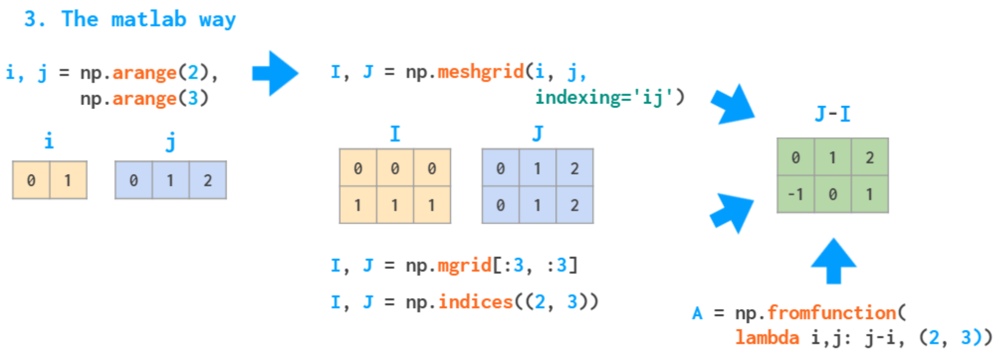
1. The c way

```
a = np.empty((2, 3))
for i in range(2):
    for j in range(3):
        a[i, j] = j - i
```

2. The python way

```
c = [[(j-i) for j in range(3)] for i in range(2)]
a = np.array(c)
```

Two obvious approaches are slow, as they use Python loops.
The MATLAB way of dealing with such problems is to create a meshgrid:


3. The matlab way

```
i, j = np.arange(2),
       np.arange(3)
```

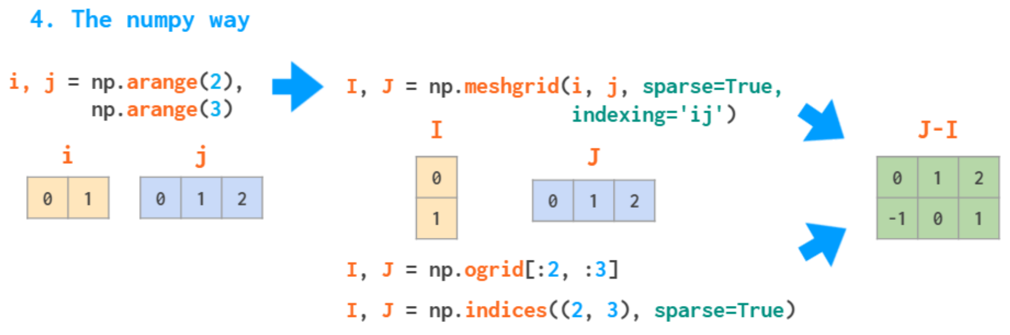
```
I, J = np.meshgrid(i, j,
                   indexing='ij')
```

```
I, J = np.mgrid[:3, :3]
I, J = np.indices((2, 3))
```

```
A = np.fromfunction(
    lambda i,j: j-i, (2, 3))
```

The `meshgrid` function accepts an arbitrary set of indices, `mgrid` — just slices and `indices` can only generate the complete index ranges. `fromfunction` calls the provided function just once, with the I and J argument as described above.

But actually, there is a better way to do it in NumPy. There’ s no need to spend memory on the whole I and J matrices (even though `meshgrid` is smart enough to only store references to the original vectors if possible). It is sufficient to store only vectors of the correct shape, and the broadcasting rules take care of the rest:


4. The numpy way

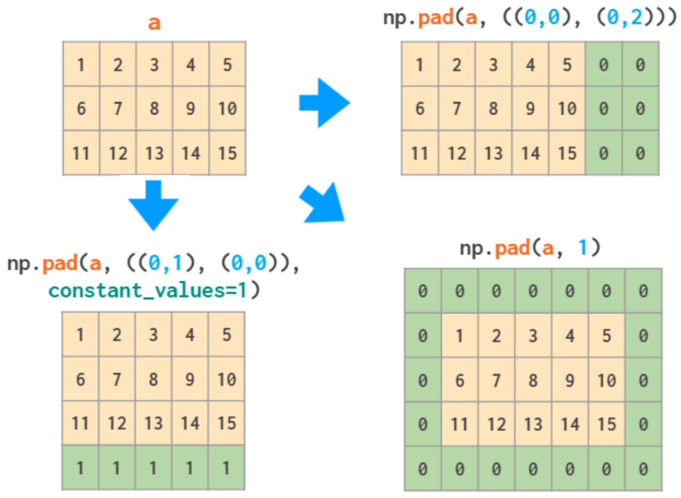
```
i, j = np.arange(2),
       np.arange(3)
```

```
I, J = np.meshgrid(i, j, sparse=True,
                   indexing='ij')
```

```
I, J = np.ogrid[:2, :3]
I, J = np.indices((2, 3), sparse=True)
```

Without the `indexing=’ ij’` argument, `meshgrid` will change the order of the arguments: `J, I= np.meshgrid(j, i)` — it is an ‘xy’ mode, useful for visualizing 3D plots (see the example from the docs).

Aside from initializing functions over a two- or three-dimensional grid, the meshes can be useful for indexing arrays:



Meshgrids

广播机制使得meshgrids变得容易。例如需要下图所示（但尺寸大得多）的矩阵：

$$A_{ij} = j - i$$

0	1	2
-1	0	1

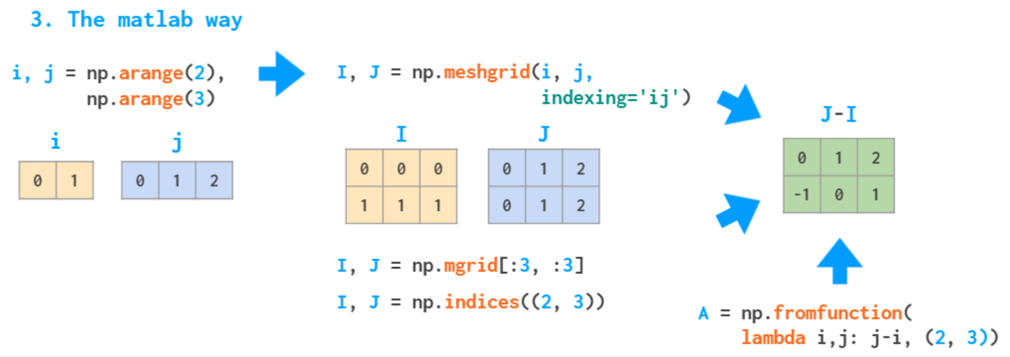
1. The c way

```
a = np.empty((2, 3))
for i in range(2):
    for j in range(3):
        a[i, j] = j - i
```

2. The python way

```
c = [[(j-i) for j in range(3)] for i in range(2)]
a = np.array(c)
```

上述两种方法由于使用了循环，因此都比较慢。MATLAB通过构建meshgrid处理这种问题。


3. The matlab way

```
i, j = np.arange(2),
       np.arange(3)
```

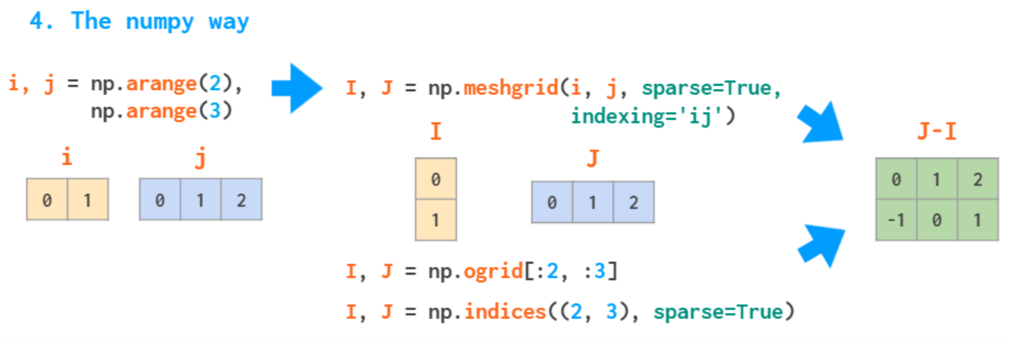
```
I, J = np.meshgrid(i, j,
                   indexing='ij')
```

```
I, J = np.mgrid[:3, :3]
I, J = np.indices((2, 3))
```

```
A = np.fromfunction(
    lambda i,j: j-i, (2, 3))
```

meshgrid函数接受任意一组索引，通过mgrid切片和indices索引生成完整的索引范围，然后，fromfunction函数根据I和J实现运算。

在NumPy中有一种更好的方法，无需在内存中存储整个I和J矩阵（虽然meshgrid已足够优秀，仅存储对原始向量的引用），仅存储形状矢量，然后通过广播规实现其余内容的处理：


