

◆ Member-only story

NumPy Illustrated: The Visual Guide to NumPy

Brush up your NumPy or learn it from scratch



Lev Maximov · [Follow](#)

Published in Better Programming

19 min read · Dec 21, 2020

Listen

Share

More

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

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} * \begin{array}{|c|} \hline 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 3 & 6 \\ \hline \end{array}$$
$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} / \begin{array}{|c|} \hline 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0.33 & 0.67 \\ \hline \end{array} \text{ np.float64}$$
$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} // \begin{array}{|c|} \hline 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline \end{array} \text{ np.int32}$$

$$\begin{array}{|c|c|} \hline 3 & 4 \\ \hline \end{array} ** \begin{array}{|c|} \hline 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 9 & 16 \\ \hline \end{array}$$

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 Alammar², as a starting point, significantly expanded its coverage, and amended a pair of nuances.

Numpy Array vs. Python List

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

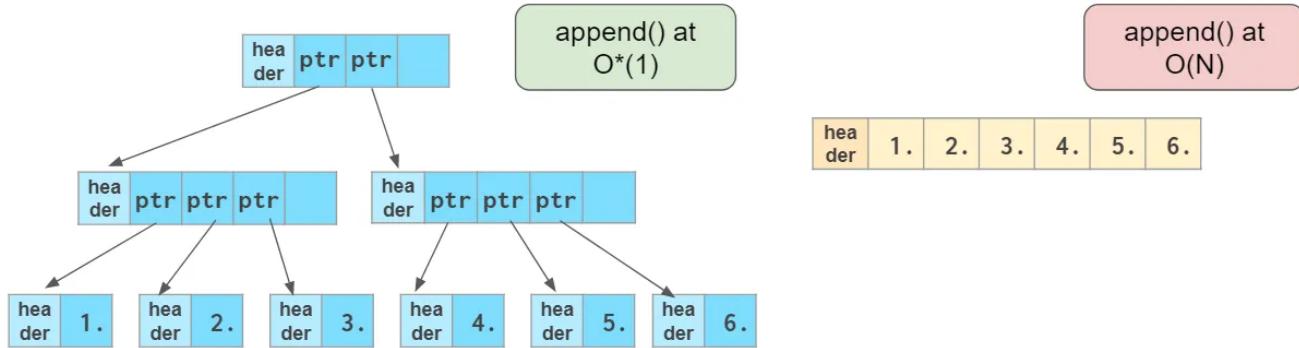
python list

vs

numpy array

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

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



Here $O(N)$ means that the time necessary to complete the operation is proportional to the size of the array (see [Big-O Cheat Sheet³](#) 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 Complexity⁴](#) wiki page)

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:



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.

More about dtypes in my new article

A Comprehensive Guide to NumPy Data Types

What else is out there besides int32 and float64?

[betterprogramming.pub](#)

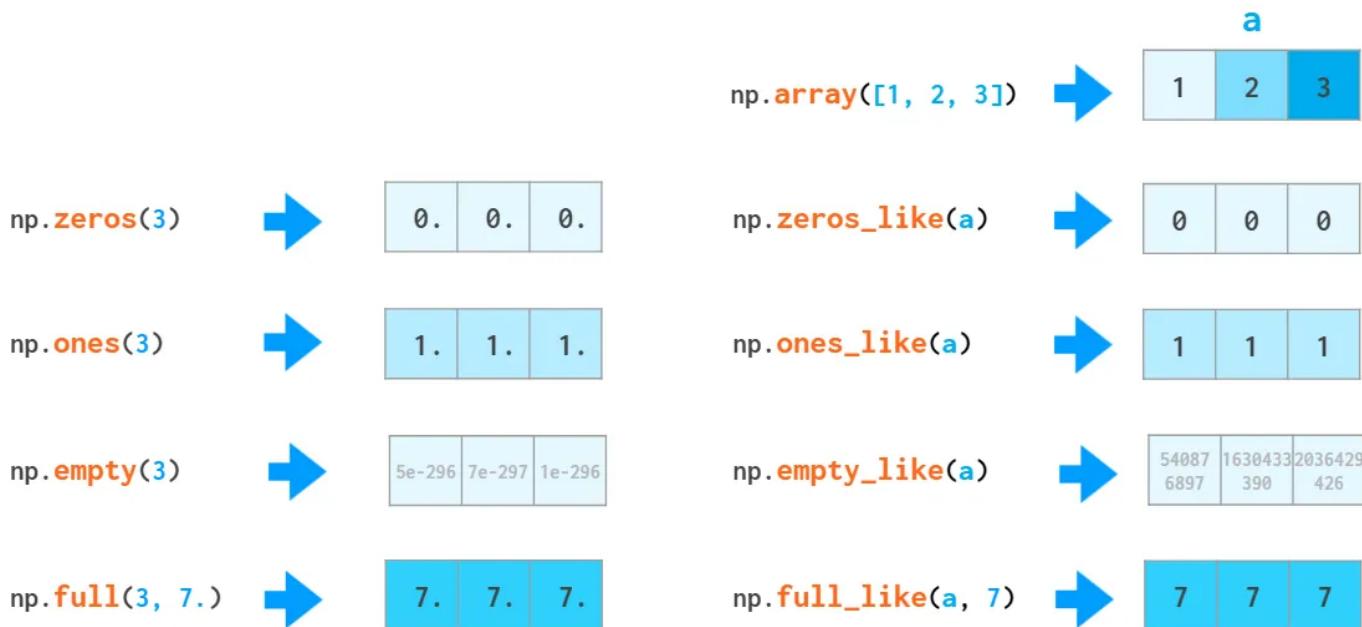
NumPy arrays cannot grow the way a Python list does: No space is reserved at the end of the array to facilitate quick appends. So it is a common practice to either grow a Python list and convert it to a NumPy array when it is ready or to preallocate the necessary space with `np.zeros` or `np.empty`:



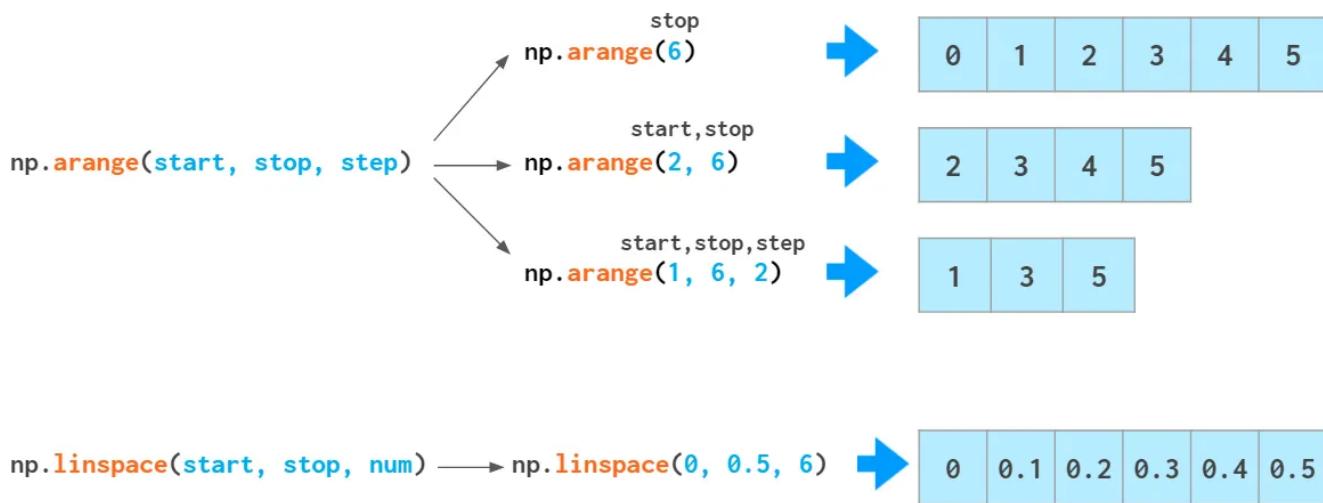
It is often necessary to create an empty array which matches the existing one by shape and elements type:



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

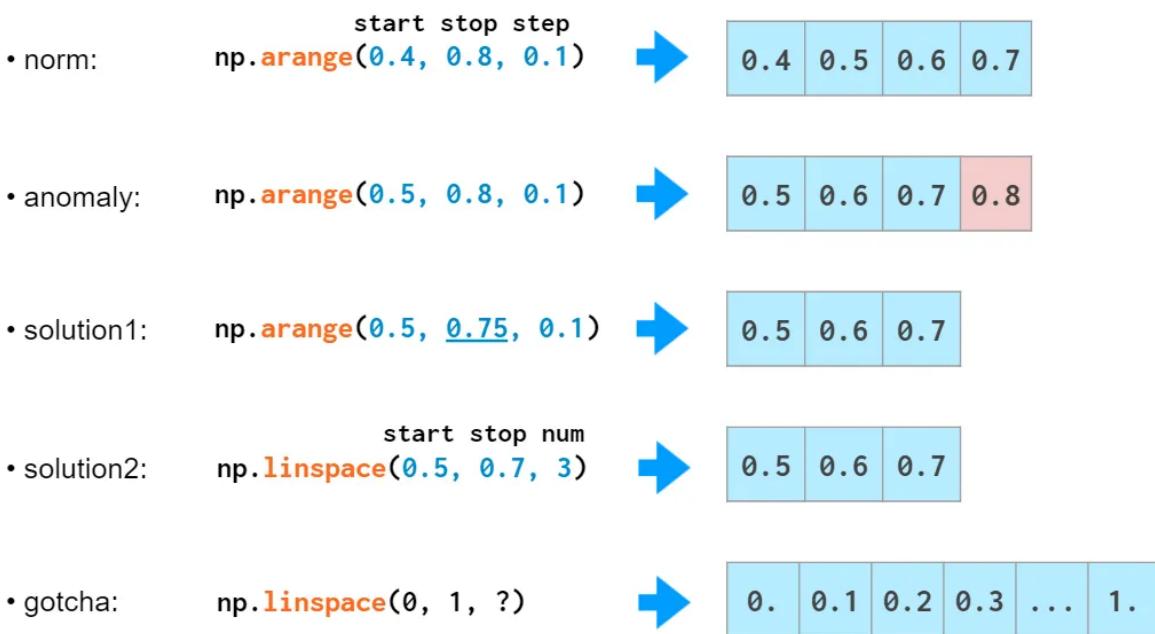


There are as many as two functions for array initialization with a monotonic sequence in NumPy:



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.

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

`np.random.randint(0, 10, 3) →`

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

Careful!

`np.random.randint(0, 10)` is $[0, 10)$, but
`random.randint(0, 10)` is $[0, 10]$

`np.random.rand(3) →`

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

`np.random.randn(3) →`

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

`np.random.uniform(1, 10, 3) →`

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

`np.random.normal(5, 2, 3) →`

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

Old-style random numbers generation (deprecated)

There's also a new interface for random arrays generation. It is:

- better suited for multi-threading,
- somewhat faster,
- more configurable (you can squeeze even more speed or even more quality by choosing a non-default so-called 'bit generator'),
- able to pass two tricky synthetic tests that the old version fails.

`rng = np.random.default_rng()`

`rng.integers(0, 10, 3) →`

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

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

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

`rng.random(3) →`

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

`rng.standard_normal(3) →`

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

`rng.uniform(1, 10, 3) →`

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

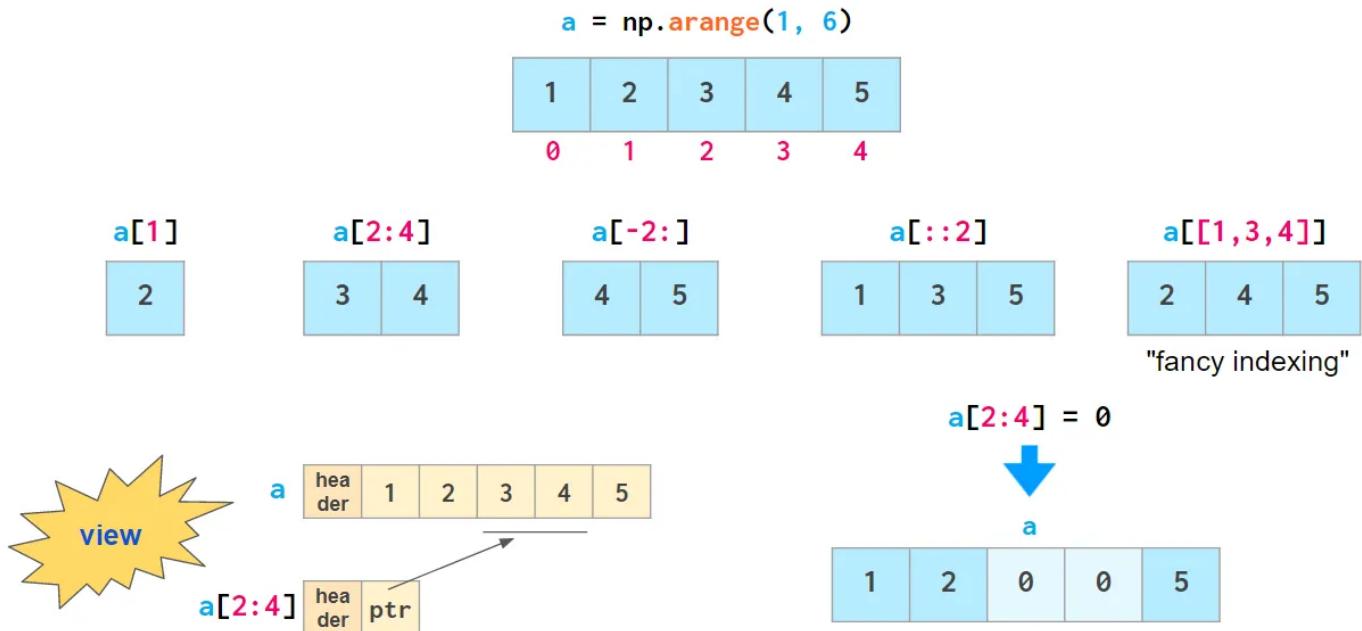
`rng.normal(5, 2, 3) →`

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

New-style random numbers generation

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

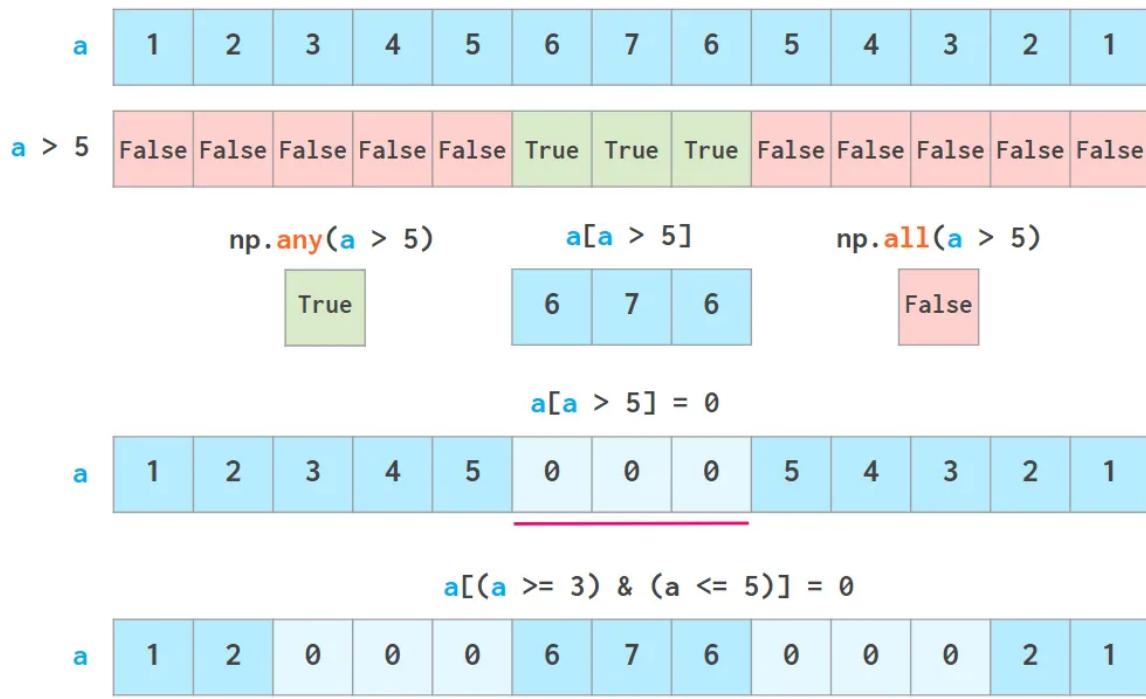
Also, such assignments must not change the size of the array, so tricks like

```
a = [1, 2, 3]
a[1:2] = [5, 6]
a
```

[1, 5, 6, 3]

won't work in NumPy — use `np.insert`, `np.append`, etc. instead (described in the "2D" section below).

