Medium Q Search Write Sign In
Sign In Get started
Open in app
Sign up. Sign in. Write. To read this story, log in/sign up, or use the Medium app. Read in app or sign up to read more.

NumPy Illustrated: The Visual Guide to NumPy

Brush up your NumPy in minutes from scratch

Lev Maximov Author
Nov 16, 2020 · 8 min read

Image credit: *ArtStation*

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 change on GPU.

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*, as a starting point, significantly expanded its coverage, and extended a pair of nuances.

Numpy Array vs. Python List

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

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

```
lst = [1, 2, 3, 4, 5]
lst * 2 # [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

arr = np.array([1, 2, 3, 4, 5])
arr * 2 # [2, 4, 6, 8, 10]
```

```
lst = [1, 2, 3, 4, 5]
lst = [x * 2 for x in lst]
lst # [2, 4, 6, 8, 10]

arr = np.array([1, 2, 3, 4, 5])
arr = arr * 2
arr # [2, 4, 6, 8, 10]
```

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 store but 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), 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 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:

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

Be sure to feed it a homogeneous list, otherwise you'll end up with `dtype='object'`, which nullifies the speed and only leaves the convenience of NumPy.

NumPy arrays cannot grow the way a Python list does. Its 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`:

```
b = np.zeros(5, int)
```

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

```
c = np.zeros_like(a)
```

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

```
d = np.full(3, 5.0)
e = np.full_like(a, 7)
f = np.ones(3, dtype=int)
g = np.ones_like(a)
h = np.eye(3, dtype=int)
i = np.full(3,fdtype, 7)
```

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

```
j = np.arange(start, stop, step)
k = np.linspace(0, 100, 5)
```

If you need a similar-looking array of floats, like `[0.0, 0.1, 0.2, ...]`, you can change the type of the `arange` output: `np.arange(0, 10, dtype=float)`, but there's a better way. The `linspace` function is type-sensitive. If you feed it no arguments, it will generate ten, and if you feed floats (e.g., `linspace(0.0, 9)`) it will generate floats.

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

```
l = np.arange(0.0, 0.7, 0.1)
m = np.arange(0.0, 0.8, 0.1)
n = np.arange(0.0, 0.9, 0.1)
o = np.arange(0.0, 1.0, 0.1)
```

Some wrong results in numerical operations (exaggerated)

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

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

```
rng = np.random.default_rng()
p = rng.integers(0, 10, 5)
q = rng.integers(0, 10, 5, endpoint=True)
r = rng.uniform(0.0, 10.0, 5)
s = rng.uniform(0.0, 10.0, 5, True)
t = rng.normal(0.0, 1.0, 5)
u = rng.normal(0.0, 1.0, 5, True)
```

New way random numbers generation

Vector indexing

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

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

Figure 5. Array creation functions.

- `np.arange` accepts all the compound numbers to be a compound array. For example, if you work with measurements, you need to divide the default `int` argument value by 5/9: `np.arange(0,100,5/9, dtype=int)`
- each `ndarray` holds no assumptions about the numbers to be compared and treat a zero or plus a reasonable `atol` value (absolute tolerance, the default `np.isclose` `atol` value of 1e-8 is good enough for numbers with a magnitude of 1): `np.isclose(1e-8,0)` → `array(True, dtype=bool)`

Aside from that, `np.isclose` has some minor issues in a formula for absolute and relative tolerance, for example, for certain `a`, `np.isclose(a, a)` → `array(False, dtype=bool)`. These issues were resolved in the Jeofred function `math.isclose` (which was introduced later). To learn more on that, take a look at the excellent *Floating point guide*[7] and the corresponding Stackoverflow[8] 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.

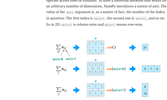
Matrix initialization syntax is similar to that of vectors:

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:

2D array random number generators

The ubiquitous double parentheses found their way to the interface of the new style (see details in the 1D section above) numbers generator routine so that as of today only `np.random.rand` has binary (other prerelease ones stringency and convenience of passing shapes as a



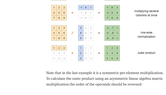
New-style random number generators

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 perform the operation along rows. 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 to operate. The first index is `axis=0`, the second one is `axis=1`, and so on. So in 2D `axis=0` is rows-wise and `axis=1` means row-wise.