4. The numpy way

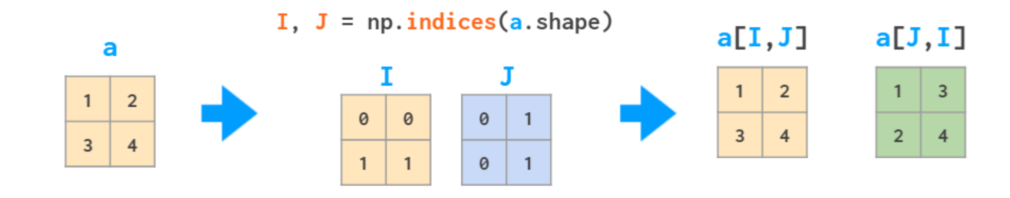
```
i, j = np.arange(2),
       np.arange(3)
```

```
I, J = np.meshgrid(i, j, sparse=True,
                   indexing='ij')
```

```
I, J = np.ogrid[:2, :3]
I, J = np.indices((2, 3), sparse=True)
```

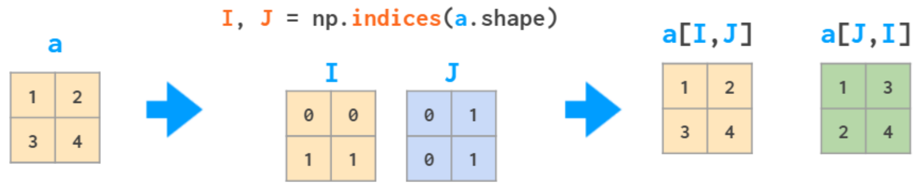
如果没有indexing ='ij'参数，那么meshgrid将更改参数的顺序，即J,I=np.meshgrid(j,i)——一种用于可视化3D绘图的“ xy”模式（详见该文档）。

除了在二维或三维网格上初始化函数外，网格还可以用于索引数组：



```
I, J = np.indices(a.shape)
```

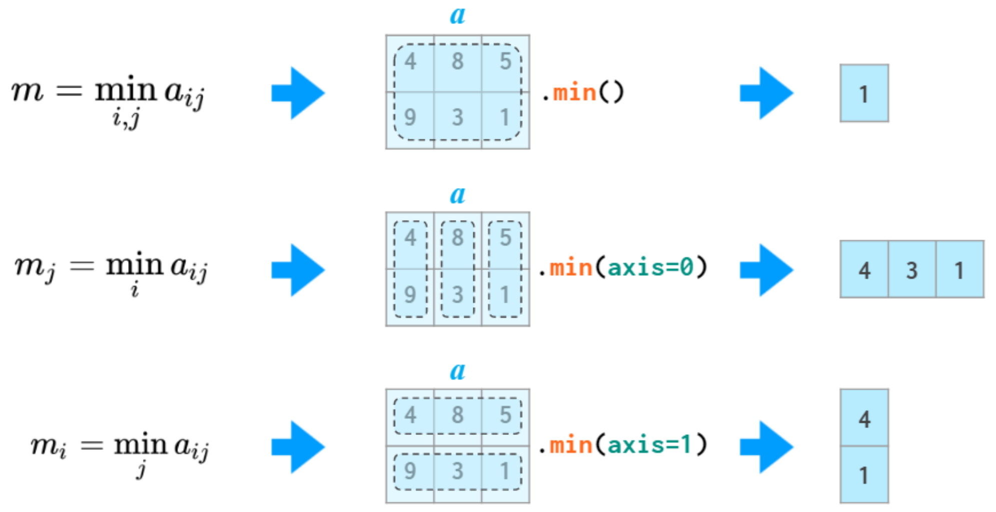
以上方法在稀疏网格中同样适用。



Works with sparse meshgrids, too

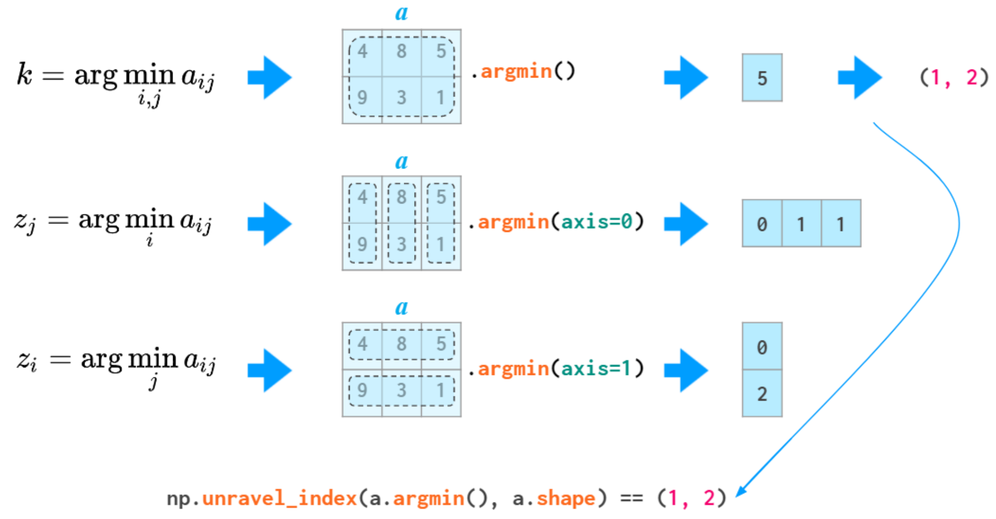
Matrix statistics

Just like `sum`, all the other stats functions (`min/max` , `argmin/argmax` , `mean/median/percentile` , `std/var`) accept the `axis` parameter and act accordingly:



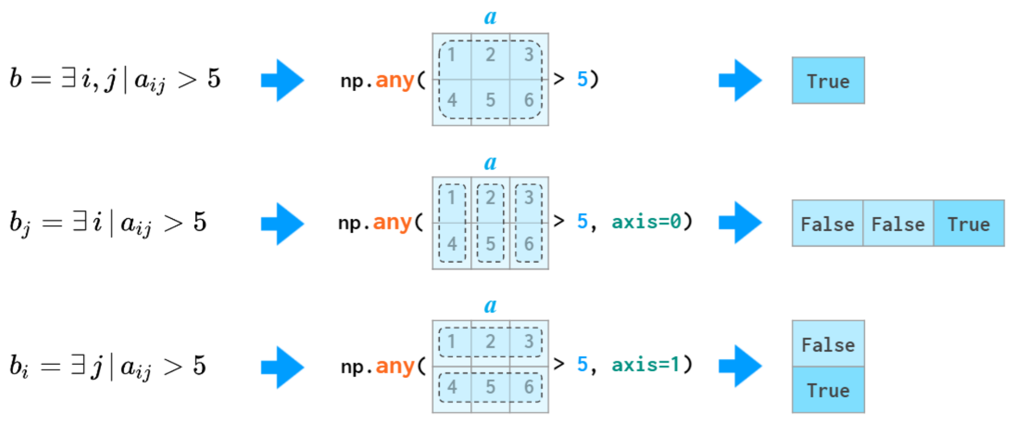
np.amin is just an alias of np.min to avoid shadowing the Python min when you write ' `from numpy import *`'

The `argmin` and `argmax` functions in 2D and above have an annoyance of returning the flattened index (of the first instance of the min and max value). To convert it to two coordinates, an `unravel_index` function is required:



$k = \arg\min_{i,j} a_{ij}$
a
4 8 5
9 3 1
.argmin()
5
(1, 2)
$z_j = \arg\min_i a_{ij}$
.argmin(axis=0)
0 1 1
$z_i = \arg\min_j a_{ij}$
.argmin(axis=1)
0
2
np.unravel_index(a.argmin(), a.shape) == (1, 2)