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.

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`.

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)
```

$a[a < 5] = 0; a[a \geq 5] = 1$

a	0	0	0	0	1	1	1	1	1	0	0	0	0
$= \text{np.where}(a \geq 5, 1, 0)$													

$a[a < 2] = 2; a[a > 5] = 5$

a	2	2	3	4	5	5	5	5	5	4	3	2	2
$= \text{np.clip}(a, 2, 5)$													

Note that `np.where` with one argument returns a tuple of arrays (1-tuple in 1D case, 2-tuple in 2D case, etc), thus you need to write `np.where(a>5)[0]` to get `np.array([5,6,7])` in the example above (same for `np.nonzero`).

For a single-side clipping there are `np.minimum` and `np.maximum`. All the three clipping functions also accept an array as a threshold.

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:

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 4 & 8 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 5 & 10 \\ \hline \end{array}$$

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

$$\begin{array}{|c|c|} \hline 4 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 2 & 5 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 8 & 40 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 4 & 8 \\ \hline \end{array} / \begin{array}{|c|c|} \hline 2 & 5 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2.0 & 1.6 \\ \hline \end{array} \text{ np.float64}$$

$$\begin{array}{|c|c|} \hline 4 & 8 \\ \hline \end{array} // \begin{array}{|c|c|} \hline 2 & 5 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 1 \\ \hline \end{array} \text{ np.int32}$$

$$\begin{array}{|c|c|} \hline 3 & 4 \\ \hline \end{array} ** \begin{array}{|c|c|} \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 9 & 64 \\ \hline \end{array}$$

As usual in Python, $a//b$ means $a \text{ div } b$ (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:

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

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

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} * \begin{array}{|c|} \hline 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 3 & 6 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} / \begin{array}{|c|} \hline 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0.33 & 0.67 \\ \hline \end{array} \text{ np.float64}$$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} // \begin{array}{|c|} \hline 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline \end{array} \text{ np.int32}$$

$$\begin{array}{|c|c|} \hline 3 & 4 \\ \hline \end{array} ** \begin{array}{|c|} \hline 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 9 & 16 \\ \hline \end{array}$$

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

$$a^2 = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline \end{array} \quad ** \quad 2 = \begin{array}{|c|c|} \hline 4 & 9 \\ \hline \end{array}$$

$$\sqrt{a} = \text{np.sqrt}\left(\begin{array}{|c|c|} \hline 4 & 9 \\ \hline \end{array}\right) = \begin{array}{|c|c|} \hline 2. & 3. \\ \hline \end{array}$$

$$e^a = \text{np.exp}\left(\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array}\right) = \begin{array}{|c|c|} \hline 2.72 & 7.39 \\ \hline \end{array}$$

$$\ln a = \text{np.log}\left(\begin{array}{|c|c|} \hline \text{np.e} & \text{np.e}^{**2} \\ \hline \end{array}\right) = \begin{array}{|c|c|} \hline 1. & 2. \\ \hline \end{array}$$

Scalar product has an operator of its own:

$$\vec{a} \cdot \vec{b} = \text{np.dot}\left(\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array}, \begin{array}{|c|c|} \hline 3 & 4 \\ \hline \end{array}\right)$$

$$= \begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} @ \begin{array}{|c|c|} \hline 3 & 4 \\ \hline \end{array} = \begin{array}{|c|} \hline 11 \\ \hline \end{array}$$

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

You don't need loops for trigonometry either:

$$\text{np.sin}(\begin{array}{|c|c|}\hline \text{np.pi} & \text{np.pi}/2 \\ \hline \end{array}) = \begin{array}{|c|c|}\hline 0. & 1. \\ \hline \end{array}$$

$$\text{np.arcsin}(\begin{array}{|c|c|}\hline 0. & 1. \\ \hline \end{array}) = \begin{array}{|c|c|}\hline 0. & 1.57 \\ \hline \end{array}$$

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

$$\text{np.arctan2}(\begin{array}{|c|c|} \hline \text{y} & \text{x} \\ \hline 0. & 1. \\ \hline \end{array}, \begin{array}{|c|c|} \hline \text{x} & \text{y} \\ \hline 1. & 0. \\ \hline \end{array}) = \begin{array}{|c|c|} \hline 0. & 1.57 \\ \hline \end{array}$$

Inverse functions are named differently from `math`: np.arcsin but math.asin

Arrays can be rounded as a whole:

$$\text{np.floor}(\begin{array}{|c|c|c|c|} \hline 1.1 & 1.5 & 1.9 & 2.5 \\ \hline \end{array}) = \begin{array}{|c|c|c|c|} \hline 1. & 1. & 1. & 2. \\ \hline \end{array}$$

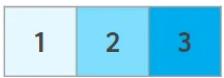
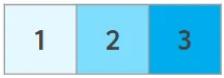
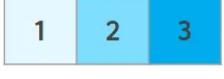
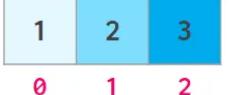
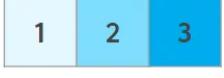
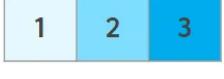
$$\text{np.ceil}(\begin{array}{|c|c|c|c|} \hline 1.1 & 1.5 & 1.9 & 2.5 \\ \hline \end{array}) = \begin{array}{|c|c|c|c|} \hline 2. & 2. & 2. & 3. \\ \hline \end{array}$$

$$\text{np.round}(\begin{array}{|c|c|c|c|} \hline 1.1 & 1.5 & 1.9 & 2.5 \\ \hline \end{array}) = \begin{array}{|c|c|c|c|} \hline 1. & 2. & 2. & 2. \\ \hline \end{array}$$

floor rounds to $-\infty$, ceil to $+\infty$ 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:

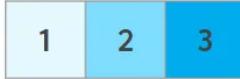
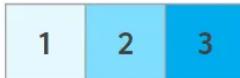
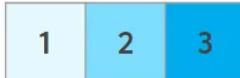
a		$\text{np.max}() =$	
		$.max() =$	
		$.min() =$	
			
		$.argmax() =$	
		$.argmin() =$	
		$.sum() =$	
		$.mean() =$	
		$.var() =$	
		$.std() =$	

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

 $a = 2 \pm 0.82$

Each of these functions has a nan-resistant variant: eg [nansum](#), [nanmax](#), etc

As you can see from the formula above, both `std` and `var` ignore [Bessel's correction](#) and give a biased result in the most typical use case of estimating std from a sample when the population mean is unknown. The standard approach to get a less biased estimation is to have `n-1` in the denominator, which is done with `ddof=1` ('delta degrees of freedom'):

$\text{np.array}($		$).std() =$		$\sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2}$
$\text{np.array}($		$).std(ddof=1) =$		$\sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2}$
$\text{pd.Series}($		$).std() =$		

Pandas `std` uses Bessel's correction by default

The effect of the Bessel's correction quickly diminishes with increasing sample size. Also, it is not a one-size-fits-all solution, e.g. for the normal distribution `ddof=1.5` is better:

