This is a sample of

Palette Programming: Invent Your Own Color Reduction Algorithm

by Pekka R. H. Väänänen

Find the full book at
https://paletteprogramming.com

22 Apr 2026

Appendix A
Image processing with NumPy

NumPy supplies a **multi-dimensional array** class that holds values of the same primitive data type, typically 64-bit floating point numbers. It's customary to import the library with an alias `np`:

```
import numpy as np
```

The name `np.ndarray` represents arrays in type annotations, but the class constructor is actually `np.array`. It can take a Python list of numbers as an argument. For example, two 3D vectors `w` and `c` can be created like this:

```
w = np.array([0.299, 0.518, 0.183])
c = np.array([0.9, 0.8, 1.0])
```

The standard addition `+`, subtraction `-`, division `/`, and multiplication `*` operators work by element. So for example the product `w*c` results in a new 3D vector `np.array([0.2691, 0.4144, 0.183])`. The rest of the library features are available as `np.array` methods or library functions. Let's say we want to convert an sRGB color in variable `c` to a single grayscale level with the formula

$$\text{grayscale}(R, G, B) = 0.299R + 0.518G + 0.183B.$$

The array in the variable `w` in the code sample above happens to already contain the per-channel weights, so we can scale and sum the color channels with `(w*c).sum()`, or `np.sum(w*c)` if you prefer a free function. To impress your friends, you could even go with a `np.dot(w,c)`, since the above operation is equivalent to a dot product.

The arrays can be one dimensional or multi dimensional. Their shapes are expressed as tuples of integers, and for example if we query the dimensions of an example vector above with `a.shape`, we get `(3,)` back. It's a one-element tuple that stands for a one-dimensional array, or a vector, of length 3. The `array` constructor interprets nested lists as multiple dimensions.

For example, to create a 2D array of shape (2,3), two rows and three columns, we would do this:

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

The constructor accepts an arrangement of nested lists only if all lists in a nesting level (dimension) are of the same length. That's why the following code wouldn't work.

```
# This throws a ValueError
x = np.array([[1, 2, 3],
              [4, 5]])   # <-- The two lists weren't the same length
```

You can interrogate an array object for its size and data type. This is useful when creating new arrays based on them, for example with the same shape but with a different type. Some attributes are listed below.

Attribute	Example value	Explanation
`x.ndim`	`2`	Number of dimensions.
`x.shape`	`(2, 3)`	Array dimensions as a tuple.
`x.size`	`6`	Total number of elements.
`x.dtype`	`dtype('int64')`	Data type.
`x.data`	`<memory at 0x7c73a55a1490>`	Backing data blob.

A.1 Aggregates

You already saw above how a sum over an array could be computed with `np.sum()`. "That's nothing new," you say, "in Python, lists can be summed with the built-in `sum()` function." That's true, but now you can *pick the dimension* to sum over. Let's say you have four colors stored in a 4×3 array like this:[13]

```
>>> colors = np.array([
...         [0.5, 0.7, 0.2],
...         [0.4, 0.8, 0.3],
...         [0.6, 0.7, 0.3],
...         [0.9, 0.7, 0.9]])
>>> colors.shape
(4, 3)
```

What if we want the average color? For that, we sum the numbers in each column and divide by the number of rows. The library calls dimensions *axes*, so we pass in `axis=0` to sum over the first axis:

```
>>> colors.sum(axis=0)
array([2.4, 2.9, 1.7])
>>> colors.sum(axis=0) / colors.shape[0]  # 'shape[0]' is 4
array([0.6  , 0.725, 0.425])
```

There's also a library function for this:

```
>>> colors.mean(axis=0)
array([0.6  , 0.725, 0.425])
```

These functions run faster than Python for loops and allow high-level operations to be expressed directly. For me, they are NumPy's main attraction. Other useful functions are `np.min()`, `np.max()`, and `np.std()` for minimum, maximum, and standard deviation, respectively.

A.2 How it works with images

Please fasten your seatbelts; it's time to move on to images, array indexing, pass-by-reference, and broadcasting rules. The good stuff.