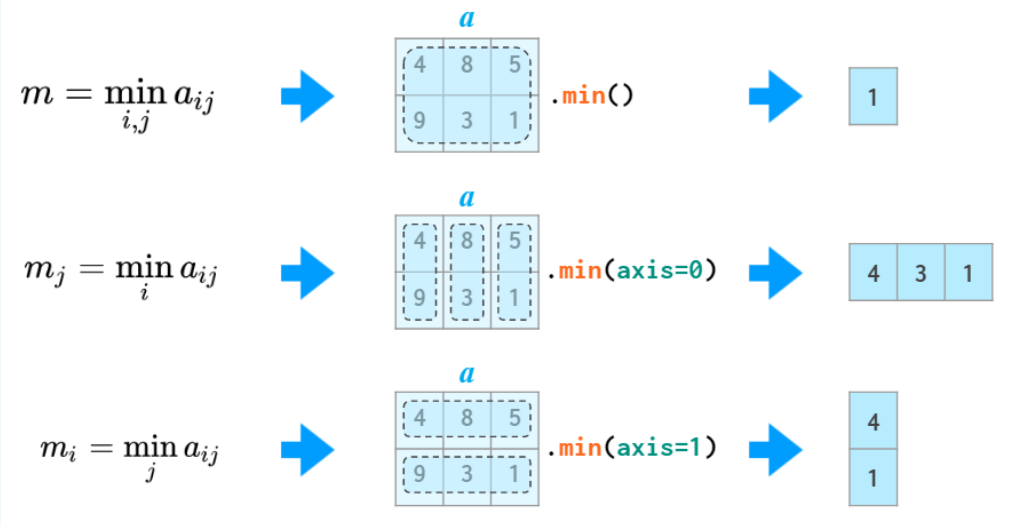
The quantifiers `all` and `any` are also aware of the `axis` argument:



$b = \exists i,j \,|\, a_{ij} > 5$
np.any(
a
1 2 3
4 5 6
> 5)
True
$b_j = \exists i \,|\, a_{ij} > 5$
> 5, axis=0)
False False True
$b_i = \exists j \,|\, a_{ij} > 5$
> 5, axis=1)
False
True

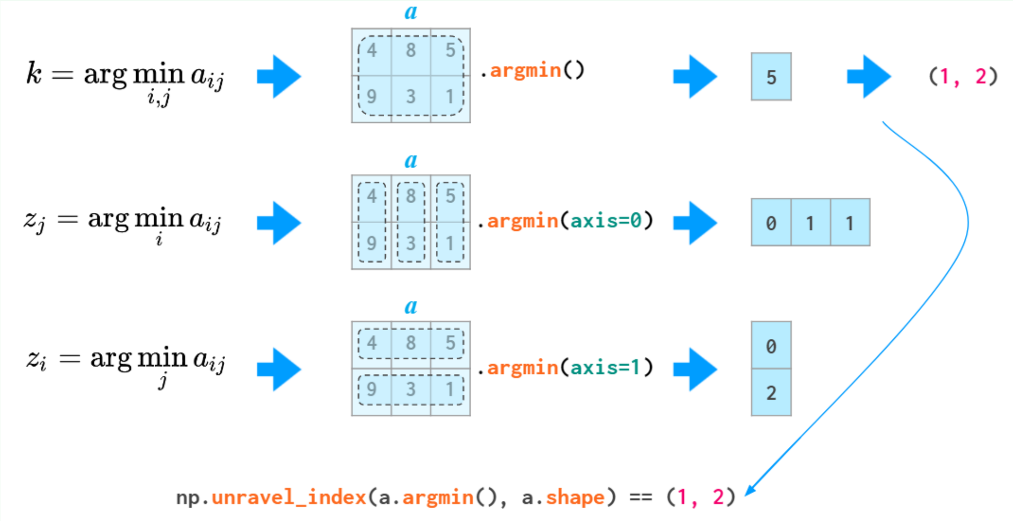
矩阵统计

就像sum函数，numpy提供了矩阵不同轴上的 `min/max` , `argmin/argmax` , `mean/median/percentile` , `std/var` 等函数。



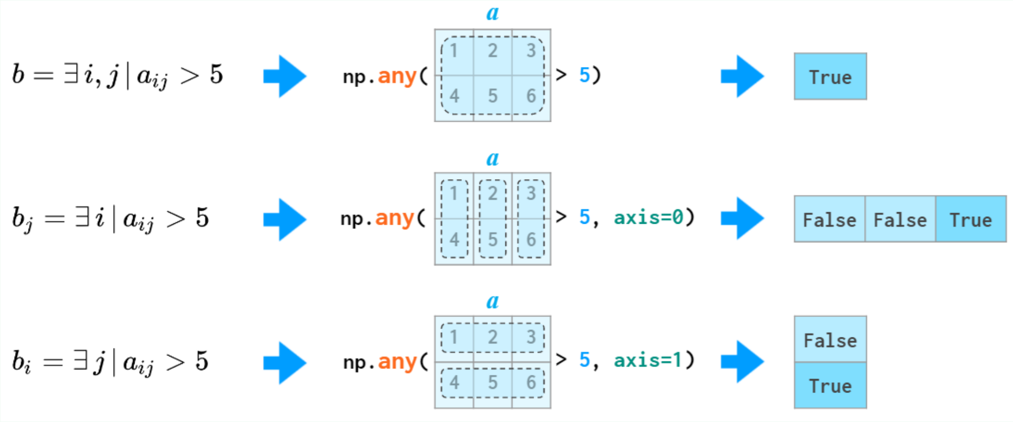
np.amin等同于np.min，这样做同样是为了避免from numpy import *可能的歧义。

2维及更高维中的argmin和argmax函数分别返回最小和最大值的索引，通过unravel_index函数可以将其转换为二维坐标：



$k = \arg\min_{i,j} a_{ij}$
a
4 8 5
9 3 1
.argmin()
5
(1, 2)
$z_j = \arg\min_i a_{ij}$
.argmin(axis=0)
0 1 1
$z_i = \arg\min_j a_{ij}$
.argmin(axis=1)
0
2
np.unravel_index(a.argmin(), a.shape) == (1, 2)

all和any同样也可作用于特定维度：



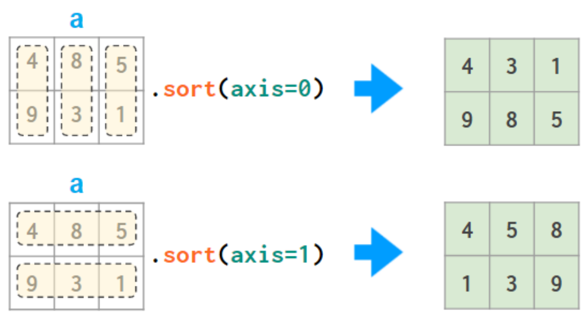
$b = \exists i,j \,|\, a_{ij} > 5$
np.any(
a
1 2 3
4 5 6
> 5)
True
$b_j = \exists i \,|\, a_{ij} > 5$
> 5, axis=0)
False False True
$b_i = \exists j \,|\, a_{ij} > 5$
> 5, axis=1)
False
True

Matrix sorting

矩阵排序

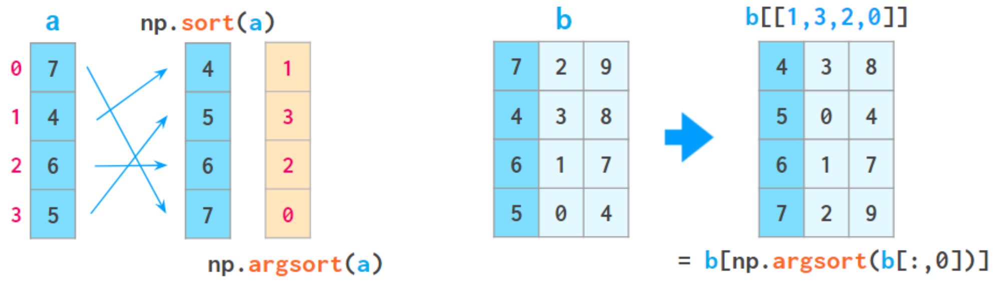
As helpful as the `axis` argument is for the functions listed above, it is as unhelpful for the 2D sorting:

python lists	numpy arrays	
a.sort()	a.sort()	sorts in-place
sorted(a)	np.sort(a)	returns new sorted array
a.sort(key=f)	–	key function
a.sort(reversed=False)	–	ascending/descending
–	a.sort(axis=-1)	which axis to sort along



It is just not what you would usually want from sorting a matrix or a spreadsheet: `axis` is in no way a replacement for the `key` argument. But luckily, NumPy has several helper functions which allow sorting by a column — or by several columns, if required:

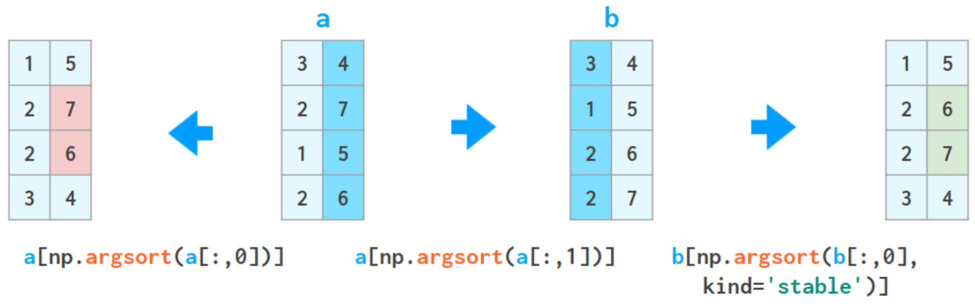
1. `a[a[:,0].argsort()]` sorts the array by the first column:



Here `argsort` returns an array of indices of the original array after sorting.