```
res1, res2, res3 = [], [], []
for i in range(10000):
    a = np.random.randn(100)
    res1.append(a.std())
    res2.append(a.std(ddof=1))
    res3.append(a.std(ddof=1.5))
```

<code>np.mean(res1)</code>	# 0.8%
0.9921820984811617	
<code>np.mean(res2)</code>	# 0.3%
0.9971805285968072	
<code>np.mean(res3)</code>	# 0.03%
0.9997082399489409	

Sorting functions have fewer capabilities than their Python counterparts:

python lists

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

numpy arrays

<code>a.sort()</code>	<code>np.sort(a)</code>	sorts in-place
<code>sorted(a)</code>		returns new sorted array
<code>a.sort(key=f)</code>	-	key function
<code>a.sort(reversed=False)</code>	-	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 request⁵](#)).

Searching for an element in a vector

As opposed to Python lists, a NumPy array does not have an `index` method. The corresponding [feature request⁶](#) 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 Numba⁷ (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.

Comparing floats

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

<code>0.1 + 0.2 == 0.3</code>	<code>np.allclose(0.1 + 0.2, 0.3)</code>	<code>math.isclose(0.1 + 0.2, 0.3)</code>
False !!!	True	True
<code>1e-9 == 2e-9</code>	<code>np.allclose(1e-9, 2e-9)</code>	<code>math.isclose(1e-9, 2e-9)</code>
False	True !!!	False
<code>0.1+0.2-0.3 == 0</code>	<code>np.allclose(0.1+0.2-0.3, 0)</code>	<code>math.isclose(0.1+0.2-0.3, 0)</code>
False	True	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`.

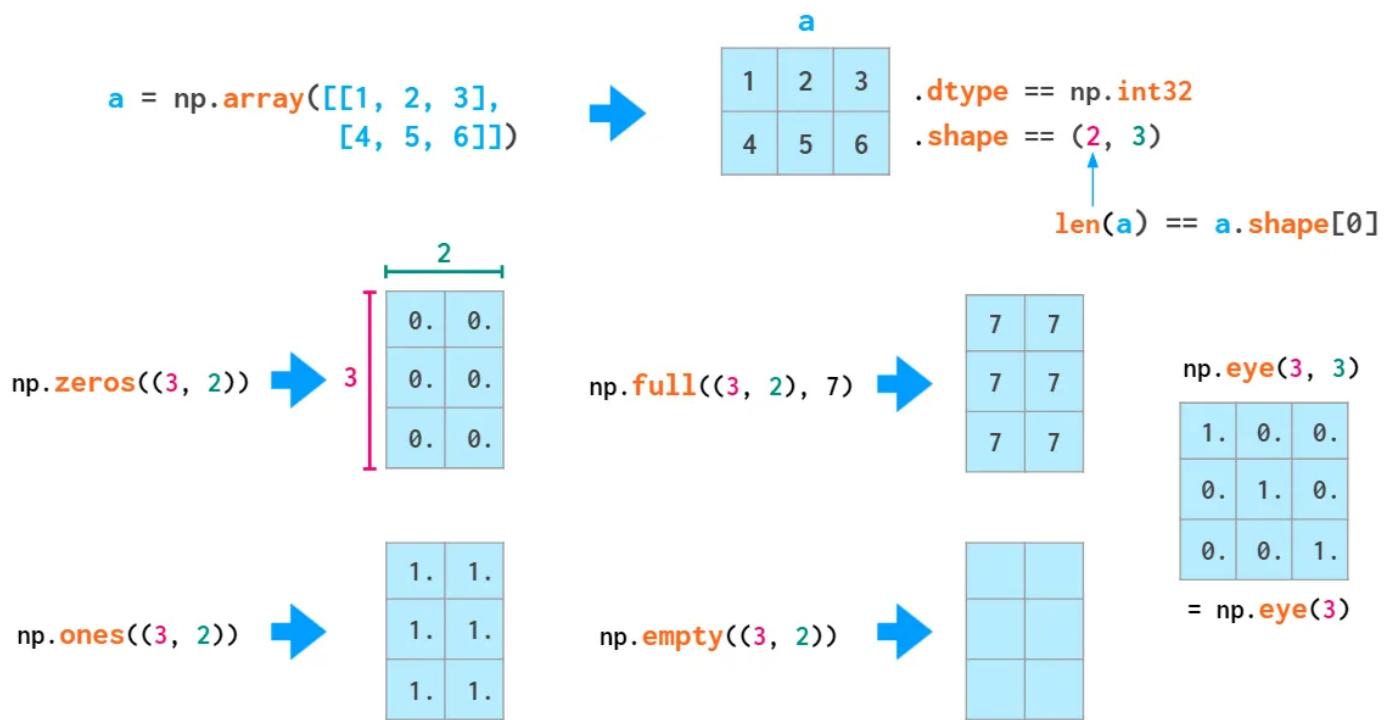
Aside from that, `np.allclose` has some minor issues in a formula for absolute and relative tolerances, for example, for certain `a,b` `allclose(a, b) != allclose(b, a)`. Those issues were resolved in the (scalar) function `math.isclose` (which was introduced

later). To learn more on that, take a look at the excellent [floating-point guide⁸](#) and the corresponding NumPy [issue⁹](#) on GitHub.

2. Matrices, the 2D Arrays

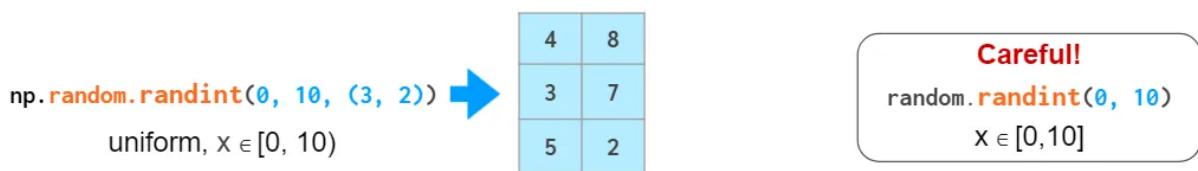
There used to be a dedicated `matrix` class in NumPy, but it is deprecated now, so I'll use the words `matrix` and `2D array` interchangeably.

Matrix initialization syntax is similar to that of vectors:



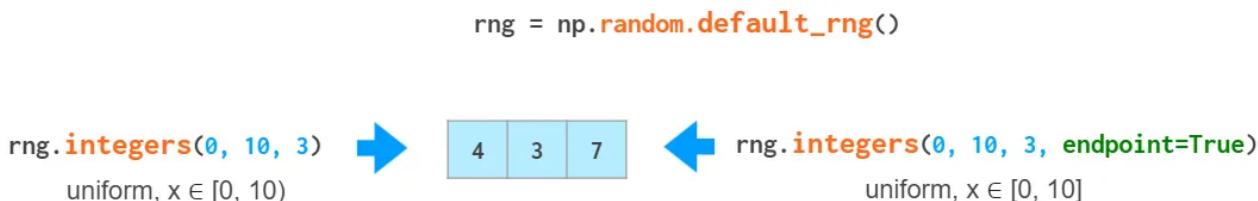
Double parentheses are necessary here because the second positional parameter is reserved for the (optional) `dtype` (which also accepts integers).

Random matrix generation is also similar to that of vectors:



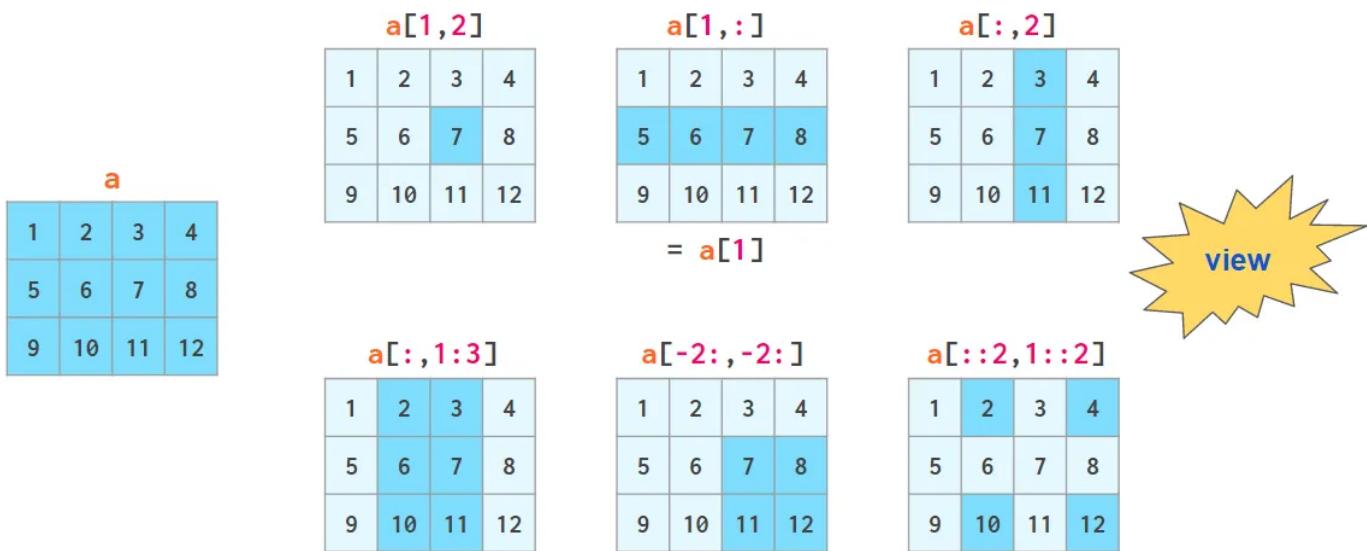
Old-style random numbers generation

The ubiquitous double parentheses found their way to the interface of the new-style (see details in the 1D section above) numbers generation routines so that as of today only in `np.eye` has beauty taken prevalence over stringency and convenience of passing shapes as is:



New-style random numbers generation

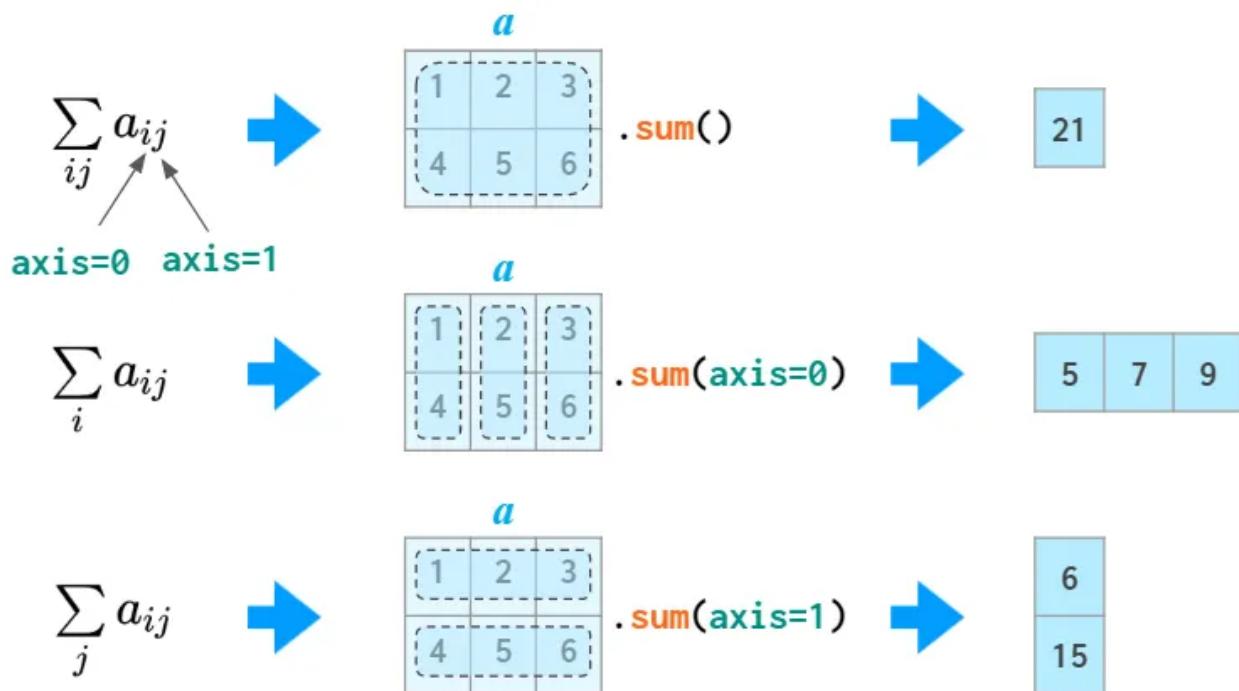
Two-dimensional indexing syntax is more convenient than that of nested lists:



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.

The axis argument

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.



Generally speaking, the first dimension `i` (`axis=0`) is responsible for indexing the rows, so `sum(axis=0)` should be read like “for any given column sum over all of its rows” rather than just “column-wise”. The 2D case is somewhat counter-intuitive: you need to specify the dimension *to be eliminated*, instead of the *remaining* one you would normally think about. In higher dimensional cases this is more natural, though: it’d be a burden to enumerate all the remaining dimensions if you only need to sum over a single one.

The notion of the ‘axis’ argument (called ‘dimension’ in MATLAB) is actually very simple: it is just number of the index you want the operation to carry over. It is used in many functions in NumPy so it pays off to understand it, but if for some reason it doesn’t work out for you, in this particular case you can use the Einstein summation — that’s a way to do the sums without ‘axes’: `np.einsum('ij->j', a)` is summing column-wise (same as `sum(axis=0)` in the image above) and `np.einsum('ij->i', a)` is row-wise (same as `sum(axis=1)`).