An RGB image of $W \times H$ pixels is commonly loaded as an `(H, W, 3)` array. Note how the number of rows `H` comes first. Let's say we have

[13] The example is an interactive Python session. Lines starting with triple angle brackets ">>>" represent code written by the programmer and the lines without the prefix are the evaluated result printouts. The "`...`" prefix stands for multiline input.

it in variable `img`. Reading a single pixel is then a matter of indexing: `img[y, x]` returns an RGB triple as an `array` object of shape `(3,)`; this is shown in Figure 1.1. The indices don't need to be simple integers but can be lists of elements or slices, or even other arrays. See Table 1.2 for a selection to whet your appetite.

As you can see in Table 1.2, you can get creative with array indexing. You can use the ellipsis `...` to represent dimensions you don't care about; it makes code more "generic" but also shorter. So `img[:, :, 0]` and `img[..., 0]` both mean the same. You can even flip a dimension by indexing it with a slice that goes from the end of the range to its beginning with a step of -1. So for example `img[::-1]` reverses the order of image rows.

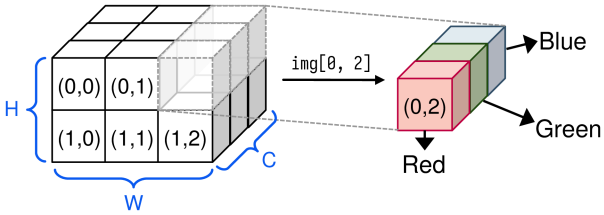

Figure 1.1. A $W \times H$ pixel image with $C = 3$ channels, represented as a multi-dimensional array. One pixel is addressed with `img[0, 2]`, resulting in a view to its color channels. The view is an array with the shape `(3,)` and it shares the underlying data with `img`.

Table 1.2. Indexing examples for an $W \times H$ pixel RGB image.

Expression	Interpretation	Result shape
`img[y, x]`	The RGB pixel at (x, y).	`(3,)`
`img[y, x, 0]`	Red intensity at (x, y).	`()`, a scalar
`img[:, :, 0]`	The whole red channel.	`(H, W)`
`img[:, :, 0:2]`	The image with only red and green channels.	`(H, W, 2)`
`img[:, :, [0,2]]`	The image with only red and blue channels. Uses a list, not a slice.	`(H, W, 2)`
`img[i, :, :]`	The image row `i`.	`(W, 3)`
`img[i, ...]`	Also, the image row `i`.	`(W, 3)`
`img[i]`	Again, the image row `i`.	`(W, 3)`
`img[-1]`	The bottom image row.	`(W, 3)`
`img[1::2]`	Odd image rows.	`(H//2, W, 3)`
`img[::-1]`	The image flipped vertically.	`(H, W, 3)`

A.3 Views and copies

One subtlety is how arrays are often *views* to other arrays. Array indexing in particular creates a view to the elements it addresses, most of the time. For example, in a code that reads a single pixel color from an image, we end up modifying the original image in the following snippet.

```
pixel = img[y, x] # returns a view to the array
pixel[1] = 0      # set the green channel at (y,x) in 'img'
```

This doesn't always happen. A copy is made if the result is a single primitive value such as `float`. More complex indexing expressions also cause a copy if the library can't figure out how to represent the result

internally as a view. For instance, using lists as indices is allowed, and below the result is a copy:

```
pixel2 = img[[y], x] # this time it's a copy
pixel2[1] = 0        # only modifies 'pixel2', not 'img'
```

Copies also happen when doing arithmetic on arrays:

```
img_normalized = img / 255  # creates a new array
```

Things get murkier with Python's own variable shadowing violating our expectations of data ownership. Knowing about NumPy's view behavior, perhaps you'd expect the following to modify the original `img` array.

```
>>> img.shape  # a 4x2 pixel RGB image
(2, 4, 3)
>>> a=img[0,0] # reference to 'img'
>>> b=img[1,0] # another reference
>>> a = b      # re-assings the Python variable only!
```

Unfortunately, the statement `a = b` only sets the name `a` to point to the same Python object as `b`. In this case `b` points to an `array` object, whose memory is shared with `img`. The situation is illustrated in Figure 1.2. This is how the language has been designed; the assignment operator itself can't be overloaded, unlike for example C++, where every equals sign is an invitation to adventure.