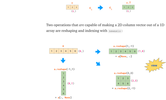


Generally speaking, the first dimension (`axis=0`) is responsible for indexing the rows, so `.argmax(axis=0)` should be read like "for given columns now out of all its rows" rather than just "column-wise". The 2D array representation is just a convenient way to look at the data, the memory allocated, instead of the remaining one-you-would-actually-think about, higher dimensional cons this is more natural, though, it'd be a burden to reconstruct 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 just number of the index you must 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 same without 'axis': `np.einsum('ij->j', a)` is summing columns wise (`'ij->i'` is summing rows in the image above) and `np.einsum('ij->ji', a)` is row-wise transpose or `np.transpose(a)`.

Matrix arithmetic

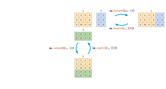
In addition to ordinary operators (like `+`,`/`,`**` and `%`) which work element-wise, there's a `@` operator that calculates a matrix product:



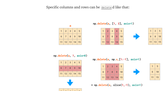
As a generalization of broadcasting from scalar that we've seen already in the first part, NumPy allows mixed operations between a vector and a matrix, and even between two vectors:

Row vectors and column vectors

In sum 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 array wherever possible (e.g., `.sum()`) — the 1D column of a 2D array `.[:,0]` 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. See surprisingly `newaxis` is not one of them:

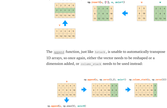


Two operations that are capable of making a 2D column vector out of a 1D array are reshaping and indexing with `.newaxis`:



Note that the 1 argument tells `.reshape` to calculate rest of the dimension sizes automatically and `-1` is the square brackets errors 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: 2D arrays, 2D row vectors and 2D column vectors. Here's a diagram of explicit conversions between them:



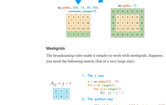
Matrix is always a row, vector is always a column, quasi-3 column-wise is possible

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 singular, is a row vector, so `a[:,0,np.newaxis,0]` is a row vector, but it's a matter of taste — I prefer to use the shortest form of the corresponding idiom, so I'd use `a[:,0]`. In contrast, a "normal" 2D vector is the clear and `np.newaxis` is per it hack. Both functions are so-so.

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 dimension-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:

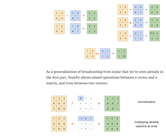
Matrix replication can be done in two ways: `.tile` acts like copy-pasting and `.repeat` like enlarged printing:

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

The inverse operation is `.insert`:

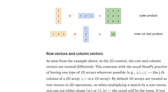
But actually, there's a better way to do it in NumPy. There's no need to specify an axis, so once again, either the vector needs to be enlarged or a dimension added, or `.broadcast_to` needs to be used instead:

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):



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

Without the `sparse=True` argument, `meshgrid` will change the order of the arguments (`'ij'` by default vs `'xy'` in the case of `sparse=True`), useful for visualizing 3D plots (see the example from the docs).

Matrix statistics

Just like `.sum`, all the other stats functions (`.prod()`, `.mean()`, `.var()`, `.std()`, `.cumsum()`, `.cumprod()`) accept the `axis` parameter and act accordingly:

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

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 an unhelpful for the 2D sorting:

It is just not what you would usually want from sorting a matrix or a spreadsheet: `axis=0` is in very a replacement for the `key` argument. But luckily, NumPy has sort-column 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 sort sort does not mess up the results of the previous one:

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

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 reversed (i.e., from bottom to top) so its usage is a bit convoluted, e.g.:

```
a = a[np.lexsort(np.flip(a,axis=1).T)]
```

`a = a[np.lexsort(np.flip(a,axis=1).T)]` sorts by column 2 first and then (where the values in column 2 are equal) by column 3. An alternative form `a = a[np.lexsort(np.flip(a,axis=(0,1)).T)]`.

Here `flip()` flips the matrix in the up-down direction (to be precise, in the `axis` direction, which is 0 by default), `.T` reflects them down-to-up, `lexsort` sorts by the last row, and `a[...]` applies the sort. `argsort` (but flags the 1D array) and `lexsort` converts the matrix in a column-wise direction (i.e., counter clockwise) by 90 degrees. Both return views.

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.