Matrix arithmetic

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

$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{orange} +$	$\begin{array}{ c c } \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 2 & 2 \\ \hline 3 & 5 \\ \hline \end{array}$				
$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{orange} -$	$\begin{array}{ c c } \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 0 & 2 \\ \hline 3 & 3 \\ \hline \end{array}$				
				$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{red} *$	$\begin{array}{ c c } \hline 2 & 0 \\ \hline 0 & 2 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 2 & 0 \\ \hline 0 & 8 \\ \hline \end{array}$
				$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{red} @$	$\begin{array}{ c c } \hline 2 & 0 \\ \hline 0 & 2 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 2 & 4 \\ \hline 6 & 8 \\ \hline \end{array}$
				$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{red} /$	$\begin{array}{ c c } \hline 2 & 1 \\ \hline 1 & 2 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 0.5 & 2. \\ \hline 3. & 2. \\ \hline \end{array}$
				$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$	$\color{red} **$	$\begin{array}{ c c } \hline 2 & 1 \\ \hline 1 & 2 \\ \hline \end{array}$	$=$	$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 16 \\ \hline \end{array}$

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:

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \text{ / } \begin{array}{|c|c|c|} \hline 9 & 9 & 9 \\ \hline 9 & 9 & 9 \\ \hline 9 & 9 & 9 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline .1 & .2 & .3 \\ \hline .4 & .5 & .7 \\ \hline .8 & .9 & 1. \\ \hline \end{array}$$

normalization

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

multiplying several columns at once

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} / \begin{array}{|c|c|c|} \hline 3 & 3 & 3 \\ \hline 6 & 6 & 6 \\ \hline 9 & 9 & 9 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline .3 & .7 & 1. \\ \hline .6 & .8 & 1. \\ \hline .8 & .9 & 1. \\ \hline \end{array}$$

row-wise normalization

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline 3 & 3 & 3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 2 & 4 & 6 \\ \hline 3 & 6 & 9 \\ \hline \end{array}$$

outer product

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:

$$\begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} @ \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 2 & 4 & 6 \\ \hline 3 & 6 & 9 \\ \hline \end{array}$$

outer product

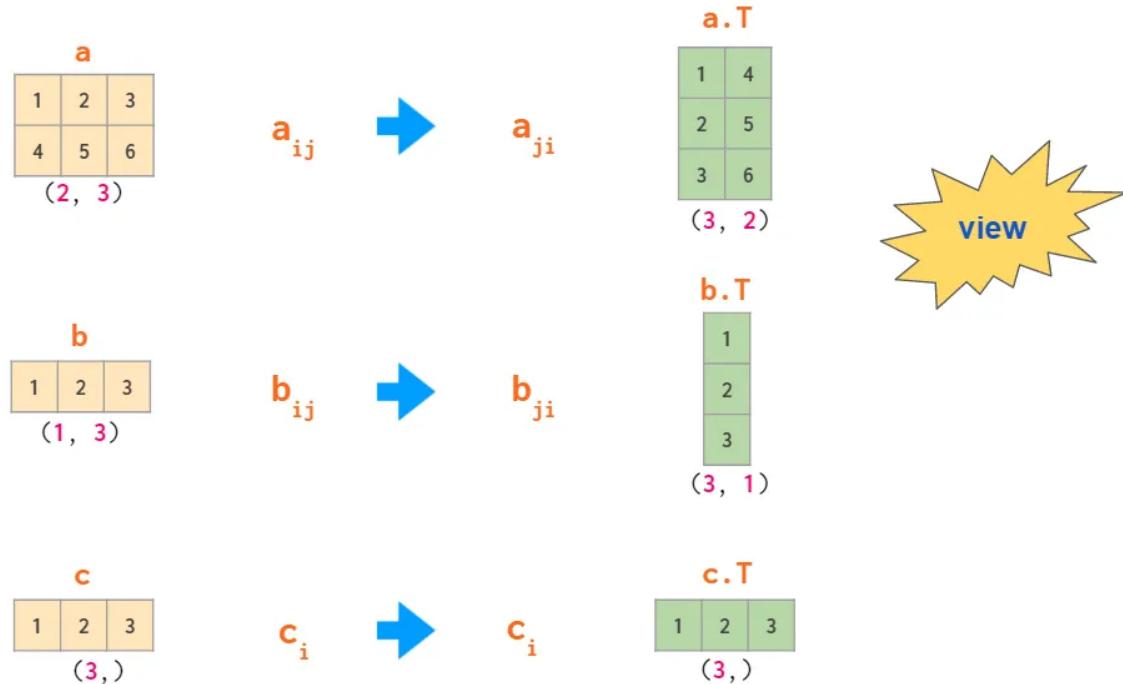
$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} @ \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} = \boxed{14}$$

inner (or dot) product

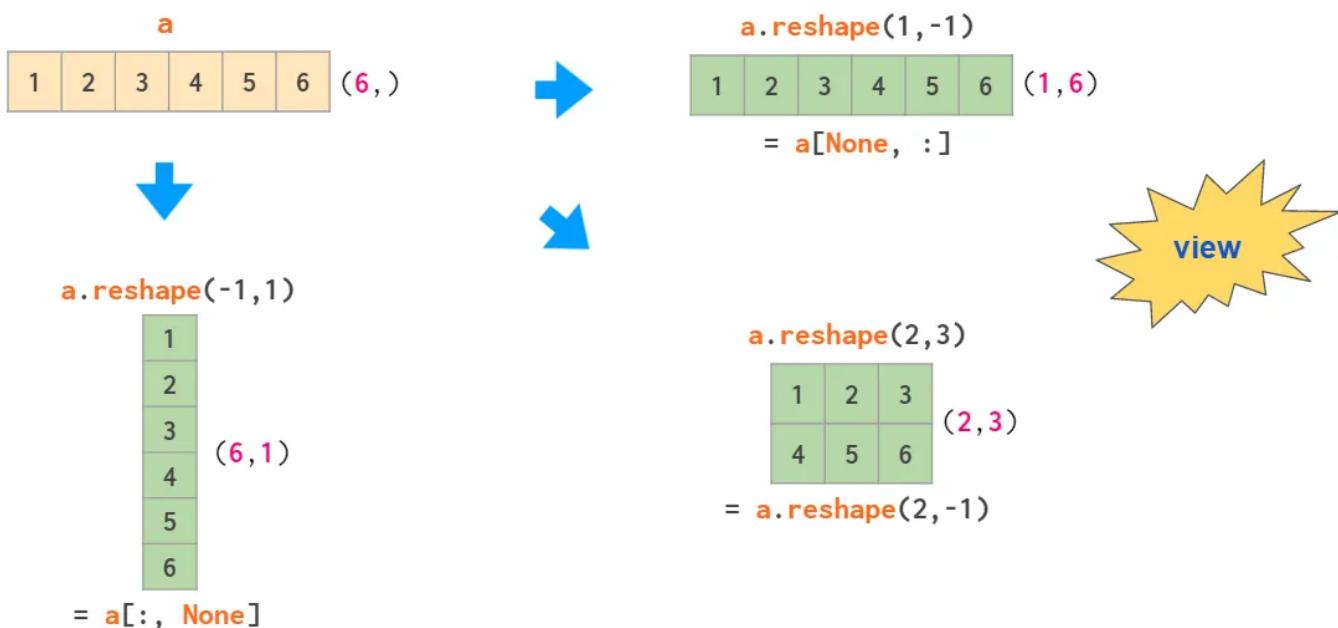
Row vectors and column vectors

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:

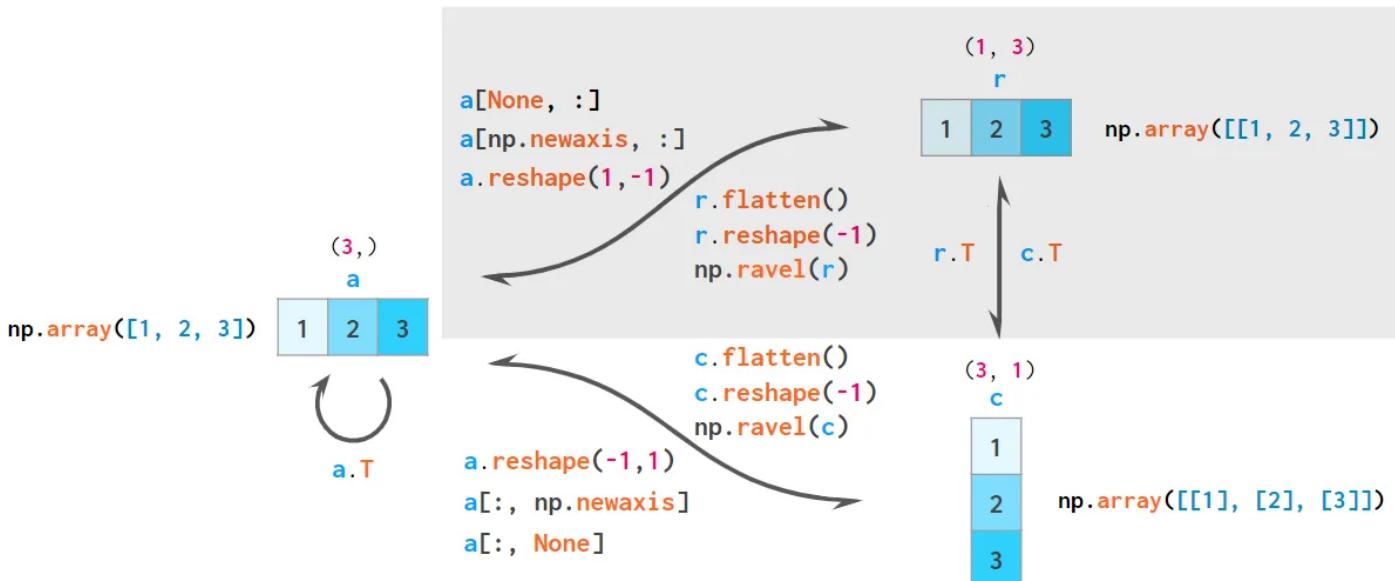


Two operations that *are* capable of making a 2D column vector out of a 1D array are reshaping and indexing with `newaxis`:



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. Yet another way to do it is through `np.expand_dims(a, axis=0)` where 0 tells NumPy where you want the new axis to appear.

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:



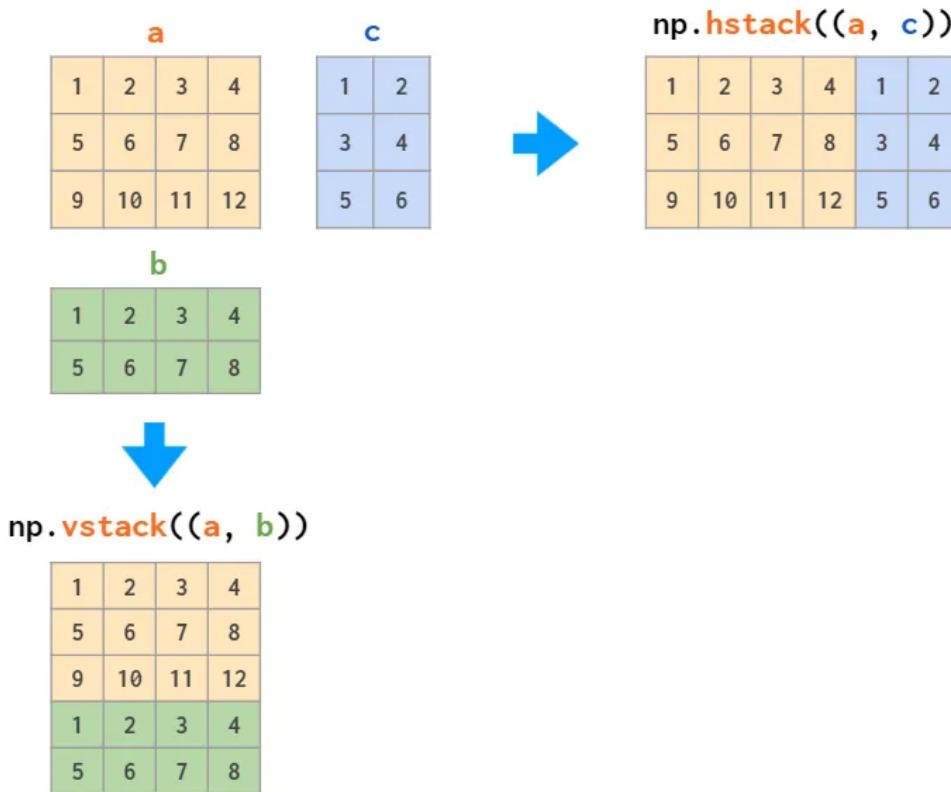
flatten is always a copy, ravel() is usually a view, reshape(-1) is a view when possible, .shape= is always a view (see “[5 ways to flatten an array](#)” for details)

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.

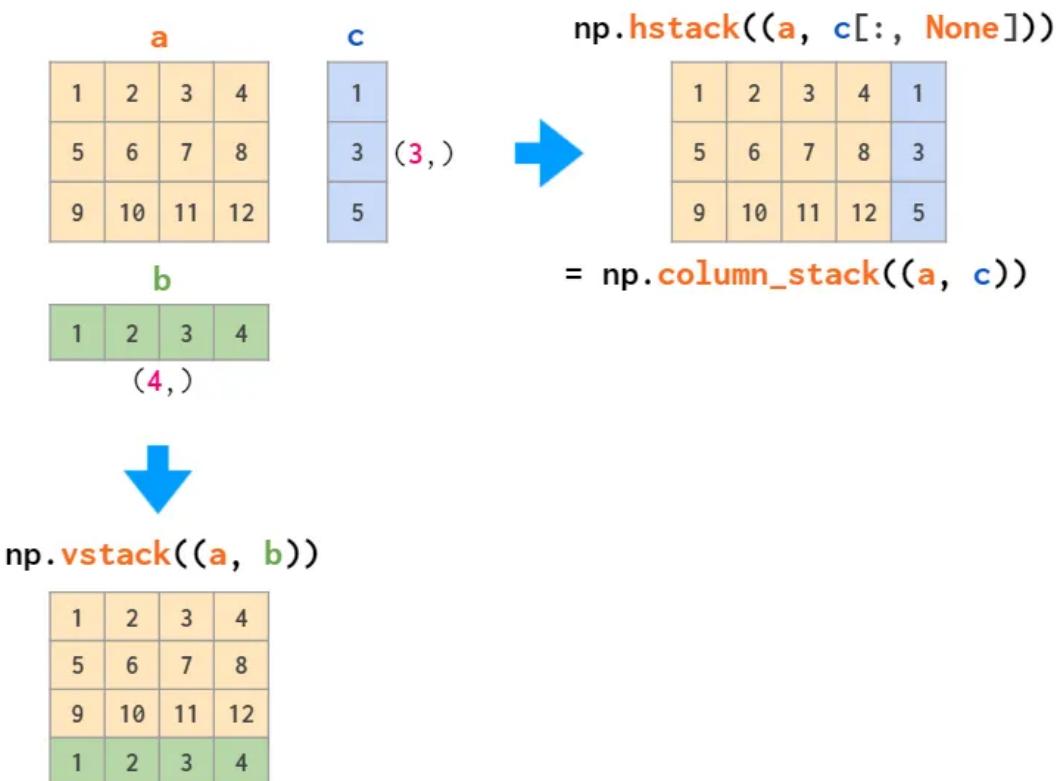
Strictly speaking, any array, all but one dimensions of which are single-sized, is a vector (eg. `a.shape==[1,1,1,5,1,1]`), so there's an infinite number of vector types in numpy, but only these three are commonly used. You can use `np.reshape` to convert a 'normal' 1D vector to this form and `np.squeeze`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