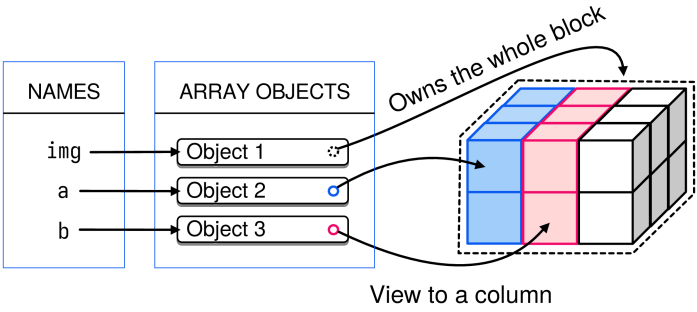

Figure 1.2. Relationships between variable names, NumPy's array objects, and the memory block allocated for `img`. The assignment `a = b` would only make the names `a` and `b` point to the same "Object 3".

So how do we actually assign the contents of `b` to the destination viewed by `a`? Python does allow custom behavior on *slice assignment*, so we can do this:

```
a[:] = b  # replace contents of 'a' with 'b'
```

The way it works internally is that the interpreter calls `np.array`'s overloaded `__setitem__` method that has a special case for slice arguments. It then modifies the array data in place. In the above, the colon ":" is a shorthand for `slice(None)`, so the example is equivalent to `a.__setitem__(slice(None), b)`.

You can check if arrays share the same data by inspecting their `.base` attribute:

```
>>> a.base is img
True
```

Finally, you can call `.copy()` if you want to make your intent clear or don't want to keep the original array around anymore:

```
>>> copied=a.copy()
>>> copied.base is img
False
```

A.4 Array data types and casting

How to compare two images? An easy way is to compute the **mean squared error** (MSE) between them, which involves taking the difference in each channel of every pixel, squaring the difference, and finally calculating their average. Of course, the images must be the same size for this to work.

Let's simulate this situation by creating two 8-bit 2×2 arrays to act as two images loaded from disk. The `np.random` module exposes a random generator factory function we use for the preparations. We can request a specific datatype from `rng.integers()` via its `dtype` argument, and we ask for an 8-bit unsigned integer, just like a PNG would be stored as.

```
>>> rng=np.random.default_rng()
>>> a = rng.integers(256, size=(2,2,3), dtype=np.uint8)
>>> b = rng.integers(256, size=(2,2,3), dtype=np.uint8)
```

Alright. Now we can square the differences `a - b` with the power operator `**` that works per-element, and feed the results to `np.mean()` to compute the average:

```
>>> np.mean((a - b)**2)
np.float64(101.0)
```

That looks like a reasonable number. Problem solved? You might feel your Numerical Issue Sense tingling and that's entirely appropriate, because a silent underflow made an appearance. See the first pixels of both images and their difference:

```
>>> a[0,0]
array([ 21, 175, 252], dtype=uint8)
>>> b[0,0]
array([245, 116, 198], dtype=uint8)
>>> a[0,0] - b[0,0]
array([32, 59, 54], dtype=uint8)
```

Looking at the first channel, $21 - 245$ is definitely not 32! By casting the operands to float first, we see the right numbers:

```
>>> a[0,0].astype(float) - b[0,0].astype(float)
array([-224.,   59.,   54.])
```

In fact, NumPy uses **data type promotion** rules similar to the C programming language, so you can get away with just casting one of the operands like this: `a[0,0] - b[0,0].astype(float)`.

For a programmer working with low-level details, unfortunate mishaps in integer math are a constant irritation, but they actually never happen in regular Python thanks to its `int` type that is an arbitrary length number (a "bignum"). Without such comforts, the way to deal with the issue is to either religiously convert every image from integers ranging $[0..255]$ to floating point $[0, 1]$ via `img/255` (division does an implicit cast), or via the `astype()` method, like done above. With the latter fix we learn the correct MSE is much higher:

```
>>> np.mean((a.astype(float) - b.astype(float))**2)
np.float64(8463.666666666666)
```

A.5 Broadcasting

For loops over big arrays are slow in Python. That's why NumPy offers **broadcasting**, automatic value replication in small arrays to match the dimensions of a larger one. Looping is delegated to the library internals, and hopefully to fast native code.