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.

```
df = pd.DataFrame(a, columns=['c1','c2','c3'])
df = df.sort_values(['c2','c3'], ascending=[1,0])
a = df.to_numpy(dtype=a.dtype).T
```

`a = np.lexsort(np.array([a[1],a[2]]))` sorts by column 2 first and then (where the values in column 2 are equal) by column 3. An alternative form `a = np.lexsort(np.flip(a,axis=(0,1)).T)`.

Here `flip()` flips the matrix in the up-down direction (to be precise, in the `axis` direction, which is 0 by default), `.T` reflects them down-to-up, `lexsort` sorts by the last row, and `a[...]` applies the sort. `argsort` (but flags the 1D array) and `lexsort` converts the matrix in a column-wise direction (i.e., counter clockwise) by 90 degrees. Both return views.

5. 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.

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

```
df = pd.DataFrame(a, columns=['c1','c2','c3'])
df = df.sort_values(['c2','c3'], ascending=[1,0])
a = df.to_numpy(dtype=a.dtype).T
```

3. 3D and Above

When you enter a 3D array by reshaping a 1D vector or converting a nested Python list, the meaning of the indices is `k,i,j`. The first index is the number of the plane, then the coordinates go in that plane:

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

But this index order is not universal. When working with RGB images, the `[i,j,k]` order is usually used. First are two pixel indices, and the last one is the color coordinate (RGB in Matplotlib, RGA in OpenCV):

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

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

Obviously, NumPy functions like `hstack`, `vstack`, `r_` and `c_` are not aware of these conventions. The index order hard-coded in them is `[i,j,k]`, the RGB image order:

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

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 the transposition, the resulting array is a different looking but still a plane 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 `axis` argument for `transpose` (or well-as-the-only `.T` operator mode) reverses the index order, which coincides with another 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 — `ndenumerate` (Numerate enumerate):

It seems the array-along the repeated indices. In this particular example `np.tensordot(a, b, axes=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 on GitHub.

Let me know (with `backtracing=1`, or twitter) 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 #14981, Revision parsers in indexing functions
6. NumPy Issue #5208, Fixes nonzero element
7. Pandas Dtype's Numpaxis
8. The Floating-Point Guide, Comparison
9. NumPy Issue #10593, nonzero indices is much inclear
10. 100 NumPy exercises on GitHub

Sign up for programming bytes
by Better Programming
A weekly newsletter sent every Friday with the best articles we published that week. Curated stories, advice, career opportunities, and more! See a sample.
Get this newsletter
Programming Python Numpy Data Science

Written by Dmitry Kisler
Lev Maximov
Software engineer and programmer
Follow
Better Programming
Advice, stories and ideas for programmers, literary play all organized on Medium's sixth floor. Hard-gloss subscribe.
Follow

More From Medium
Understanding Spark RDDs - Part 2
Understanding Spark RDDs - Part 2
Dmitry Kisler in Towards Data Science
Sort A and Maximum Reversed Array Has Minimum Index Equal to Minimum in Even Index
Sort A and Maximum Reversed Array Has Minimum Index Equal to Minimum in Even Index
Lev Maximov
What is rust and for when reading rust.
What is rust and for when reading rust.
Dmitry Kisler in Towards Data Science
The Ultimate Guide to Understanding Stack Overflow Tag Synonyms
The Ultimate Guide to Understanding Stack Overflow Tag Synonyms
Jon Udell in Towards Data Science
7 Data Science Job Post Mistakes Which Will Make Recruiters Fall Out
7 Data Science Job Post Mistakes Which Will Make Recruiters Fall Out
Dmitry Kisler in The Renaissance Friend
Tensorboard.dev Crashes Your Colab VM
Tensorboard.dev Crashes Your Colab VM
Lev Maximov
A top-alternative of Tensorboard for Keras models and deep learning in general. Fast-to-use-efficient and...
A top-alternative of Tensorboard for Keras models and deep learning in general. Fast-to-use-efficient and...
Dmitry Kisler in Towards Data Science
Music Recommendation System
Music Recommendation System
Jerome Blanchard

Learn more.
Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here you'll find the latest and best writing about everything from politics and culture to science and the future of work and life, with topics ranging from Self-Help to Technology to Design.
Make Medium yours.
Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore
Write a story on Medium.
It takes more than a sharp pen to be a great writer. Medium's editor will help you craft your story, from the first draft to the last edit. We'll help you find the right audience for your writing, and your feedback will be waiting for you. Start a story
About Write Help Legal
Medium