Matrix manipulations

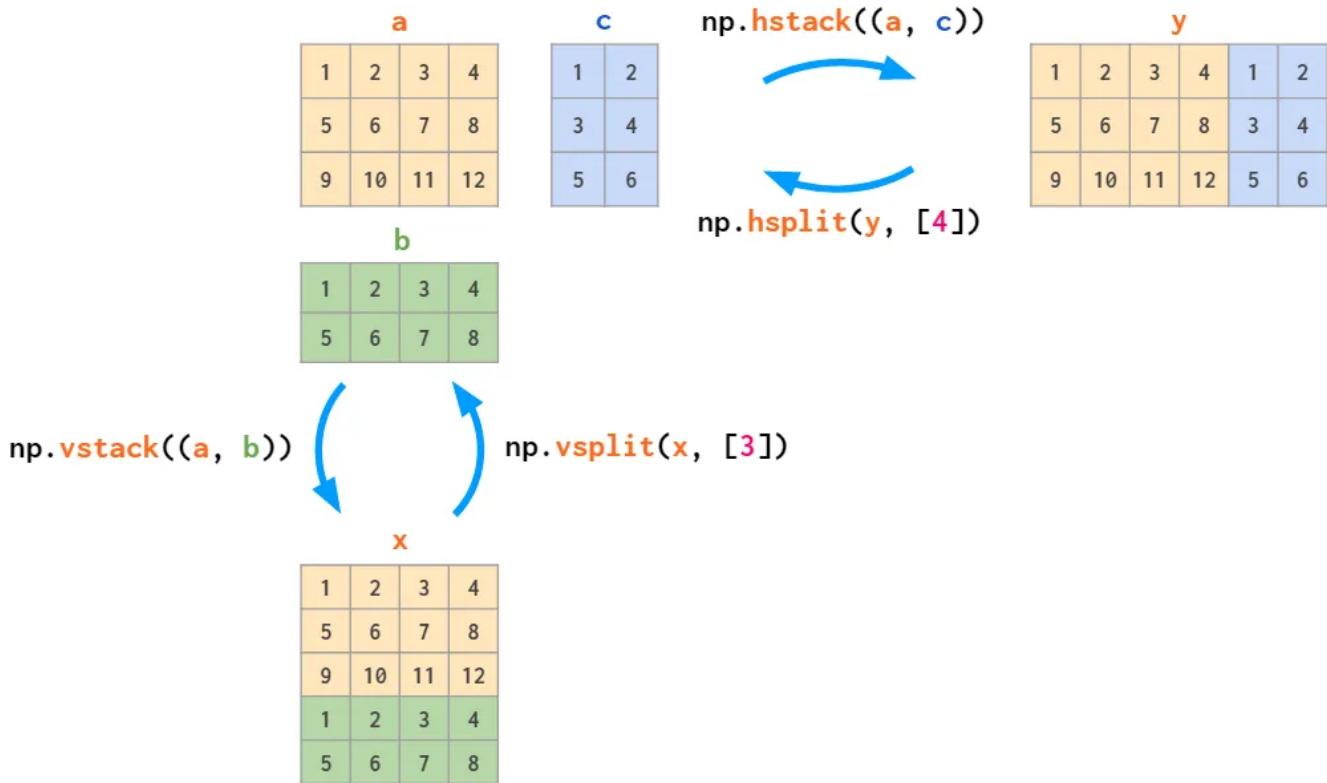
There are two main functions for joining the arrays:



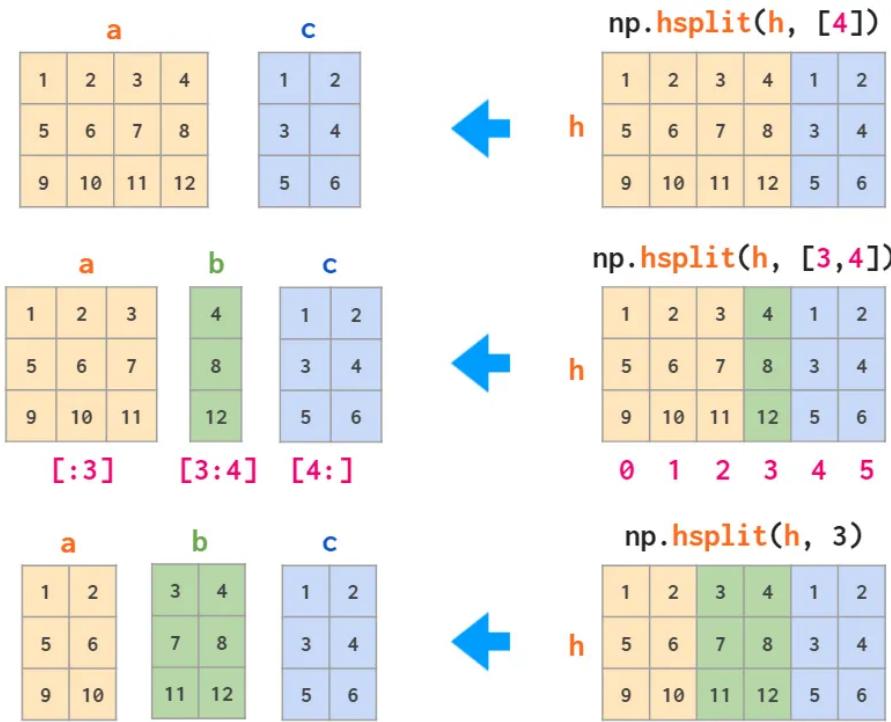
Those two work fine with stacking matrices only or vectors only, but when it comes to mixed stacking of 1D arrays and matrices, only the `vstack` works as expected: The `hstack` generates a dimensions-mismatch error because as described above, the 1D array is interpreted as a row vector, not a column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



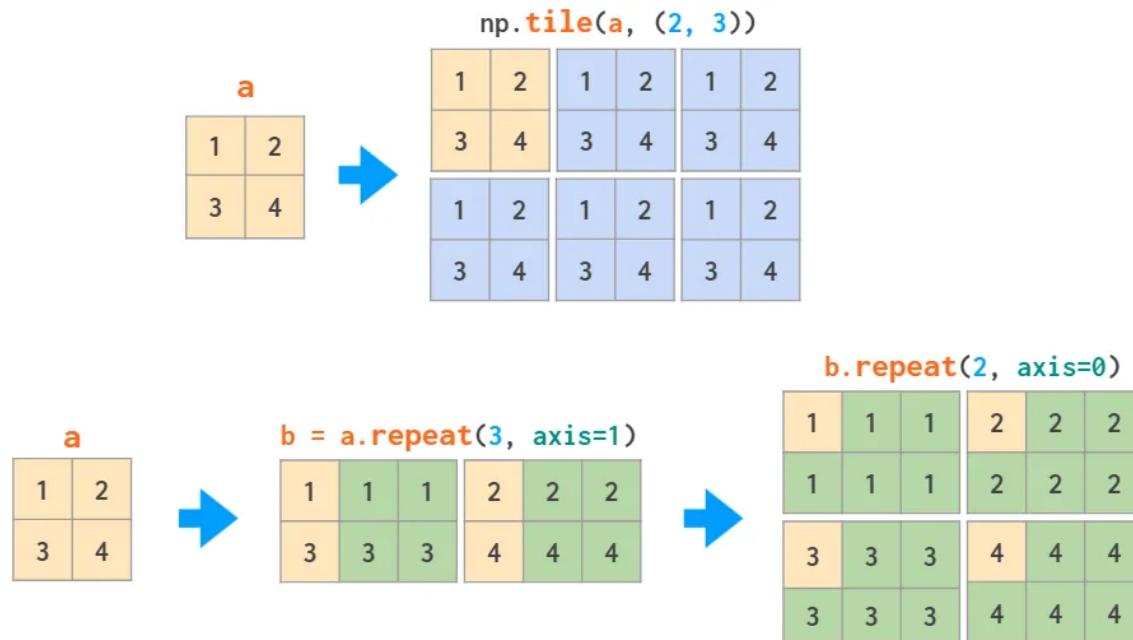
The inverse of stacking is splitting:



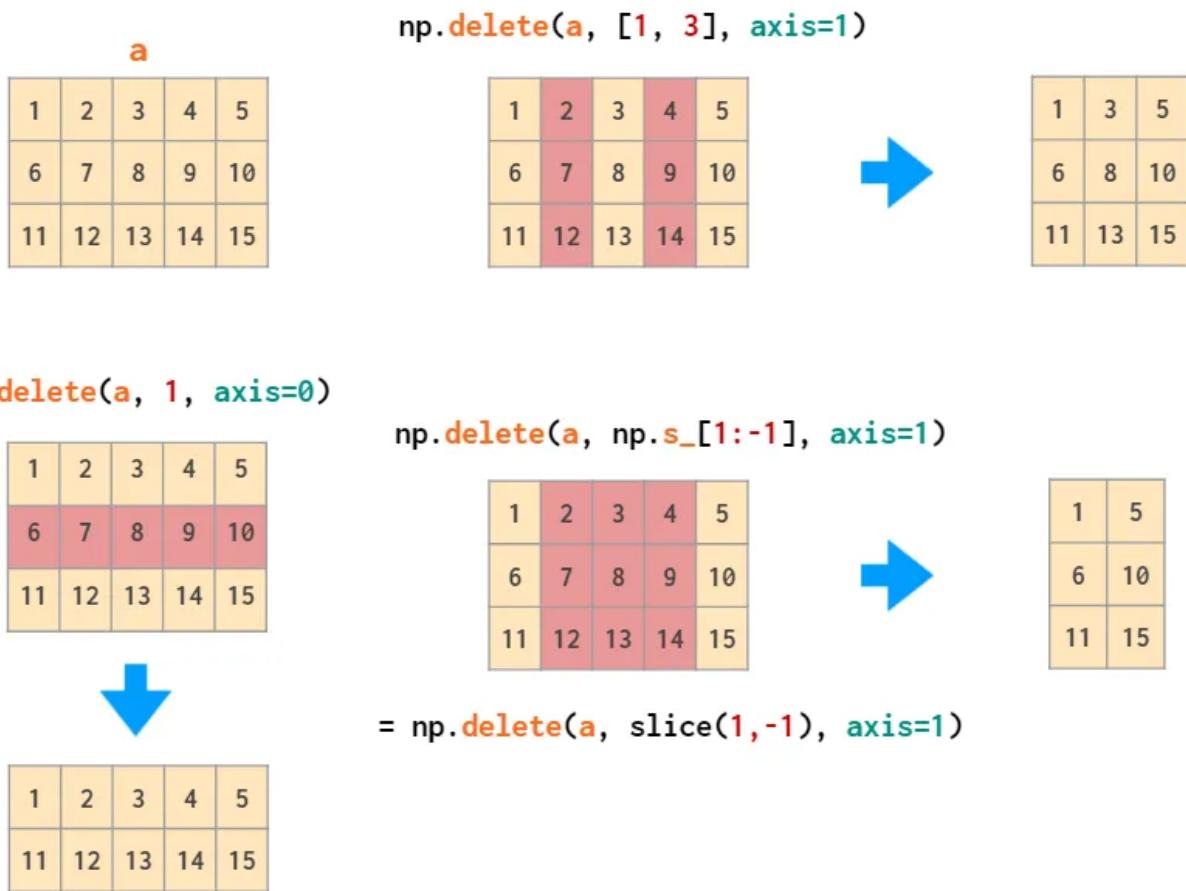
All `split` flavors accept either a list of indices to split at, or a single integer, the number of equal sized parts:



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



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



The inverse operation is `insert`:

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

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

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



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`

`w`

1	5
6	10
11	15

2	3	4
7	8	9
12	13	14

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`

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

`np.append(a, np.zeros(3,2), axis=1) == np.column_stack(a, np.zeros(3))`



1	2	3	4	5	0	0
6	7	8	9	10	0	0
11	12	13	14	15	0	0



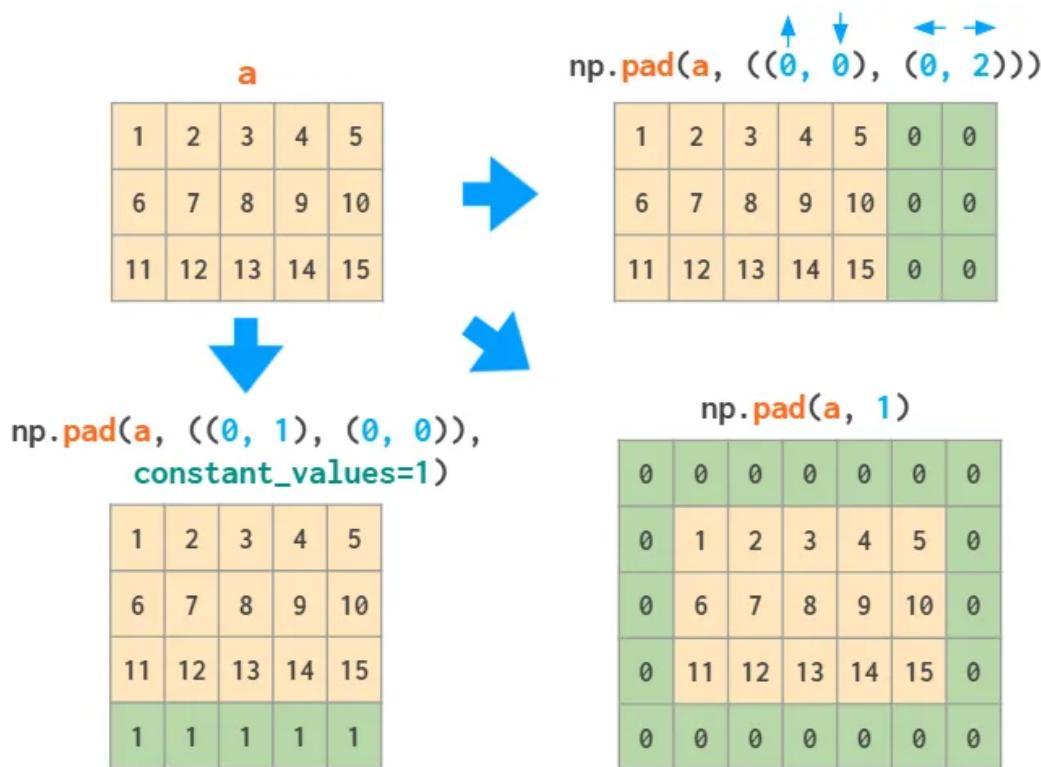
1	2	3	4	5	0
6	7	8	9	10	0
11	12	13	14	15	0
1	1	1	1	1	1

`np.append(a, np.ones(5), axis=0)`

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

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:



Meshgrids

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

1. The c way

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

0	1	2
-1	0	1

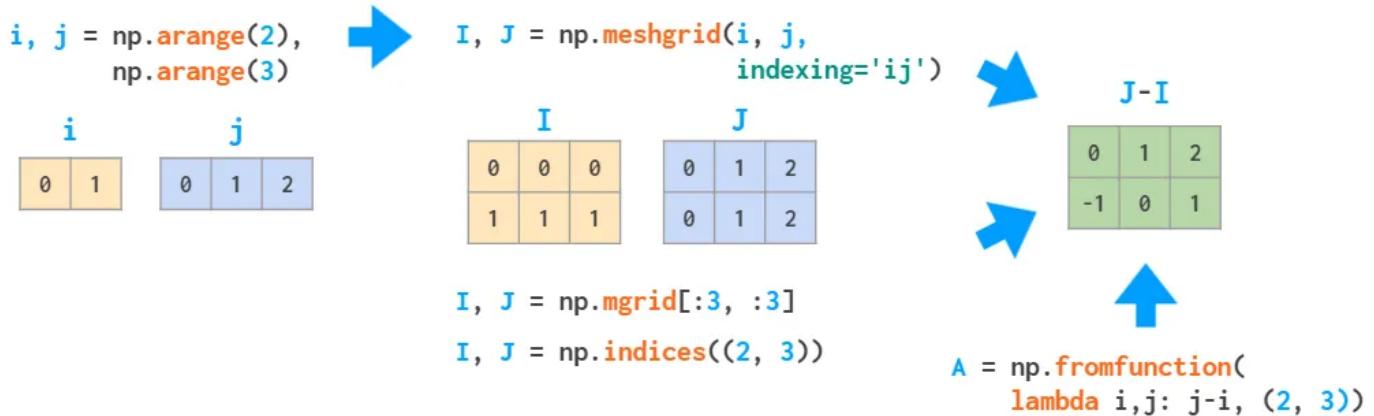
```
a = np.empty((2, 3))
for i in range(2):
    for j in range(3):
        a[i, j] = j - i
```

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:

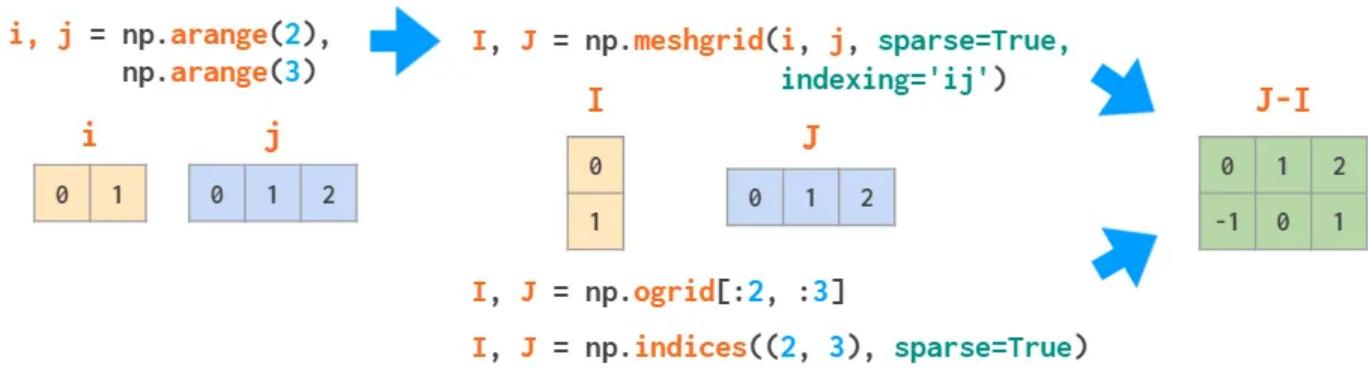
3. The matlab way



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.

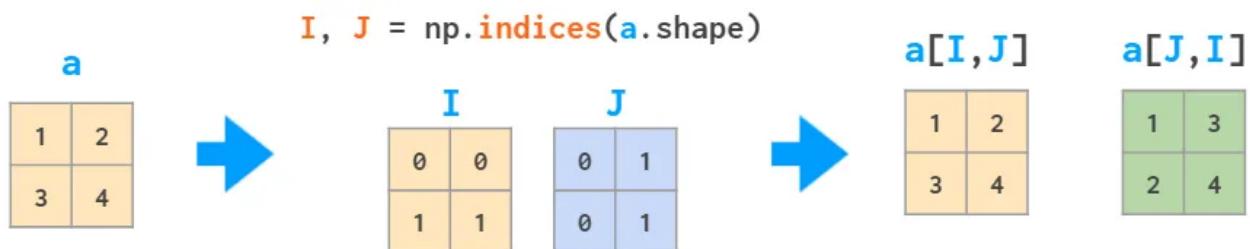
But actually, there is a better way to do it in NumPy. There's no need to spend memory on the whole `I` and `J` matrices (even though `meshgrid` is smart enough to only store references to the original vectors if possible). It is sufficient to store only vectors of the correct shape, and the broadcasting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

4. The numpy way



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](#)).

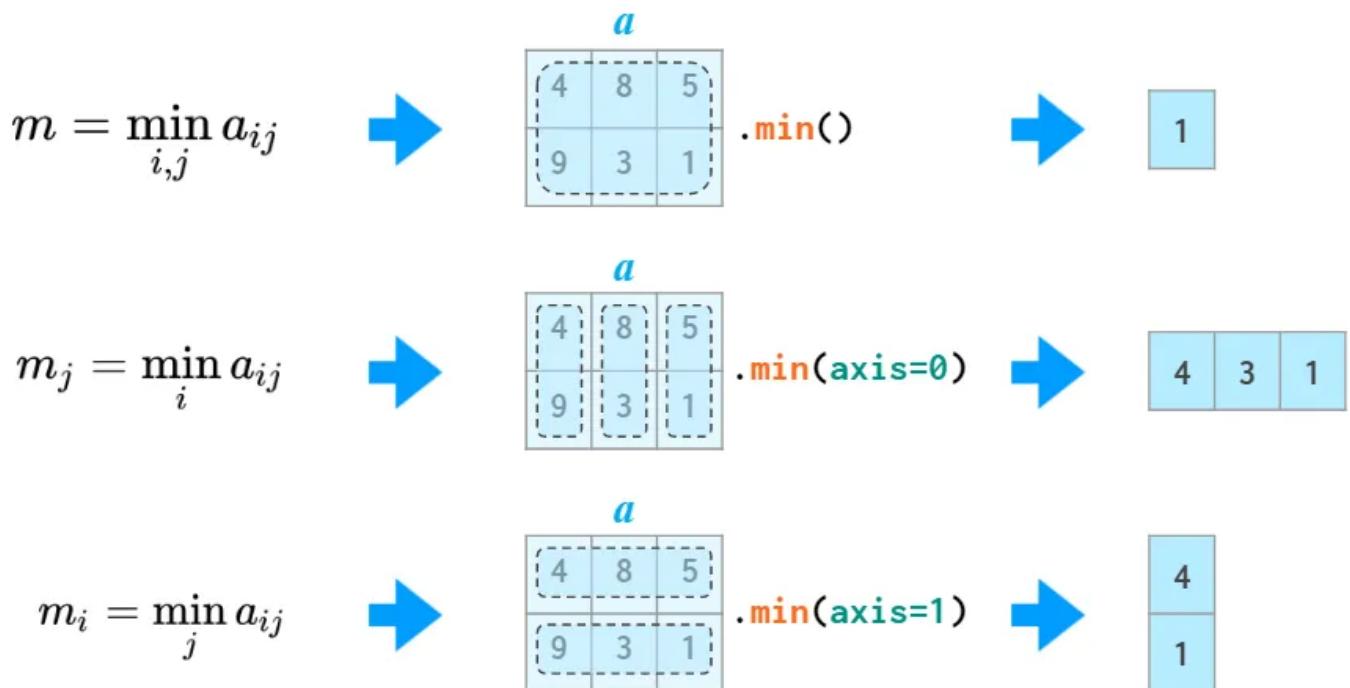
Aside from initializing functions over a two- or three-dimensional grid, the meshes can be useful for indexing arrays:



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 *`'

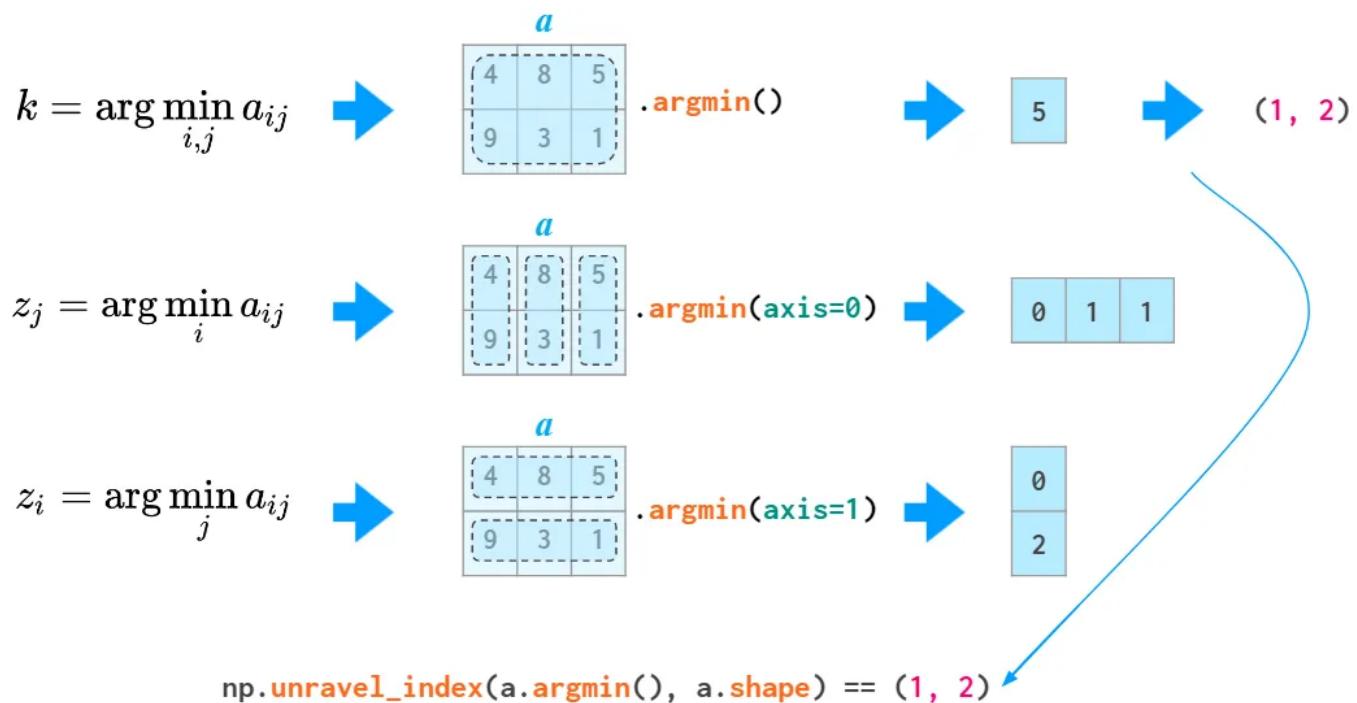
The `argmin` and `argmax` functions in 2D and above have an annoyance of returning the flattened index (of the first instance of the min and max value). To convert it to two

Open in app ↗

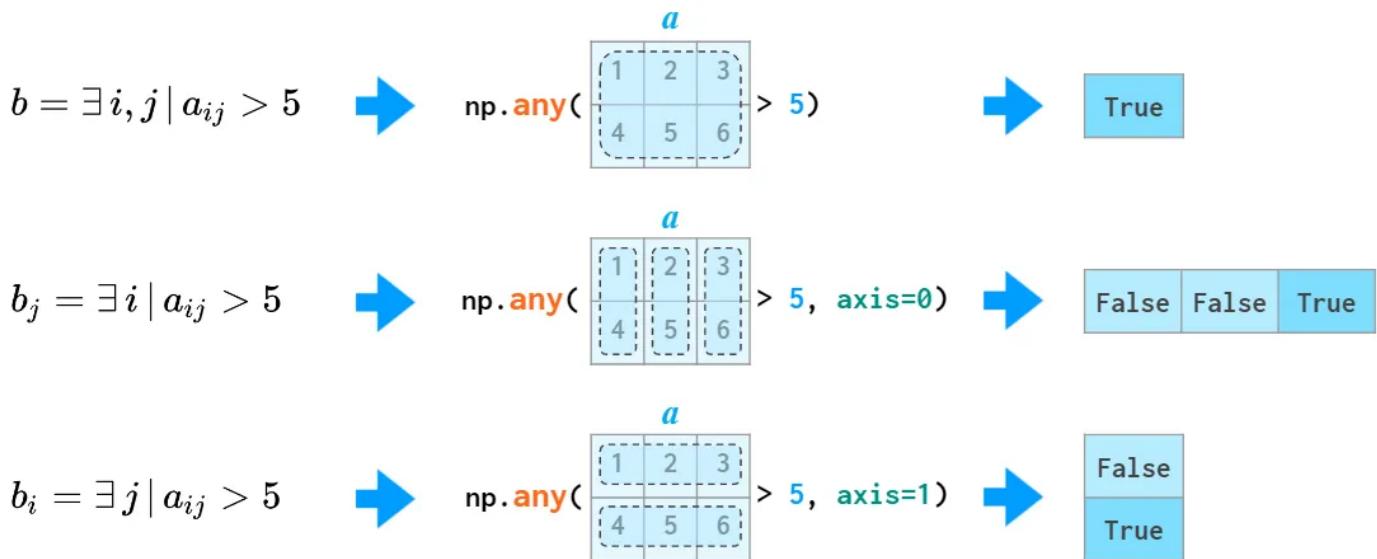


Search Medium





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



Matrix sorting

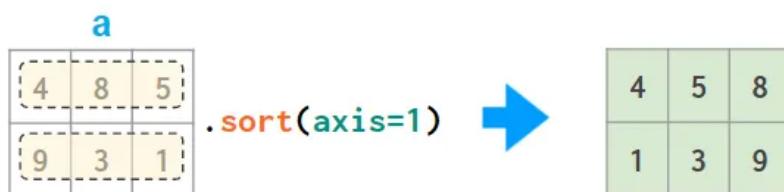
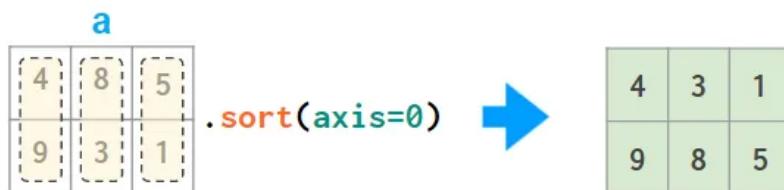
As helpful as the `axis` argument is for the functions listed above, it is as unhelpful for the 2D sorting:

python lists

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

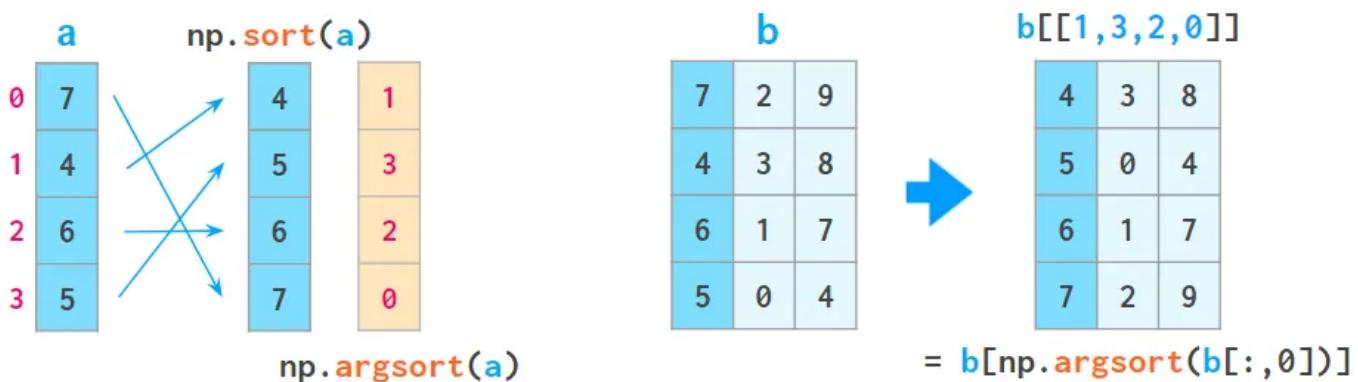
numpy arrays

`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



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:

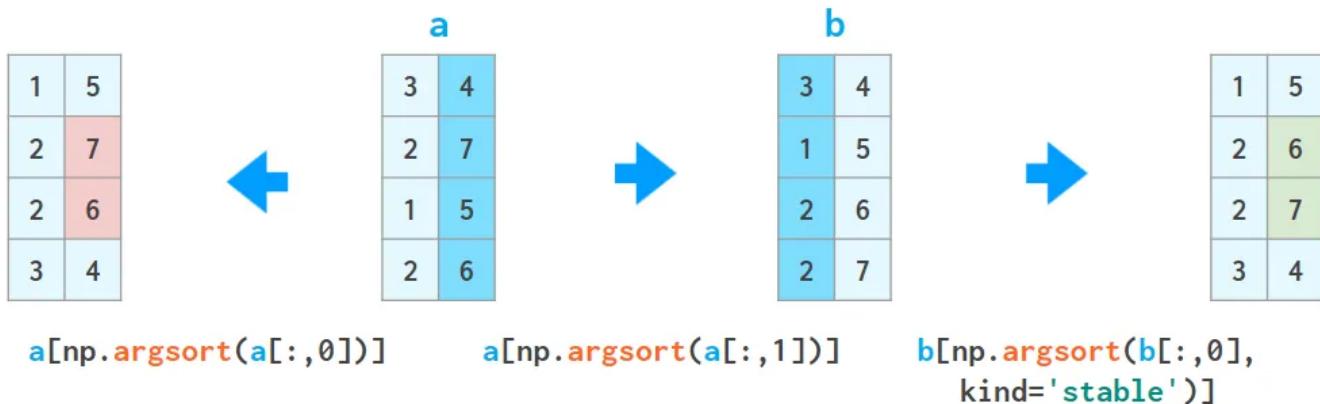
1. `a[a[:,0].argsort()]` sorts the array by the first column:



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

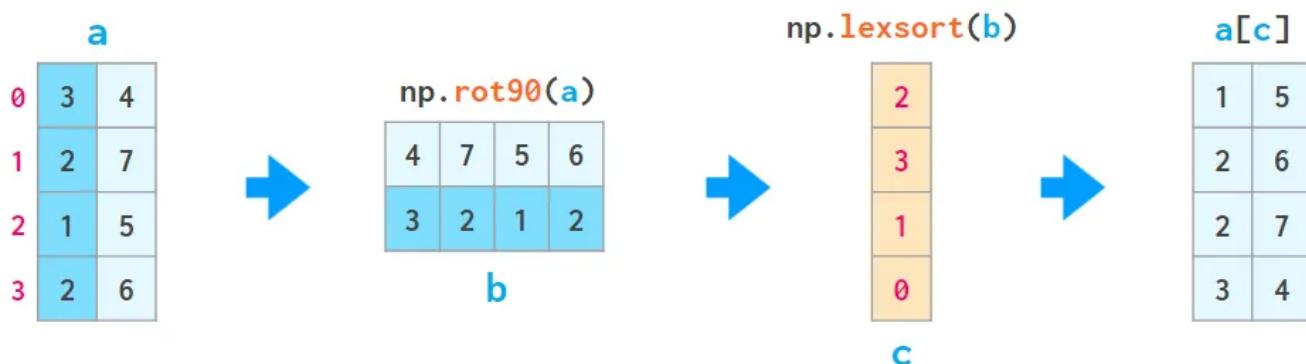
This trick can be repeated, but care must be taken so that the next sort does not mess up the results of the previous one:

```
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.rot90(a))]` sorts by all columns in left-to-right order.

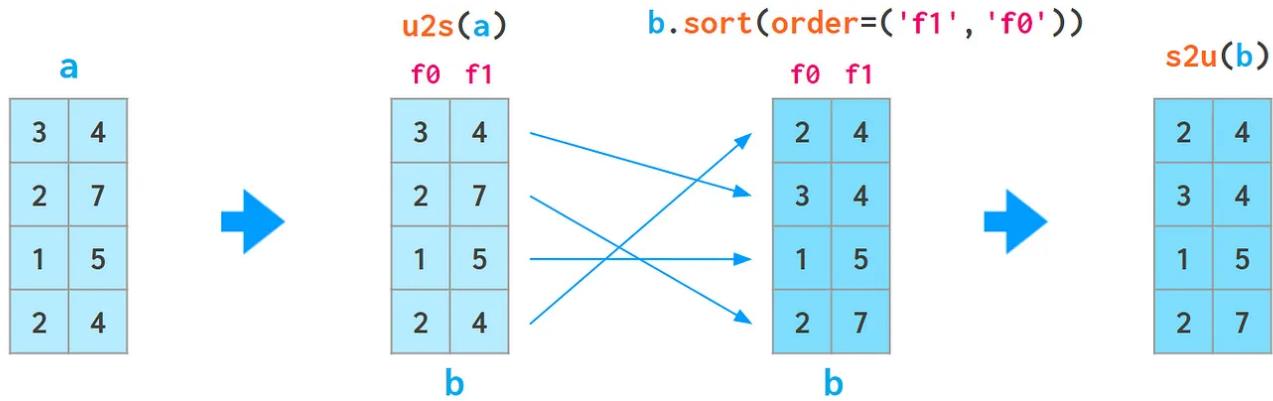


- `a[np.lexsort(np.rot90(a[:,[2,5]]))]` sorts by column 2 first and then (where the values in column 2 are equal) by column 5. An alternative form is `a[np.lexsort(np.flipud(a.T[[2,5]]))]`

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 `flplr`, that flips the 1D arrays) and `rot90` rotates the

matrix in the positive direction (i.e., counter-clockwise) by 90 degrees. Both flips and rot90 return views.

3. To use the `order` argument of `sort`, first convert the array to structured form, then sort it, and finally convert it back to the plain ('unstructured') form:



Both conversions are actually views, so they are fast and don't require any extra memory. But the functions `u2s` and `s2u` need to be imported first with the horrific

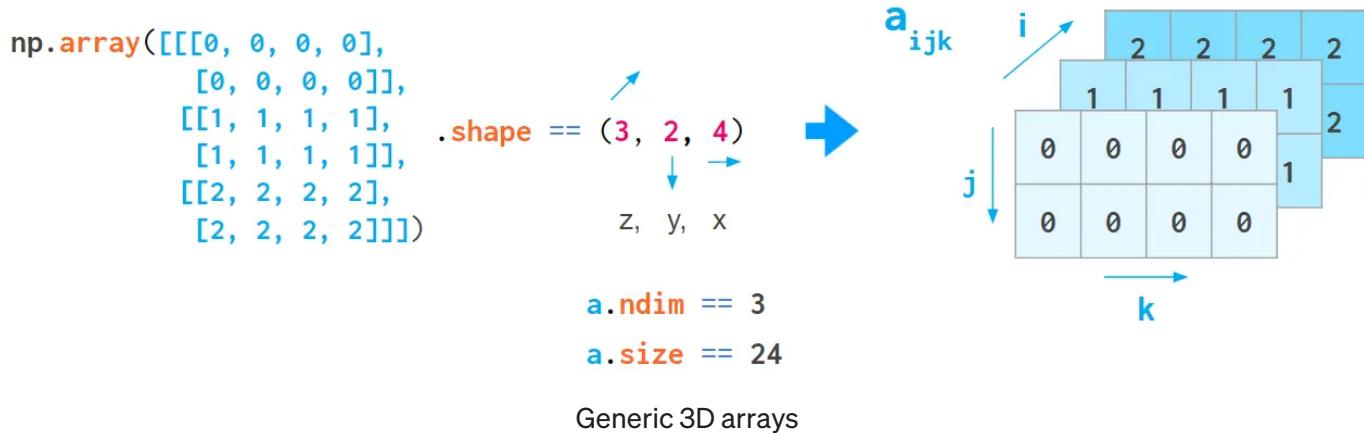
```
from numpy.lib.recfunctions import unstructured_to_structured as u2s,
structured_to_unstructured as s2u
```

4. It might be a better option to do it in Pandas since this particular operation is 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.

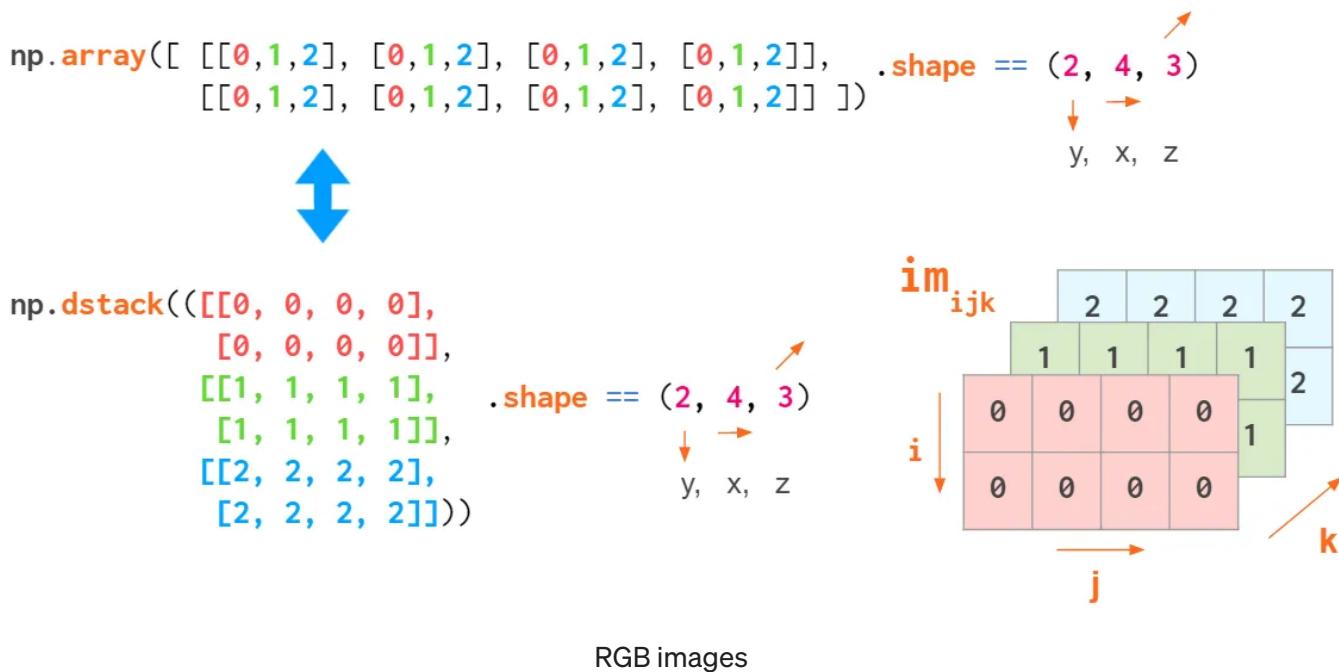
3. 3D and Above

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:



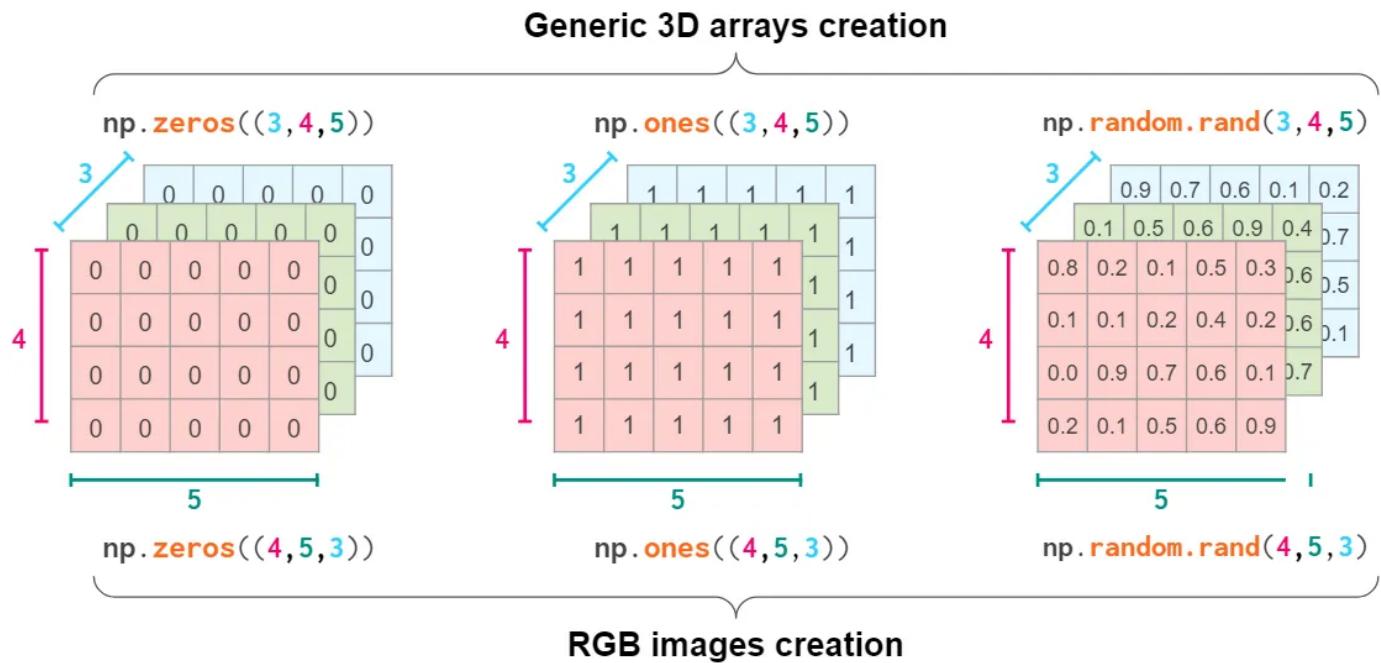
This index order is convenient, for example, for keeping a bunch of gray-scale images: $a[i]$ is a shortcut for referencing the i -th image.

