This is a sample of

Palette Programming: A Modern Guide to Color Quantization
by Pekka R. H. Väänänen

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

05 Jan 2026

Chapter 2
Python image processing

I wish I could say this book's code examples were written in *executable pseudocode*, that mythical beast of notation that's both simple and precise. But it's just Python.

To make the code look simpler I had to make some tough choices. First, I ignore performance for clarity. The *algorithms* are solid but the implementations shown can be sloppy. Second, the sample programs are standalone scripts. That means we are not building a big application one step at a time, but a series of experiments that exhibit one or two interesting things. Third, I use NumPy, the standard numerical computing library. It just makes working with images, colors, and arrays so much easier.

NumPy is also a problem. While popular in data science, statistics, and engineering, most people programming in Python never need to touch it. I want this book to be approachable to anyone with intermediate-level programming skills and an interest in graphics. That's why this chapter is a guide to a grab bag of concepts needed to understand the code examples in the book. It won't make you an expert, but it should

give you some confidence for further self study. I can recommend the official "quickstart" guide[1] for a more complete picture.

And hey, we even design some palettes at the end!

2.1. Just enough NumPy

NumPy supplies a *multi-dimensional array* type 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
```

Type annotations use `np.ndarray` to represent arrays, but the constructor is actually `np.array` that takes 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 array objects have new methods too. Let's say we want to convert an sRGB color in variable `c` to a single grayscale level with the formula $0.299R + 0.518G + 0.183B$. The array in the variable `w` above happens to already contain the same numbers, so we can weight 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.

[1] https://numpy.org/doc/stable/user/quickstart.html

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]])
```

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.

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:[2]

```
>>> 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)
```

[2]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.

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.

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 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 illustrated in Figure 2.1. The indices don't need to be simple integers but can be lists of elements or slices, or even other arrays. See Table 2.2 for a selection to whet your appetite.

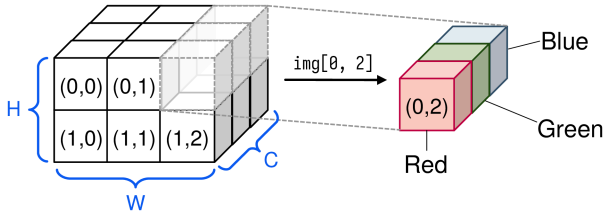

Figure 2.1. A $W \times H$ pixel image with $C = 3$ channels, represented as a multi-dimensional array. One pixel is addressed with `img[0,3]`.

As you can see in Table 2.2, 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, for example `img[::-1]` is interpreted as the slice from 0 to end of the list, with step -1, reversing the order of image rows.

Table 2.2. 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)`

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  # makes a copy
```

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 2.2. 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.

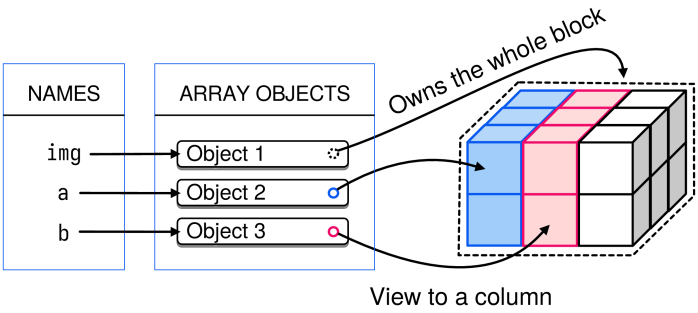

Figure 2.2. Relationships between variable names, NumPy's array objects, and the memory block allocated for `img`. The assignment `a = b` would only make the names `a` and `b` point to the same "Object 3".

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
```