This trick can be repeated, but care must be taken so that the next sort does not mess up the results of the previous one:

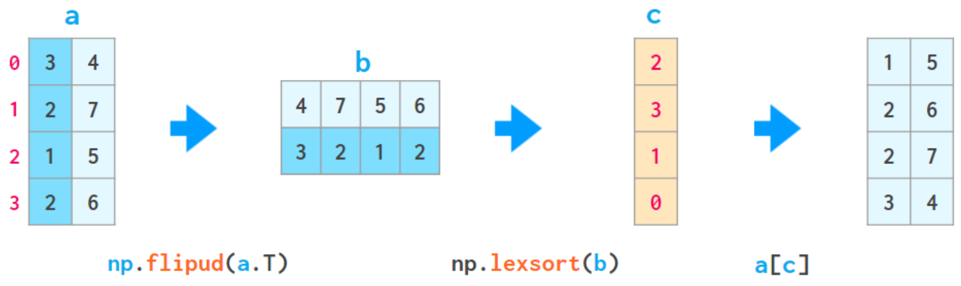
```
a = a[a[:,2].argsort()]
a = a[a[:,1].argsort(kind='stable')]
a = a[a[:,0].argsort(kind='stable')]
```



2. There's a helper function `lexsort` which sorts in the way described above by all available columns, but it always performs row-wise, and the order of rows to be sorted is inverted (i.e., from bottom to top) so its usage is a bit contrived, e.g.

– `a[np.lexsort(np.flipud(a[2,5].T))]` sorts by column 2 first and then (where the values in column 2 are equal) by column 5.

– `a[np.lexsort(np.flipud(a.T))]` sorts by all columns in left-to-right order.

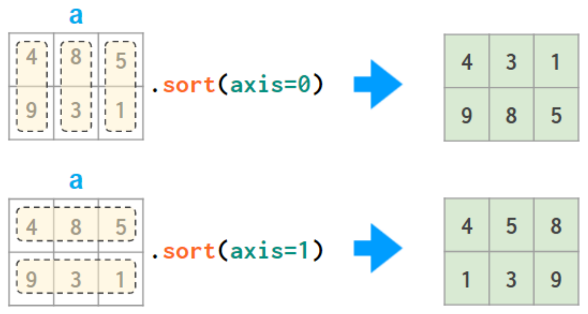


Here `flipud` flips the matrix in the up-down direction (to be precise, in the `axis=0` direction, same as `a[::-1,...]`, where three dots mean "all other dimensions" — so it's all of a sudden `flipud`, not `fliplr`, that flips the 1D arrays).

3. There also is an `order` argument to `sort`, but it is neither fast nor easy to use if you start with an ordinary (unstructured) array.

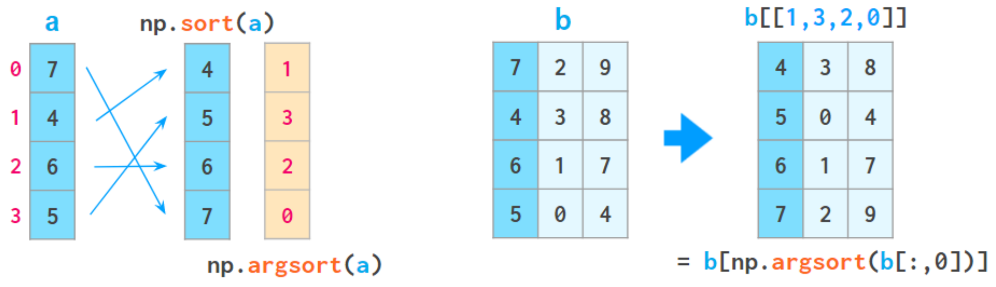
虽然在前文中，axis参数适用于不同函数，但在二维数组排序中影响较小：

python lists	numpy arrays	
a.sort()	a.sort()	sorts in-place
sorted(a)	np.sort(a)	returns new sorted array
a.sort(key=f)	–	key function
a.sort(reversed=False)	–	ascending/descending
–	a.sort(axis=-1)	which axis to sort along



你通常不需要上述这样的排序矩阵，axis不是key参数的替代。但好在NumPy提供了其他功能，这些功能允许按一列或几列进行排序：

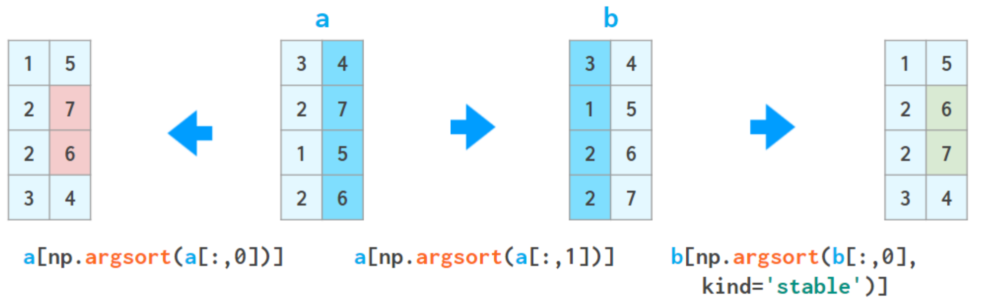
1、a[a [:,0] .argsort()]表示按第一列对数组进行排序：



其中，argsort返回排序后的原始数组的索引数组。

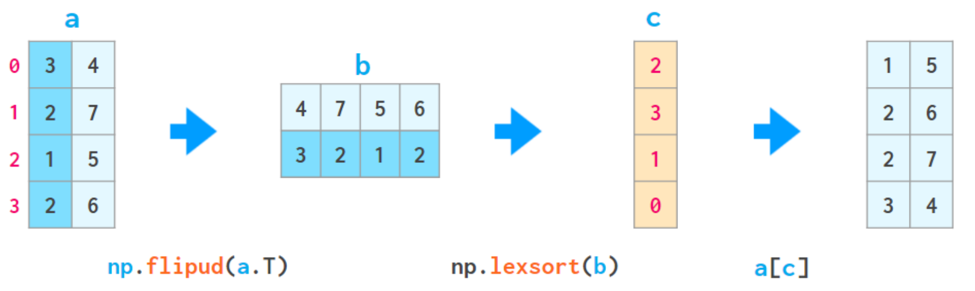
可以重复使用该方法，但千万不要搞混：

```
a = a[a[:,2].argsort()]

a = a[a[:,1].argsort(kind='stable')]

a = a[a[:,0].argsort(kind='stable')]
```



2、函数lexsort可以像上述这样对所有列进行排序，但是它总是按行执行，并且排序的行是颠倒的（即从下到上），其用法如下：

a[np.lexsort(np.flipud(a[2,5].T))]，首先按第2列排序，然后按第5列排序；a[np.lexsort(np.flipud(a.T))]，从左到右依次排序各列。



其中，flipud沿上下方向翻转矩阵（沿axis = 0方向，与a [::-1，...]等效，其中...表示“其他所有维度”），注意区分它与fliplr，fliplr用于1维数组。

3、sort函数还有一个order参数，但该方法极不友好，不推荐学习。

4、在pandas中排序也是不错的选择，因为在pandas中操作位置确定，可读性好且不易出错：

- pd.DataFrame(a).sort_values(by=[2,5]).to_numpy()，先按第2列排序，再按第5列排序。

-pd.DataFrame(a).sort_values().to_numpy()，按从左到右的顺序对所有列进行排序。

4. It might be a better option to do it in pandas since this particular operation is way more readable and less error-prone there:

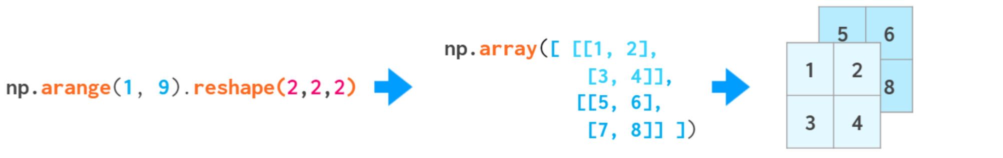
– `pd.DataFrame(a).sort_values(by=[2,5]).to_numpy()` sorts by column 2, then by column 5.