For example, let's see how to adjust image color balance. We need some test data. Let's create a matrix with numbers 1, 2, 3, 4 that represent gray values.

```
>>> square = np.array([[1,2],[3,4]], dtype=float)
>>> square
array([[1., 2.],
       [3., 4.]])
```

Then repeat it along the channel dimension three times to arrive at a 2×2 RGB image.

```
>>> img = np.repeat(square.reshape(2, 2, 1), 3, axis=2)
>>> img.shape
(2, 2, 3)
```

Now `img` is a 3-channel grayscale image:

```
>>> img
array([[[1., 1., 1.],
        [2., 2., 2.]],

       [[3., 3., 3.],
        [4., 4., 4.]]])
```

Our refined taste tells us the image must be adjusted exactly with per-channel weights of $(0.9, 0.8, 0.7)$ that we save in array `a`.

```
>>> a = np.array([0.9, 0.8, 0.7])
```

Thanks to broadcasting, we can multiply each pixel per-channel with the weights like this:

```
>>> img * a
array([[[0.9, 0.8, 0.7],
        [1.8, 1.6, 1.4]],

       [[2.7, 2.4, 2.1],
        [3.6, 3.2, 2.8]]])
```

What happened here? From NumPy's point of view, we just tried multiplying together two arrays of incompatible shapes: `(2,2,3)` and `(3,)`. To make it work, the library first had to extend `a`'s dimensions from one to three to match the array `img`, resulting in a shape `(1,1,3)`. Note how the "missing" dimensions became ones and were added before the original. The reshaped array still has three values in it; it's as if the list `[0.9, 0.8, 0.7]` was turned into `[[[0.9, 0.8, 0.7]]]`.

Now the operation is between two arrays of size `(2,2,3)` and `(1,1,3)`. Again, the library tries its best to make them compatible. This time the three numbers in `a`, the per-channel weights, are copied to create a larger `(2,2,3)` array. Finally, a simple per-element multiplication becomes possible, and its result is the one above. All this happened inside an unassuming `img * a` expression. The product in the example is illustrated in Figure 1.3.

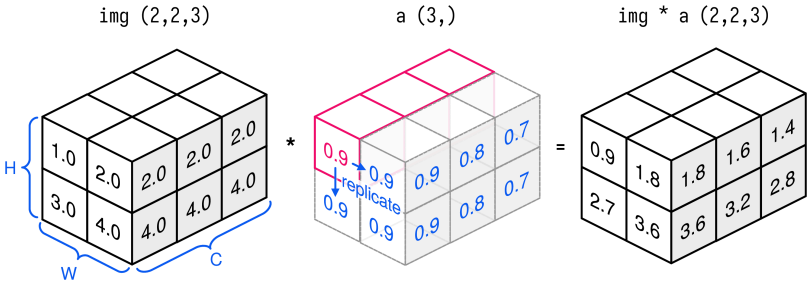

Figure 1.3. An example of broadcasting during an element-wise product. NumPy first extends the smaller array `a` to match the dimensions of the larger `img`, and only then multiplies the arrays per element. This way the programmer can delegate repetitive array operations to the library's fast internal routines instead of allocating temporary arrays or writing slow Python for loops.

Broadcasting has two main rules and both were applied here: missing dimensions behave as if they did have size one, and the smaller array's values are replicated over dimensions whose size is one. The values don't actually get copied, but it's helpful to think about array shapes that way. For more details I refer you to the library's own guide[14], but if you encounter an expression where the array sizes don't

[14] https://numpy.org/doc/stable/user/basics.broadcasting.html

seem to match yet it still works, then have no fear: you're witnessing broadcasting.

A.6 Image as a set of vectors

An image can be viewed as an unordered set of pixels. This move discards structural information such as edges, since now we can't tell which pixels are neighbors. But it makes the data convenient to work with.