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](#)):

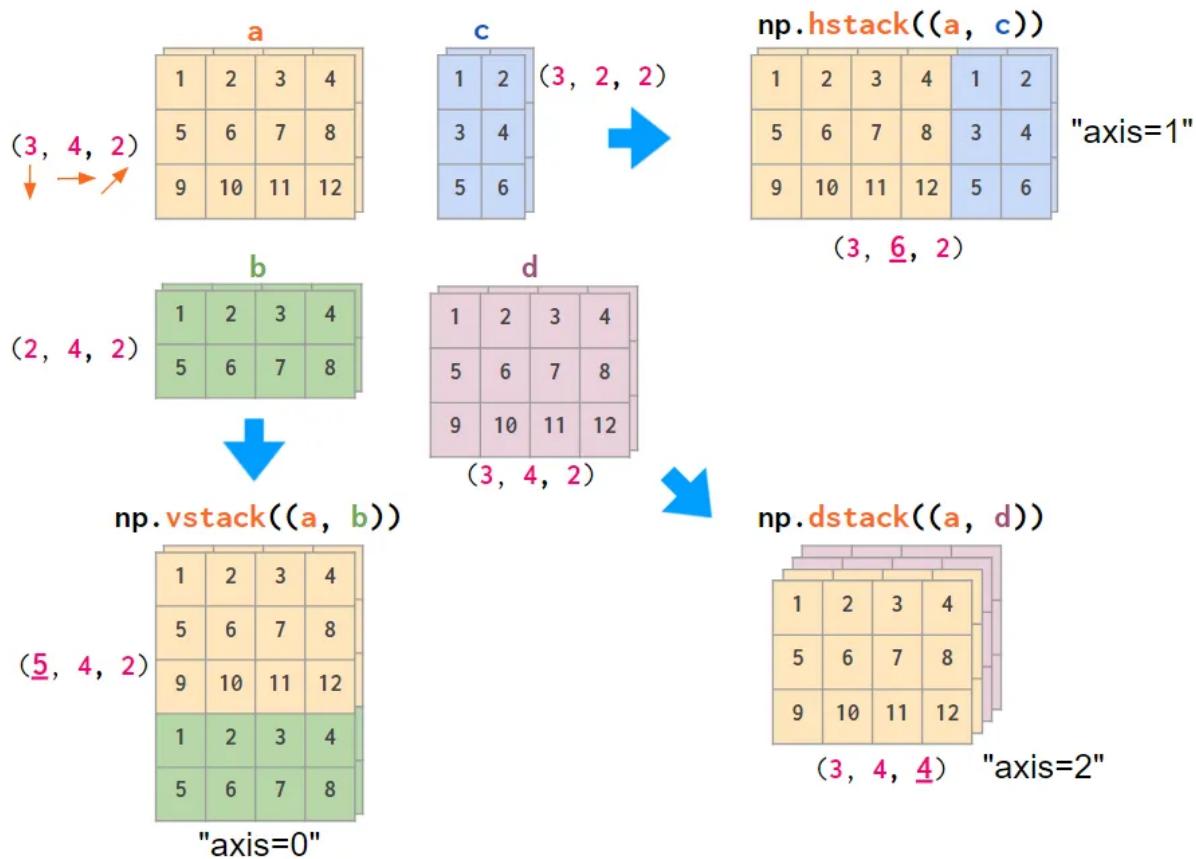


This way it is convenient to reference a specific pixel: `a[i,j]` gives an `(r,g,b)` tuple of the `(i,j)` pixel.

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

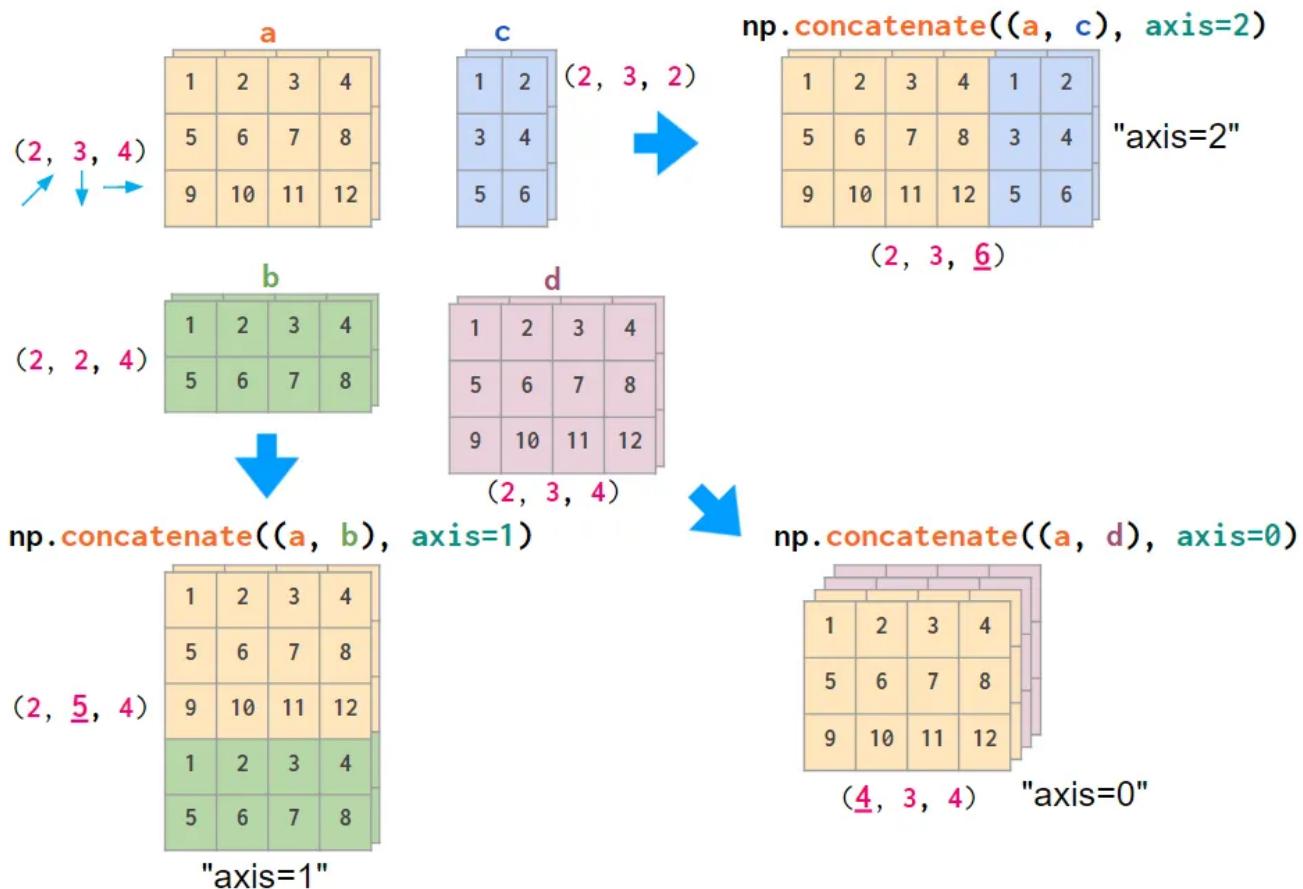


Obviously, NumPy functions like `hstack`, `vstack`, or `dstack` are not aware of those conventions. The index order hardcoded in them is `(y,x,z)`, the RGB images order:



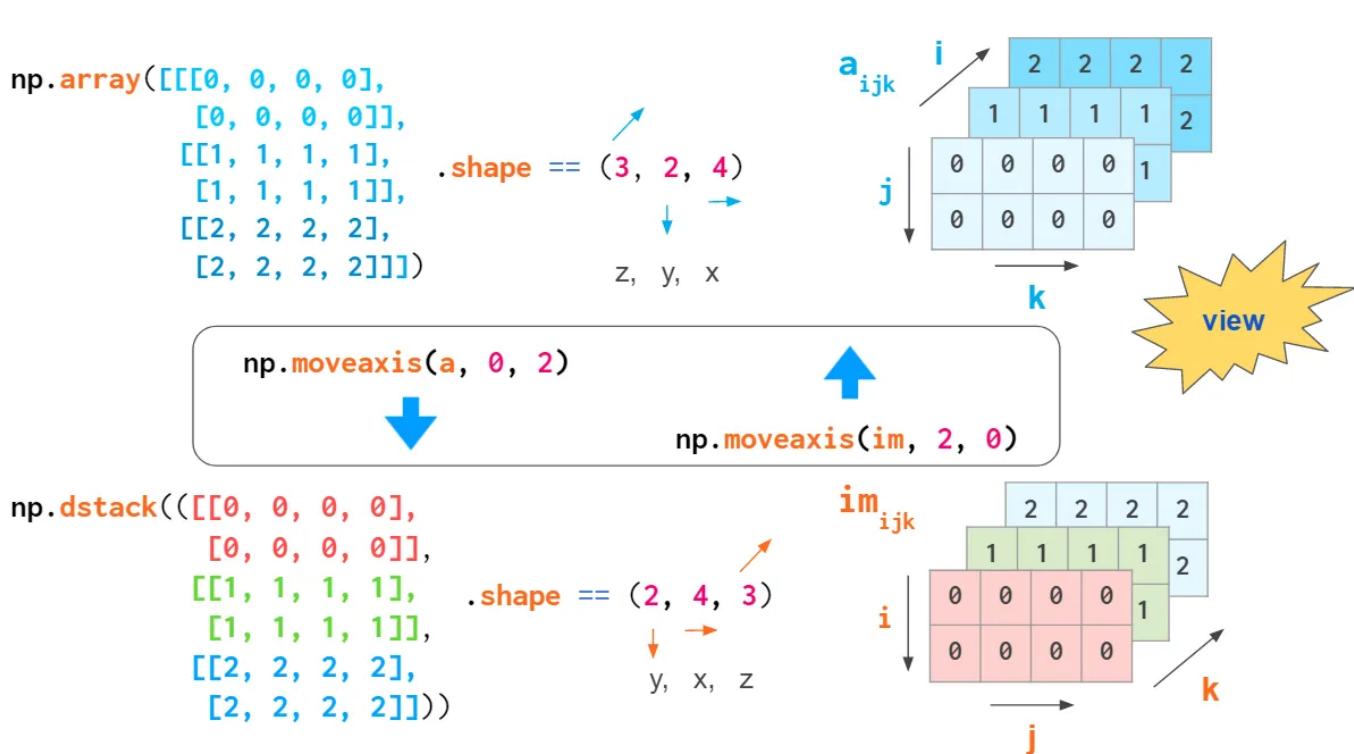
Stacking RGB images (only two colors here)

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:



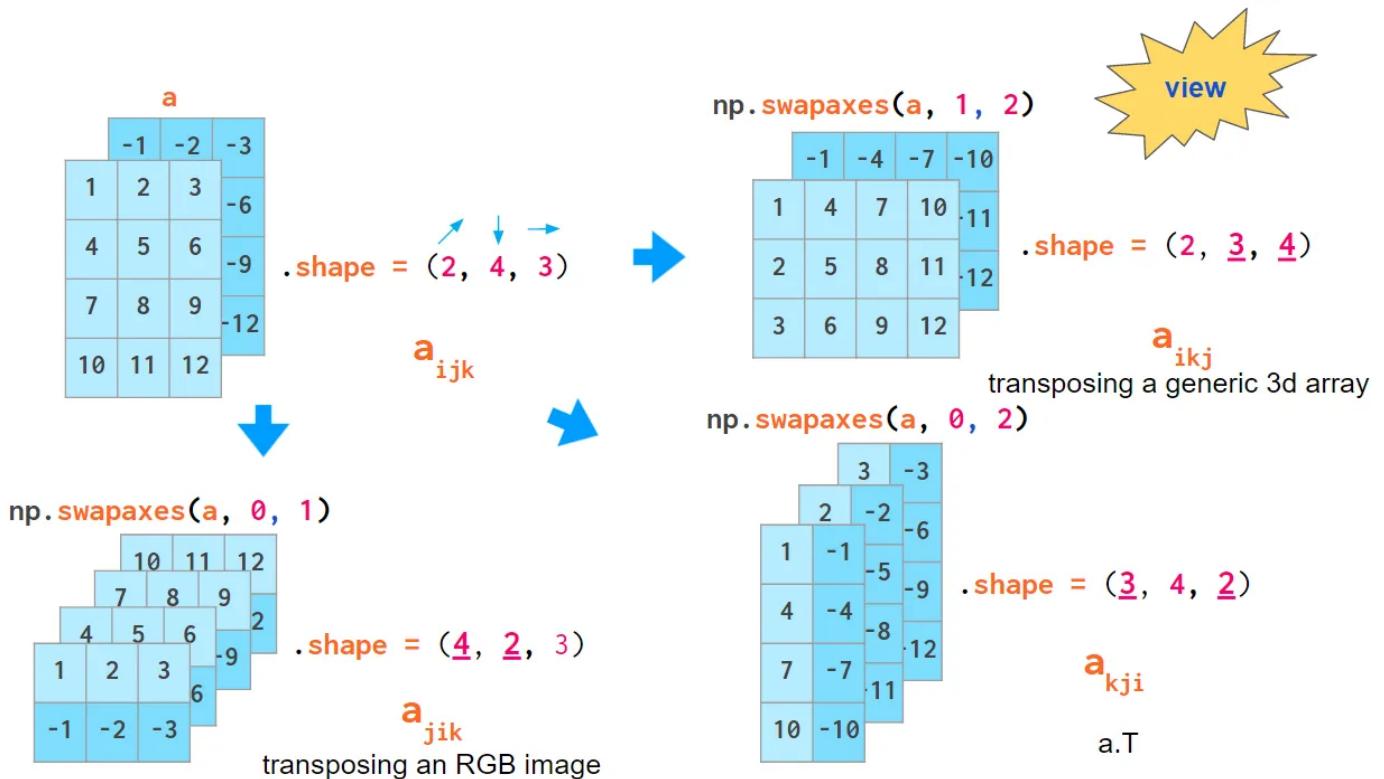
Stacking generic 3D arrays

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:



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:



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 brief article "[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 (see my new [Einsum Visualized](#) guide for details).

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](#), or [twitter](#)) if I have missed your favorite NumPy feature, and I'll try to fit it in!

2023 Update

I've addressed some of the mentioned issues (e.g. `find`, `irange`, `sort by column`, etc.) in an open-source library called ‘numpy-illustrated’:

The NumPy Illustrated Library

Softening the rough edges of NumPy

betterprogramming.pub

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](#)

License

All rights reserved (=you cannot distribute, alter, translate, etc. without author's permission).