– `pd.DataFrame(a).sort_values().to_numpy()` sorts by all columns in the left-to-right order.

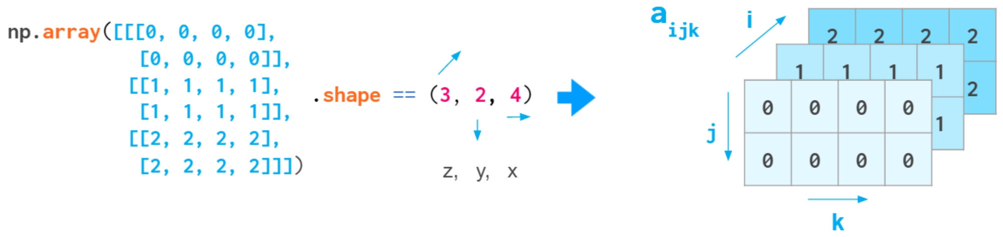
15

3. 3D and Above

When you create a 3D array by reshaping a 1D vector or converting a nested Python list, the meaning of the indices is (z,y,x). The first index is the number of the plane, then the coordinates go in that plane:



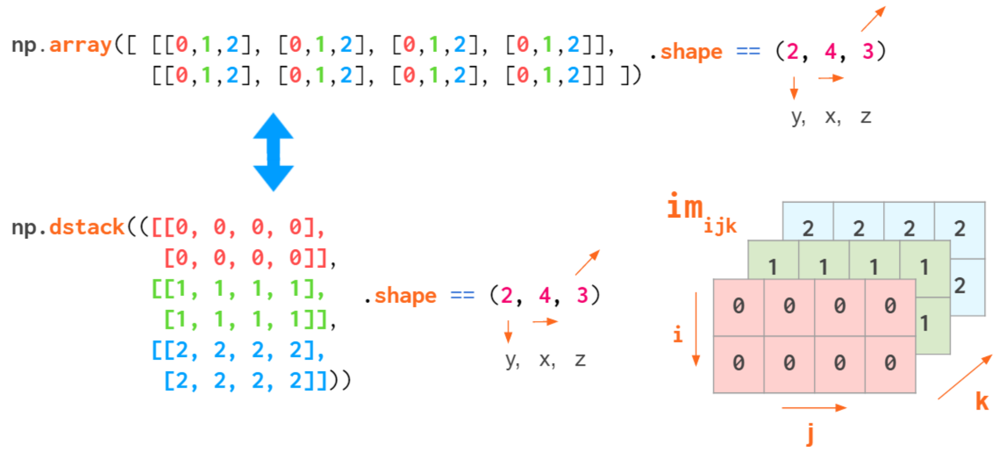
```
np.arange(1, 9).reshape(2,2,2)

np.array([ [[1, 2],
            [3, 4]],
           [[5, 6],
            [7, 8]] ])
```



```
np.array([[[0, 0, 0, 0],
           [0, 0, 0, 0]],
          [[1, 1, 1, 1],
           [1, 1, 1, 1]],
          [[2, 2, 2, 2],
           [2, 2, 2, 2]]])

.shape == (3, 2, 4)
```

This index order is convenient, for example, for keeping a bunch of gray-scale images: `a[i]` is a shortcut for referencing the i-th image.

But this index order is not universal. When working with RGB images, the (y,x,z) order is usually used: First are two pixel coordinates, and the last one is the color coordinate (RGB in Matplotlib, BGR in OpenCV):

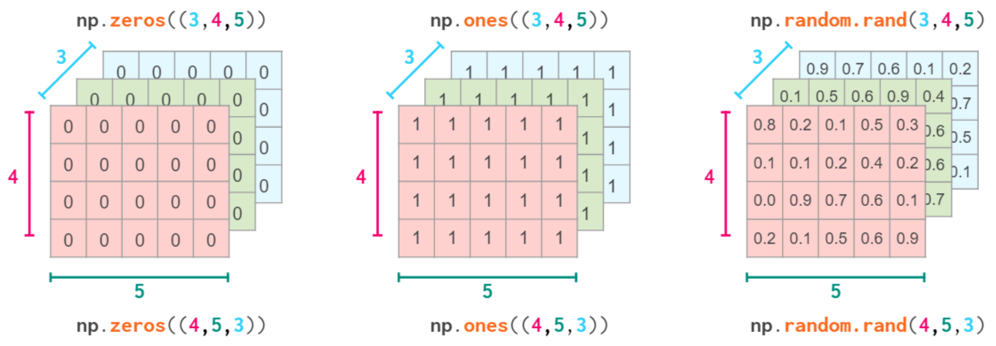


```
np.array([ [[0,1,2], [0,1,2], [0,1,2], [0,1,2]],
           [[0,1,2], [0,1,2], [0,1,2], [0,1,2]] ])

.shape == (2, 4, 3)

np.dstack(([[0, 0, 0, 0],
            [0, 0, 0, 0]],
           [[1, 1, 1, 1],
            [1, 1, 1, 1]],
           [[2, 2, 2, 2],
            [2, 2, 2, 2]]))

.shape == (2, 4, 3)
```

This way it is convenient to reference a specific pixel: `a[i,j]` gives an RGB tuple of the `(i,j)` pixel.

So the actual command to create a certain geometrical shape depends on the conventions of the domain you' re working on:

Generic 3D arrays creation



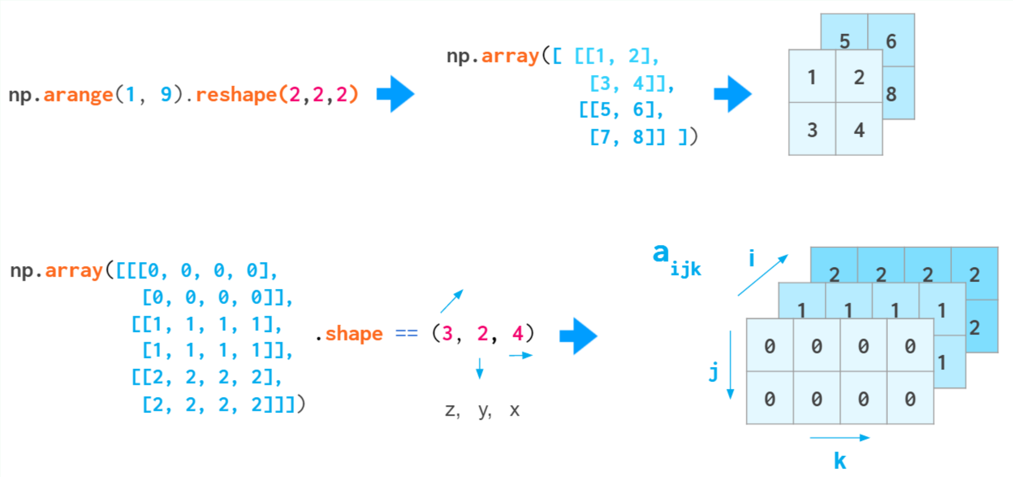
```
np.zeros((3,4,5))      np.ones((3,4,5))      np.random.rand(3,4,5)
np.zeros((4,5,3))      np.ones((4,5,3))      np.random.rand(4,5,3)
```

RGB images creation

Obviously, NumPy functions like `hstack`, `vstack`, or `dstack` are not aware of those conventions. The index order hardcoded in them is (y,x,z), the RGB images order:

3、3维及更高维数组

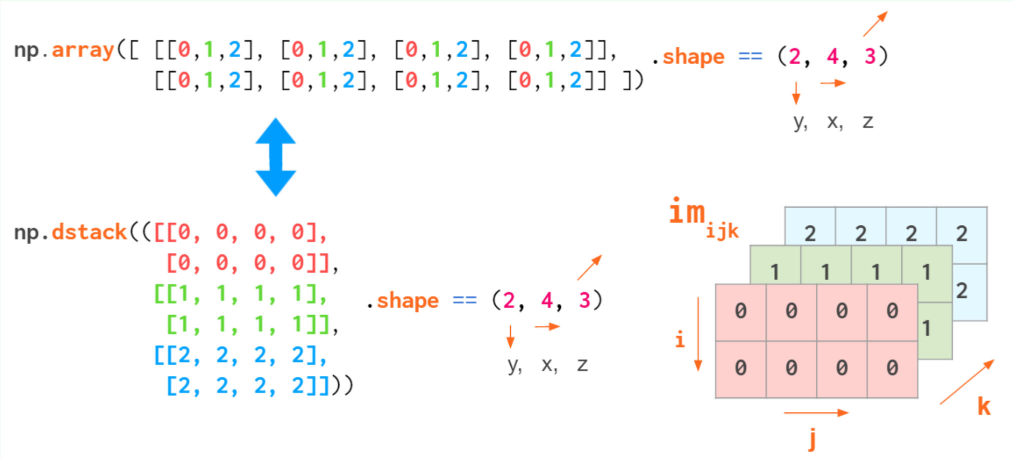
通过重塑1维向量或转换嵌套Python列表来创建3维数组时，索引分别对应(z,y,x)。索引z是平面编号，(y,x)坐标在该平面上移动：



```
np.arange(1, 9).reshape(2,2,2)

np.array([ [[1, 2],
            [3, 4]],
           [[5, 6],
            [7, 8]] ])

np.array([[[0, 0, 0, 0],
           [0, 0, 0, 0]],
          [[1, 1, 1, 1],
           [1, 1, 1, 1]],
          [[2, 2, 2, 2],
           [2, 2, 2, 2]]])

.shape == (3, 2, 4)
```

通过上述索引顺序，可以方便的保留灰度图像，a[i]表示第i个图像。

但这样的索引顺序并不具有广泛性，例如在处理RGB图像时，通常使用(y,x,z)顺序：首先是两个像素坐标，然后才是颜色坐标（Matplotlib中的RGB，OpenCV中的BGR）：

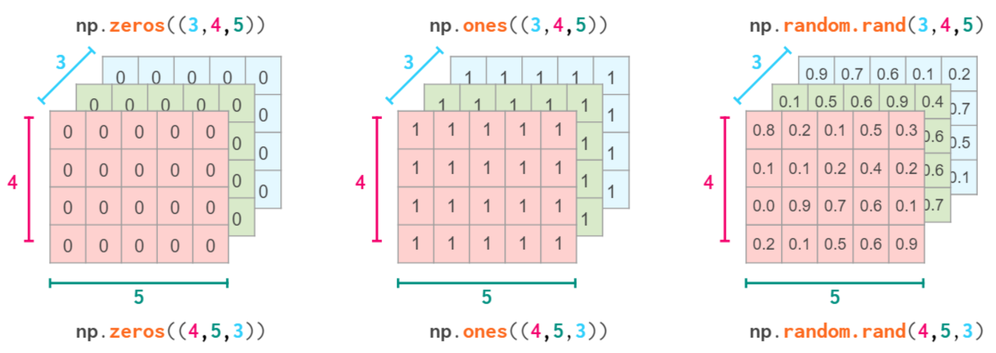


```
np.array([ [[0,1,2], [0,1,2], [0,1,2], [0,1,2]],
           [[0,1,2], [0,1,2], [0,1,2], [0,1,2]] ])

.shape == (2, 4, 3)

np.dstack(([[0, 0, 0, 0],
            [0, 0, 0, 0]],
           [[1, 1, 1, 1],
            [1, 1, 1, 1]],
           [[2, 2, 2, 2],
            [2, 2, 2, 2]]))

.shape == (2, 4, 3)
```

这样可以方便地定位特定像素，如a[i,j]给出像素(i,j)的RGB元组。

因此，几何形状的创建实际取决于你对域的约定：

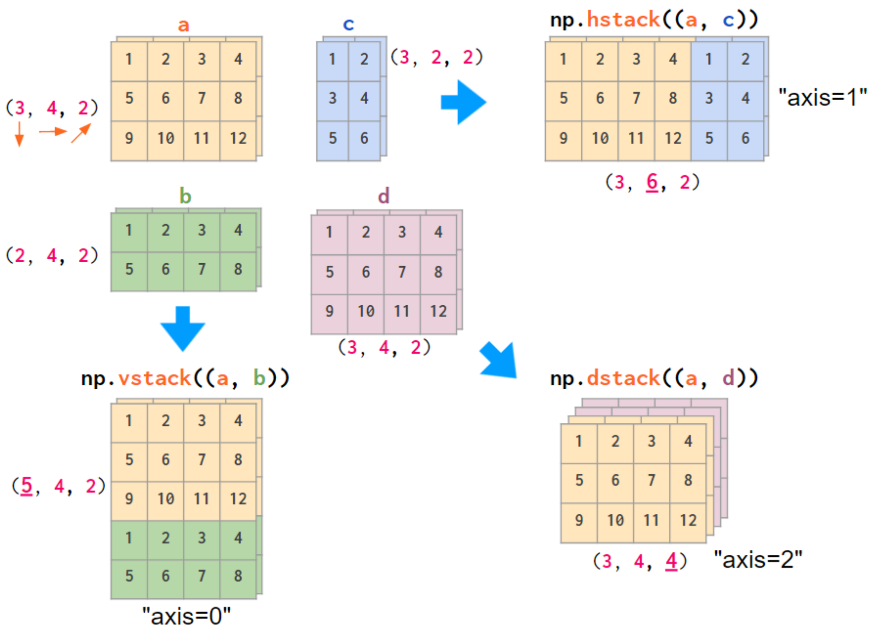
Generic 3D arrays creation



```
np.zeros((3,4,5))      np.ones((3,4,5))      np.random.rand(3,4,5)
np.zeros((4,5,3))      np.ones((4,5,3))      np.random.rand(4,5,3)
```