Let me show what I'm talking about using the tiny image inset on the right as an example. We load the image with the Pillow library, discard its alpha channel with `convert("RGB")`, and convert it into an array `img`:

```
>>> import numpy as np
>>> from PIL import Image
>>> obj = Image.open("tiny.png").convert("RGB")
>>> obj
<PIL.Image.Image image mode=RGB size=30x19 at 0x7CDC579D1010>
>>> img = np.array(obj)/255
>>> img.shape
(19, 30, 3)
>>> img.dtype
dtype('float64')
```

How to represent a set of pixels in code? We could put a bunch of `array` objects in Python's `set` container, but a 2D array works better if we want to do arithmetic. An image can be interpreted as a 2D array via `img.reshape(-1, 3)` where `-1` in one dimension says we want its size to be automatically deduced. In this case the new shape is `(570, 3)`:

```
>>> X = img.reshape(-1, 3)
>>> X.shape
(570, 3)
>>> X[0]
array([0.59215686, 0.78039216, 0.95686275])
>>> img[0,0]
array([0.59215686, 0.78039216, 0.95686275])
```

In machine learning and data science, it's customary to call the 2D **data array** a capital bold $\boldsymbol{X}$, so that's what I used above. You can see a visual explanation of what's going in Figure 1.4.

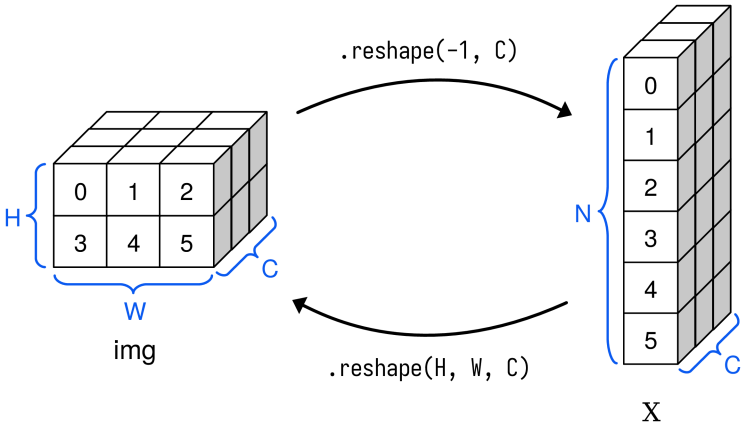

Figure 1.4. From an array `img` to a data array $\boldsymbol{X}$. An array can be "reshaped" to a new size as long as the number of elements stays the same. Passing `-1` as a size of a dimension calculates its size based on the other arguments.

What is a data array good for? At least indexing individual pixels becomes simpler since they can be identified by a single integer. As an example, say we wanted to artificially corrupt an image by setting 5% of its pixels to white. Having a 2D data array, we can generate a list of random indices and assign `1.0` to those rows in $\boldsymbol{X}$.

So first we need random numbers to act as indices. The number of rows in `X` is accessible as `X.shape[0]` (it's 570) and we calculate in `num` the rounded fraction used for random sampling:

```
>>> rng = np.random.default_rng()
>>> num = int(np.round(X.shape[0] * 0.05))
```

Then we call `rng.choice()` that takes in as its first argument the exclusive upper limit of generated numbers and in `size` how many we want. We also turn off "sampling with replacement" so that the same index can't appear twice.

```
>>> inds = rng.choice(X.shape[0], size=num, replace=False)
>>> inds.shape
```

```
(28,)
>>> inds
array([524, 516, 553, 135,  19, 419, 289, 6, 478, 561,
       348, 280,  15, 376, 488, 173, 241, 8,  14,  43,
       255, 383, 126,  51, 551,   7, 375, 325])
```

As mentioned earlier, arrays can be indexed with other arrays, so clearing the pixels is simple. Broadcasting will assign `1.0` to every channel on the selected rows.

```
>>> X[inds] = 1.0
```

What is not simple is going back from a floating point data array to an 8-bit RGB image, even though the code is short. First the $[0, 1]$ range data has to be scaled back to 8-bit range, and then rounded with clipping to integers. The `X_8bit` array is still `(N, 3)` shaped at this point, so it must be reshaped back to `(H, W, 3)` for Pillow's `fromarray()` to understand it.

```
X_8bit = (X * 255).round().clip(0, 255).astype(np.uint8)
new_obj = Image.fromarray(X_8bit.reshape(img.shape))
new_obj.save("tiny_corrupted.png")
```

On the left you see how the corrupted result looks like. Another success! That concludes this guide.

References

Like mentioned earlier, *NumPy quickstart* is a great introduction that's broader than what was covered here:

🔗 https://numpy.org/doc/stable/user/quickstart.html