[Programming](#)[Python](#)[Numpy](#)[Numpy Array](#)[Data Science](#)

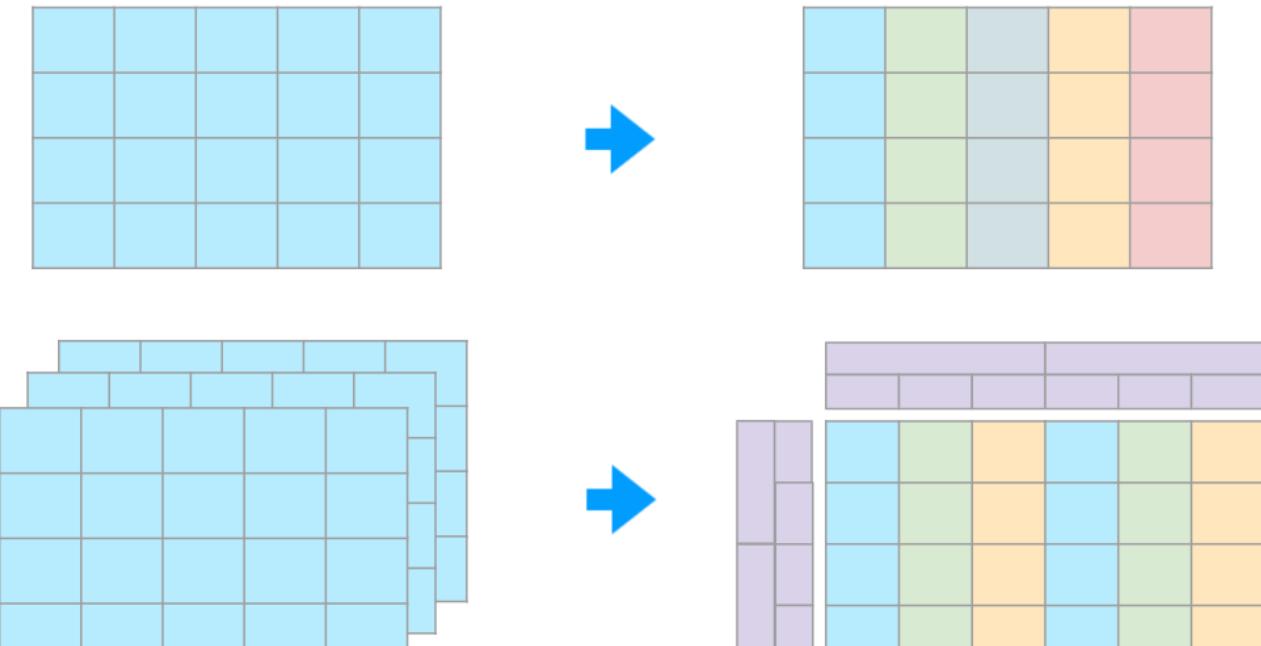
[Follow](#)

Written by Lev Maximov

1.2K Followers · Writer for Better Programming

Thrilled about programming

More from Lev Maximov and Better Programming



 Lev Maximov in Better Programming

Pandas Illustrated: The Definitive Visual Guide to Pandas

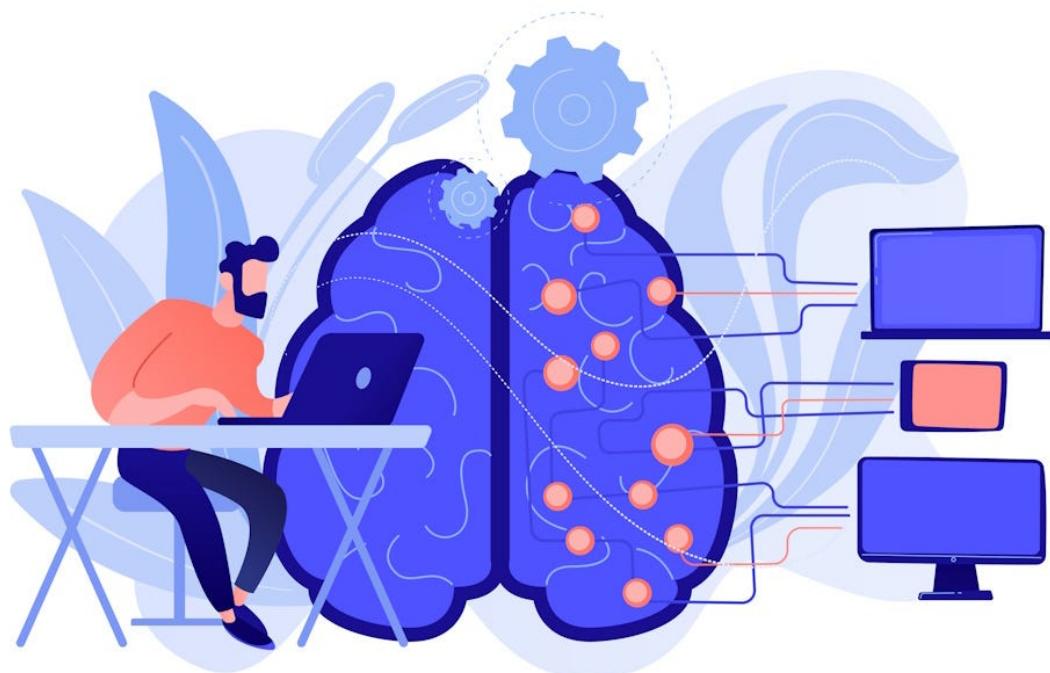
Is it a copy or a view? Should I merge or join? And what the heck is MultiIndex?

★ · 50 min read · Jan 25

 1.1K  14



...





Allen Helton in Better Programming

ChatGPT Changed How I Write Software

Generative AI has completely changed the way I approach software design. I don't think I could ever go back.

◆ · 8 min read · Jun 1

👏 1.7K 💬 20



...



Sami Maameri in Better Programming

Private LLMs on Your Local Machine and in the Cloud With LangChain, GPT4All, and Cerebrium

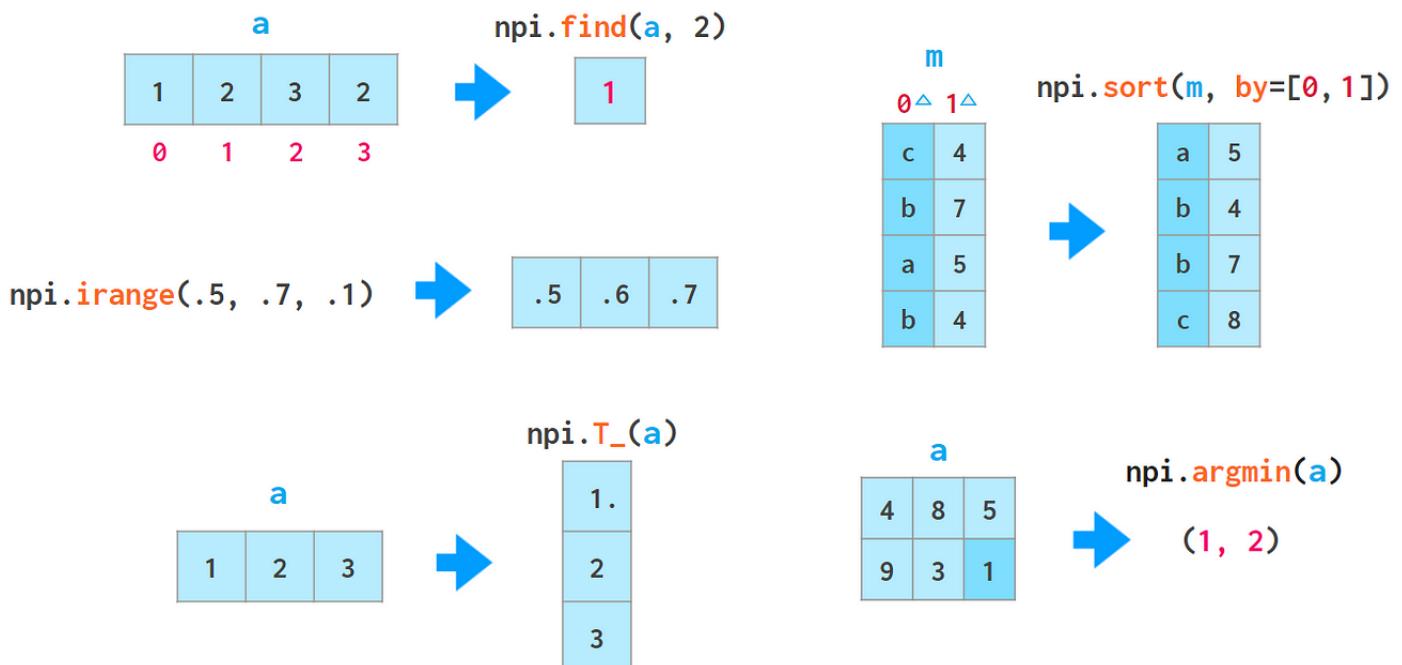
The idea of private LLMs resonates with us for sure. The appeal is that we can query and pass information to LLMs without our data or...

17 min read · Jun 15

👏 736 💬 6



...



Lev Maximov in Better Programming

The NumPy Illustrated Library

Softening the rough edges of NumPy

★ · 5 min read · May 12

102



...

See all from Lev Maximov

See all from Better Programming

Recommended from Medium



Youssef Hosni in Level Up Coding

How To Eliminate Loops From Your Python Code

Although using loops when writing Python code isn't necessarily a bad design pattern, using extraneous loops can be inefficient and costly...

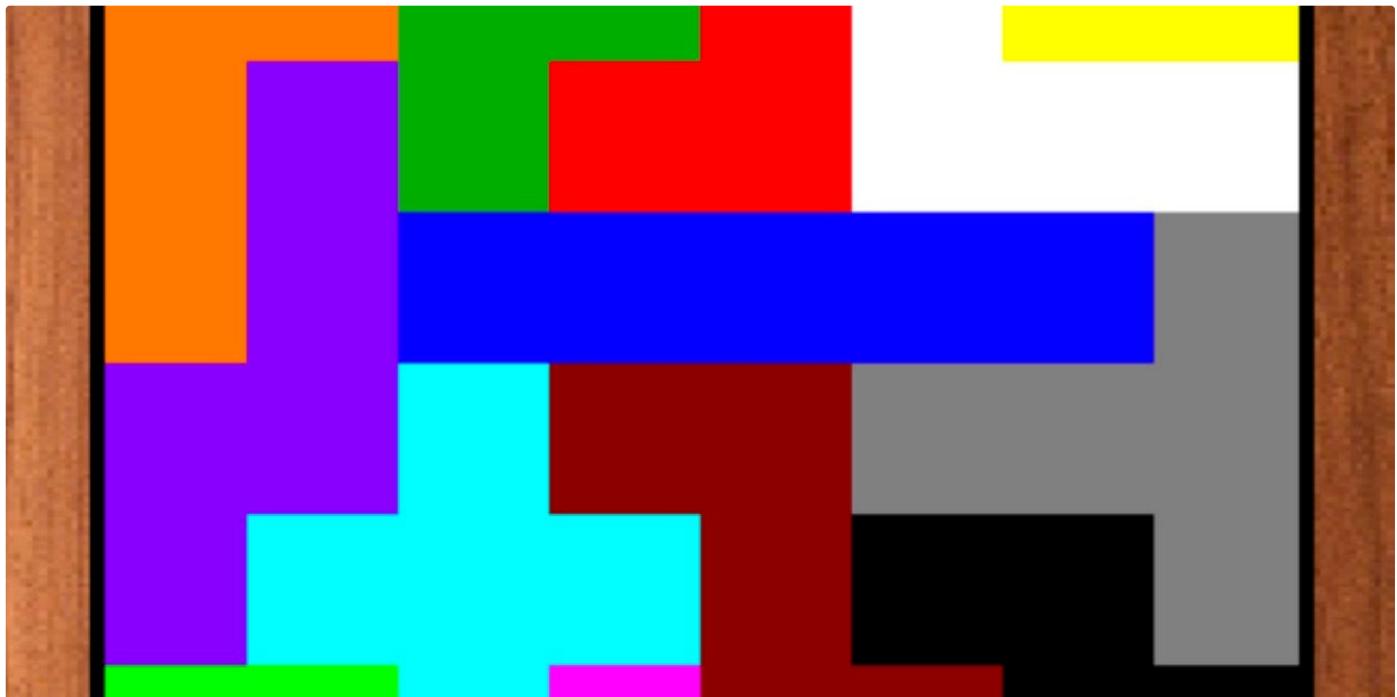
◆ · 8 min read · Oct 19, 2022

👏 780

💬 4



...



Aydin Schwartz in Better Programming

Automating Puzzle Solving With Python

Can't quite figure out that puzzle? Have you tried brute forcing every possible combination?

9 min read · Jan 23



187



1



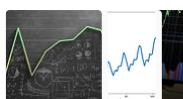
...

Lists



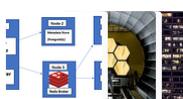
Coding & Development

11 stories · 25 saves



Predictive Modeling w/ Python

18 stories · 63 saves



New_Reading_List

173 stories · 8 saves



General Coding Knowledge

20 stories · 41 saves



 Tomer Gabay in Towards Data Science

5 Python Tricks That Distinguish Senior Developers From Juniors

Illustrated through differences in approaches to Advent of Code puzzles

◆ · 6 min read · Jan 16

 765

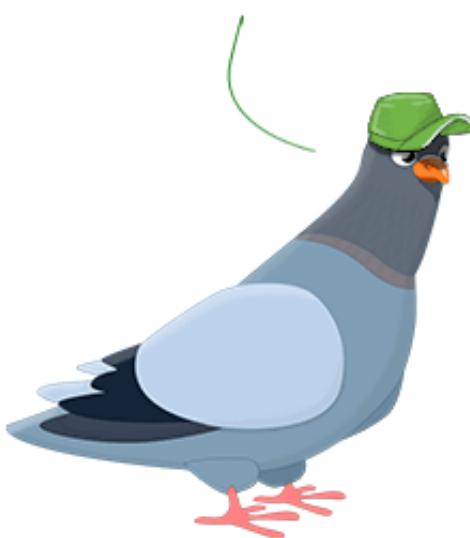
 19



...

what are other
words for
recurrence?

reoccurrence, reappearance,
frequency, recrudescence,
repeat, regularity, reiteration



 Felipe Morelli

Solving recurrence relations using the general method

This is a SymPy application to understand the general method for solving recurrence relations.

6 min read · Jan 28

 14

...

 Satnam Singh in Python in Plain English

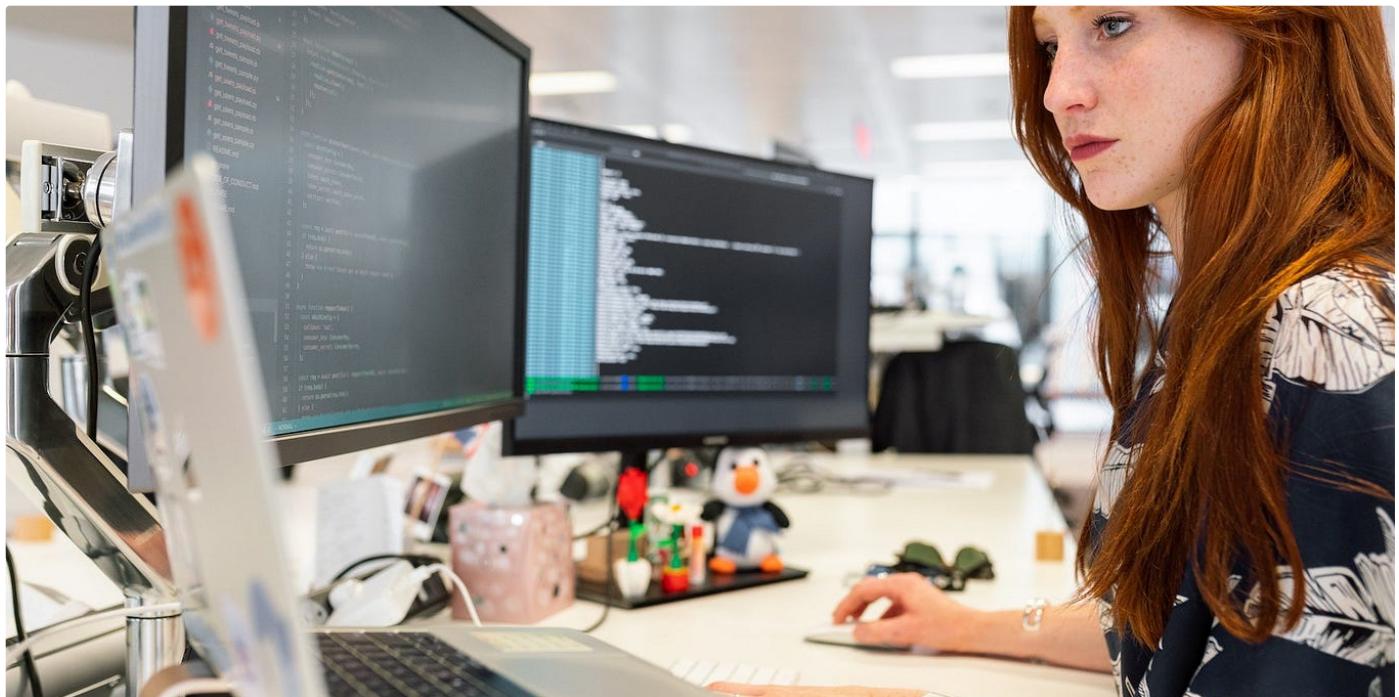
Advanced Python Programming: Writing Efficient Code with Cython and Numba

Python is a popular programming language for its simplicity, ease of use, and flexibility. However, when it comes to performance, Python...

6 min read · Feb 28

 46

...



The Coding Diaries in The Coding Diaries

Why Experienced Programmers Fail Coding Interviews

A friend of mine recently joined a FAANG company as an engineering manager, and found themselves in the position of recruiting for...

◆ · 5 min read · Nov 1, 2022

👏 4.5K

💬 102



...

See more recommendations