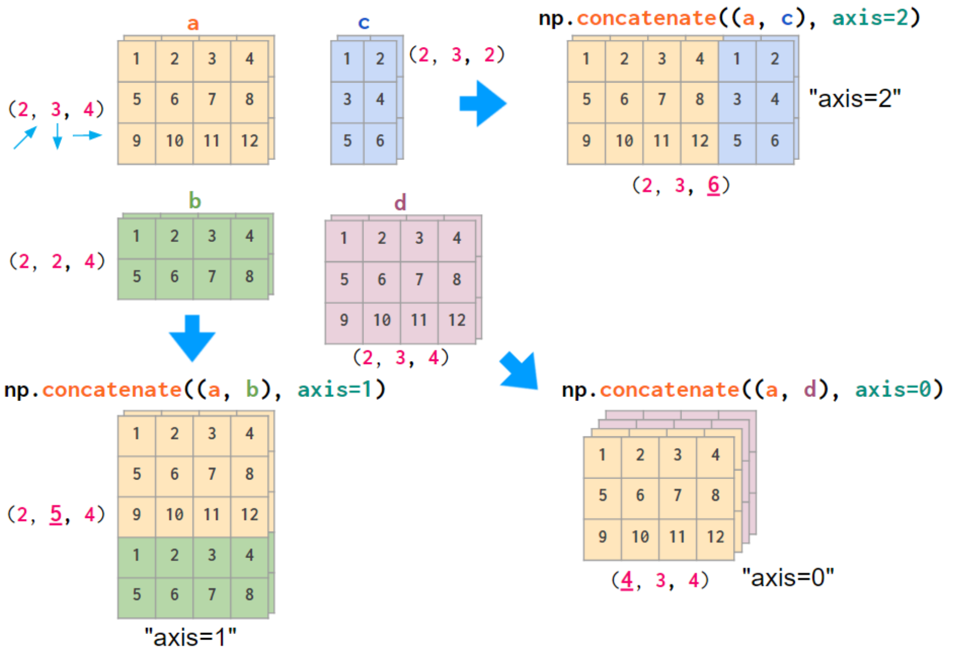
RGB images creation

显然，hstack，vstack或dstack之类的NumPy函数并不一定满足这些约定，其默认的索引顺序是(y,x,z)，RGB图像顺序如下：



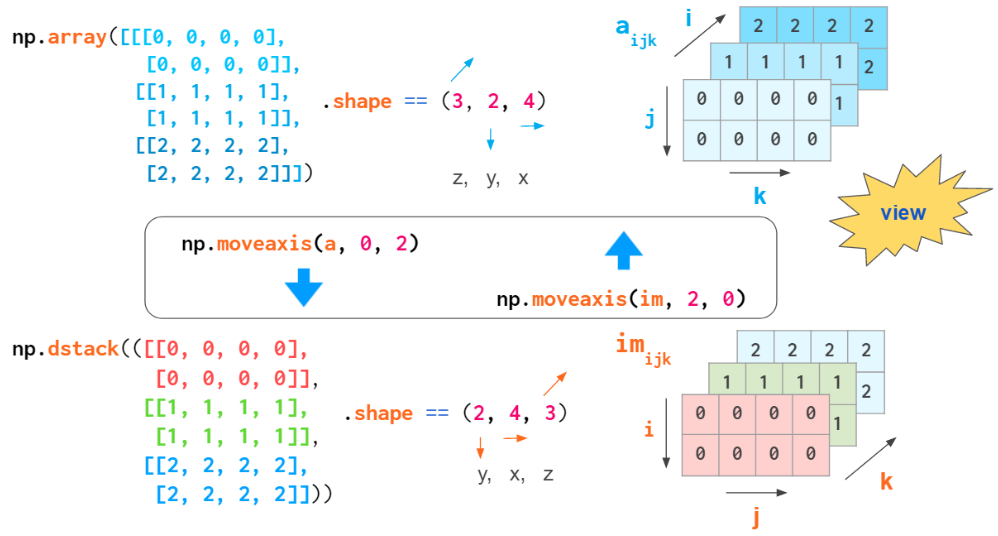
Stacking RGB images (only two colors here)

If your data is laid out differently, it is more convenient to stack images using the `concatenate` command, feeding it the explicit index number in an `axis` argument:



Stacking generic 3D arrays

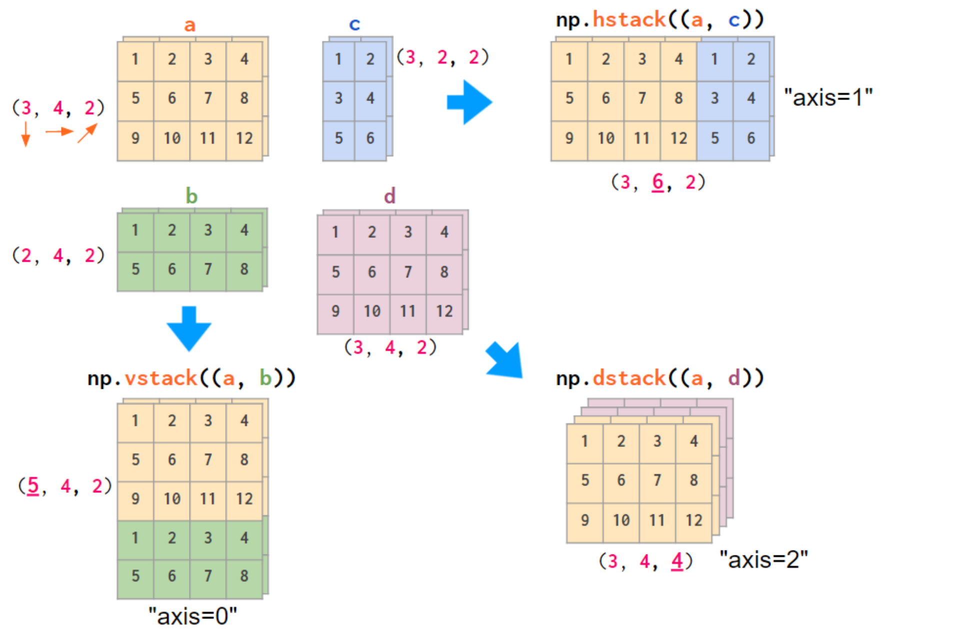
If thinking in terms of axes numbers is not convenient to you, you can convert the array to the form that is hardcoded into `hstack` and co:



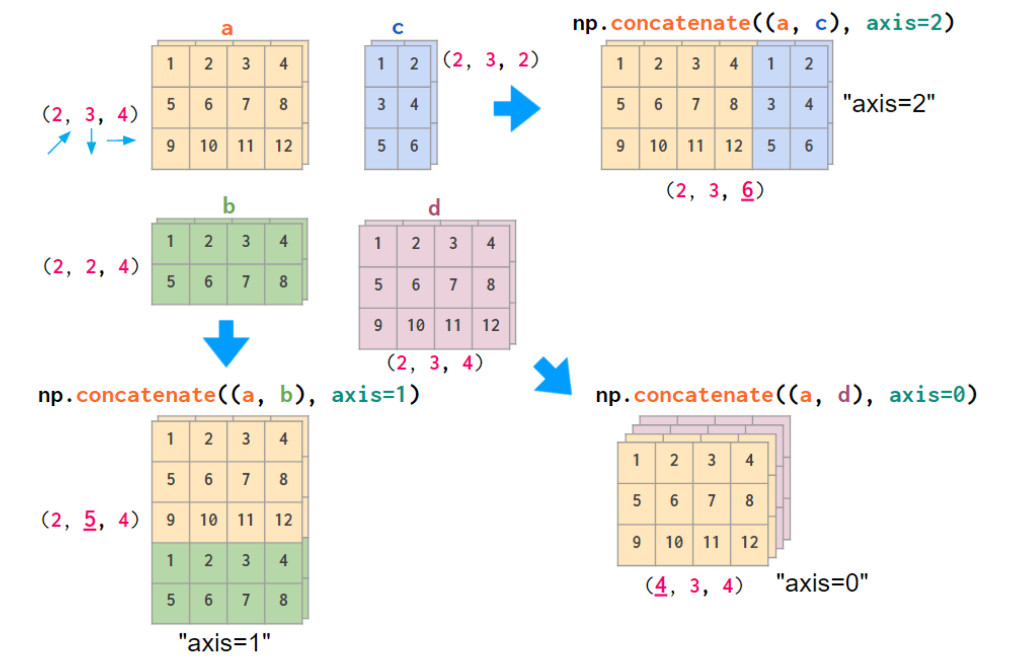
```
np.array([[[0, 0, 0, 0],
           [0, 0, 0, 0]],
          [[1, 1, 1, 1],
           [1, 1, 1, 1]],
          [[2, 2, 2, 2],
           [2, 2, 2, 2]]])
.shape == (3, 2, 4)

np.moveaxis(a, 0, 2)
np.moveaxis(im, 2, 0)

np.dstack(([[0, 0, 0, 0],
            [0, 0, 0, 0]],
           [[1, 1, 1, 1],
            [1, 1, 1, 1]],
           [[2, 2, 2, 2],
            [2, 2, 2, 2]]))
.shape == (2, 4, 3)
```

This conversion is cheap: No actual copying takes place; it just mixes the order of indices on the fly.

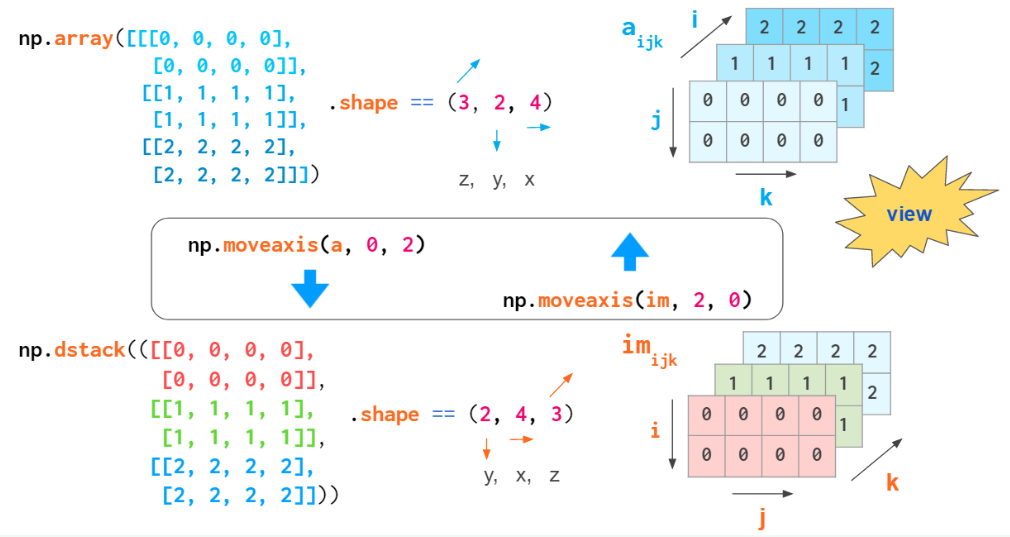
Another operation that mixes the order of indices is array transposition. Examining it might make you feel more familiar with 3D arrays. Depending on the order of axes you’ve decided on, the actual command to transpose all planes of your array will be different: For generic arrays, it is swapping indices 1 and 2, for RGB images it is 0 and 1:



如果数据不是这样的布局，使用concatenate命令可以方便的堆叠图像，并通过axis参数提供索引号：

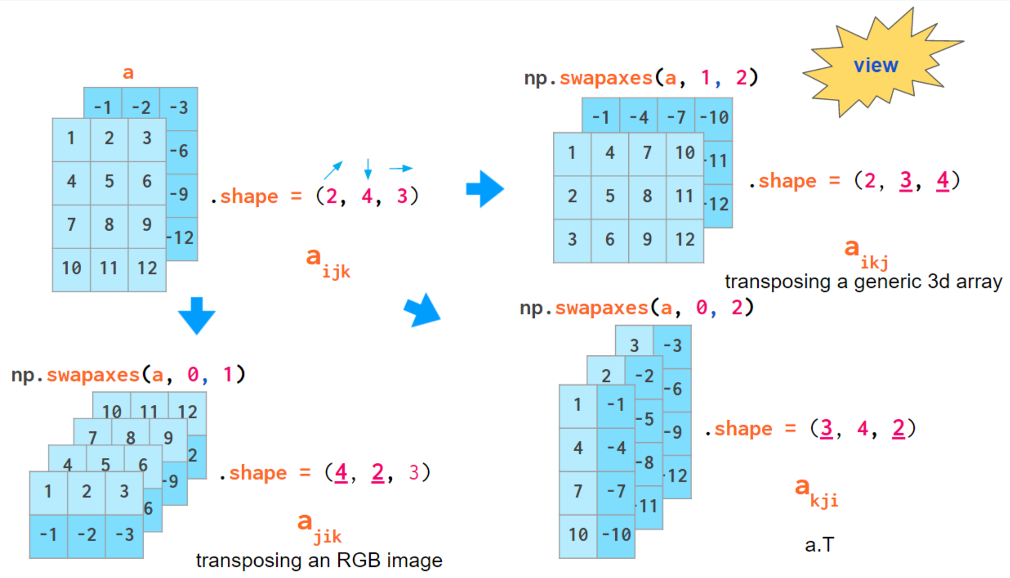

np.concatenate((a, c), axis=2)
np.concatenate((a, b), axis=1)
np.concatenate((a, d), axis=0)
"axis=2"
"axis=1"
"axis=0"

如果不考虑轴数，可以将数组转换hstack和相应形式：

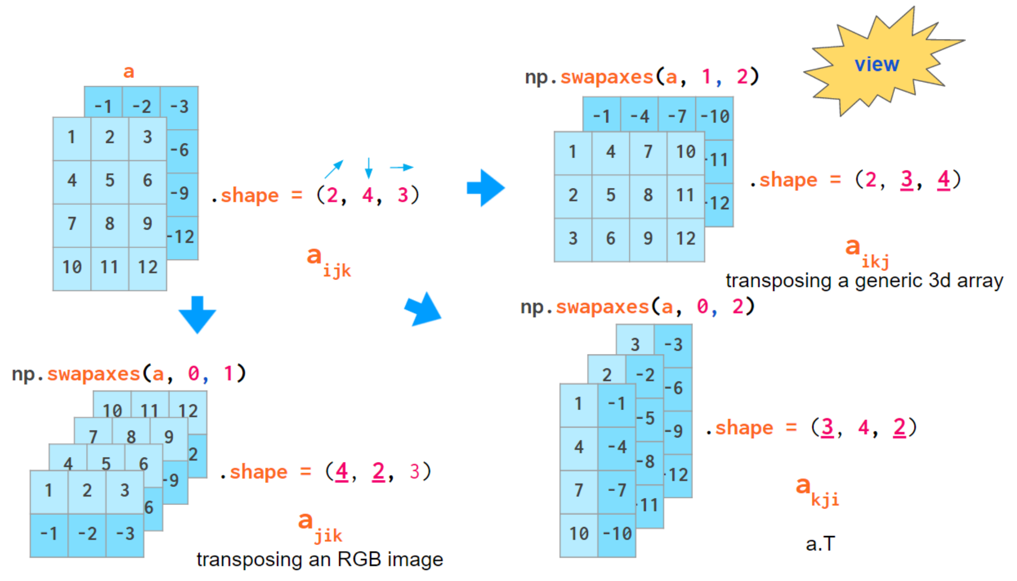


这种转换非常方便，该过程只是混合索引的顺序重排，并没有实际的复制操作。

通过混合索引顺序可实现数组转置，掌握该方法将加深你对3维数据的了解。根据确定的轴顺序，转置数组平面的命令有所不同：对于通用数组，交换索引1和2，对于RGB图像交换0和1：



```
np.swapaxes(a, 1, 2)
np.swapaxes(a, 0, 2)
np.swapaxes(a, 0, 1)
```



Interesting though that the default `axes` argument for `transpose` (as well as the only `a.T` operation mode) reverses the index order, which coincides with neither of the index order conventions described above.

Broadcasting works with higher dimensions as well, see my note "Broadcasting in NumPy" for details.

And finally, here's a function that can save you a lot of Python loops when dealing with the multidimensional arrays and can make your code cleaner — `einsum` (Einstein summation):

Matrix multiplication

$$c_{ik} = \sum_j a_{ij} b_{jk}$$

```
c = np.einsum('ij,jk->ik', a, b)
```

Tensor multiplication

$$c_{ijlm} = \sum_k a_{ijk} b_{klm}$$

```
c = np.einsum('ijk,klm->ijlm', a, b)
```

It sums the arrays along the repeated indices. In this particular example `np.tensordot(a, b, axis=1)` would suffice in both cases, but in the more complex cases `einsum` might work faster and is generally easier to both write and read — once you understand the logic behind it.

If you want to test your NumPy skills, there's a tricky crowd-sourced set of 100 NumPy exercises[10] on GitHub.

Let me know (on reddit, hackernews, etc) if I have missed your favorite NumPy feature, and I'll try to fit it in!

References

1. Scott Sievert, NumPy GPU acceleration
2. Jay Alammar, A Visual Intro to NumPy and Data Representation
3. Big-O Cheat Sheet site
4. Python Time Complexity wiki page
5. NumPy Issue #14989, Reverse param in ordering functions
6. NumPy Issue #2269, First nonzero element
7. Numba library homepage
8. The Floating-Point Guide, Comparison
9. NumPy Issue #10161, numpy.isclose vs math.isclose
10. 100 NumPy exercises on GitHub

注意，transpose（a.T）的默认轴参数会颠倒索引顺序，这不同于上述述两种索引顺序。

广播机制同样适用多维数组，更多详细信息可参阅笔记“ NumPy中的广播”。

最后介绍einsum(Einstein summation)函数，这将使你在处理多维数组时避免很多Python循环，代码更为简洁：

Matrix multiplication

$$c_{ik} = \sum_j a_{ij} b_{jk}$$

```
c = np.einsum('ij,jk->ik', a, b)
```

Tensor multiplication

$$c_{ijlm} = \sum_k a_{ijk} b_{klm}$$

```
c = np.einsum('ijk,klm->ijlm', a, b)
```

该函数对重复索引的数组求和。在一般情况下，使用np.tensordot(a,b,axis=1)就可以，但在更复杂的情况下，einsum速度更快，读写更容易。

如果你想看看自己的NumPy水平到底如何，可以在GitHub上进行练习——例如100个NumPy练习。

对于本文未介绍到的NumPy常用功能，欢迎各位读者通过reddi、hackernews给我留言，我将进一步完善本文！

参考

1. Scott Sievert, NumPy GPU acceleration
2. Jay Alammar, A Visual Intro to NumPy and Data Representation
3. Big-O Cheat Sheet site
4. Python Time Complexity wiki page
5. NumPy Issue #14989, Reverse param in ordering functions
6. NumPy Issue #2269, First nonzero element
7. Numba library homepage
8. The Floating-Point Guide, Comparison
9. NumPy Issue #10161, numpy.isclose vs math.isclose
10. 100 NumPy exercises on GitHub