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. But they are easy mistakes to make when working with NumPy. The way to deal with the issue is to either religiously convert every image from $[0..255]$ range integers to floating point $[0, 1]$ with `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)
```

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. The reshaped array `a` still has just three numbers in it.

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 2.3.

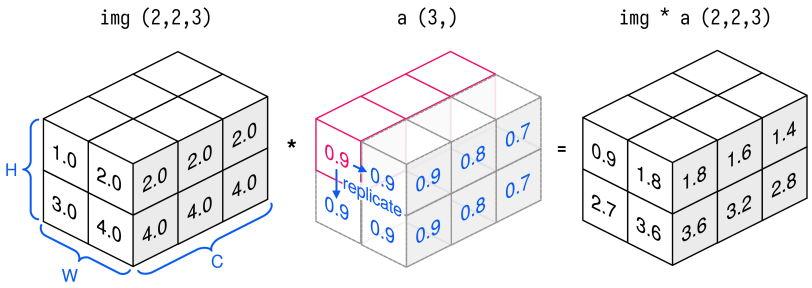

Figure 2.3. An example of broadcasting during an element-wise product. NumPy first extends the smaller array `a` to match the dimensions of the larger `img`, and only then multiplies the arrays per element. This way the programmer can delegate repetitive array operations to the library's fast internal 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[3], but if you encounter an expression where the array sizes don't seem to match yet it still works, then have no fear: you're witnessing broadcasting.

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`:

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

```
>>> 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 2.4.

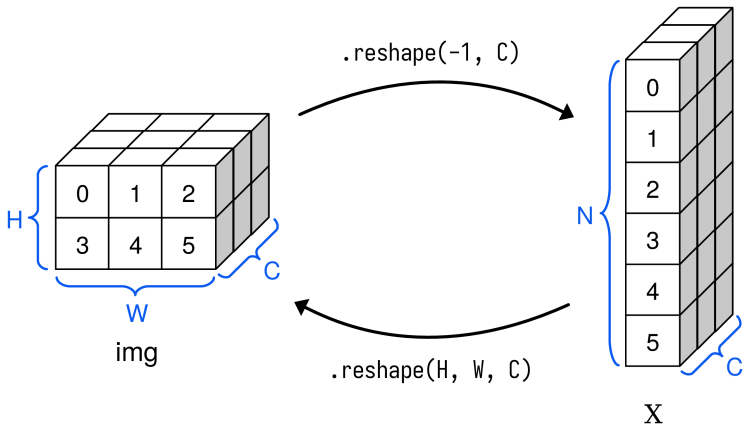

Figure 2.4. From an array `img` to a data array $\boldsymbol{X}$. An array can be "reshaped" to a new size as long as the number of elements stays the same. Passing `-1` as a size of a dimension calculates its size based on the other arguments.

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 simple 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! If this operation seemed cumbersome without a good reason, I hope you'll be convinced by the next section that shows how to implement this book's first algorithm for palette design.

2.2. Easy palettes with sklearn's k-means

In the earlier section we saw how an image could be converted to a set of pixels, represented as the data array $\boldsymbol{X}$. Now we finally put it into good use and feed that array to the *k-means* clustering algorithm.

Conveniently, our data happens to already be in a format expected by the scikit-learn machine learning library: a 2D array. The library interprets each row as an independent *data point*, which in our case means each pixel conceptually becoming a 3D point inside an RGB cube.

Now we'll write this book's first *color quantization* routine that takes in an arbitrary image, finds a palette, and maps each pixel of the original to a palette color. Even better, the results will be high quality even though

Original k-means++ Palette

Figure 2.5. Designing a 16-color palette for MONA LISA with scikit-learn's k-means clustering. The standard k-means++ initialization gives an excellent result and the only thing missing is dithering to hide color banding.

the code is short. You can see an example in Figure 2.5. As a downside, it doesn't scale to large images as is.

The scikit-learn library's module is called `sklearn`, and its `cluster` submodule exports the `KMeans` object (along with 25 other algorithms).

sklearn_kmeans.py begins

```
1 import numpy as np
2 from PIL import Image
3 from sklearn.cluster import KMeans
```

This time we'll wrap the code in a helper function that takes in an image and the color count K. The first line normalizes the image to $[0, 1]$ range, which also does an implicit floating point conversion. Like earlier, $\boldsymbol{X}$ is a 2D data array of shape $N \times C$, where C is the number of channels.

```
6 def run(img: Image.Image, K: int):
7     data = np.array(img) / 255.0
8     H, W, C = data.shape
9     X = data.reshape(-1, C)
10    N = X.shape[0]
```

As an example, a 128×128 RGB image has $N = 16384$ and $C = 3$. Then we cluster the data. The `KMeans` constructor takes in parameters of the process and returns an object. Note that $\boldsymbol{X}$ wasn't passed in yet.

```
12      kmeans = KMeans(
13          n_clusters=K,          # each cluster is a palette color
14          init="k-means++",      # standard k-means++ init
15          n_init=1,              # try initialization just once
16          max_iter=100,          # usually stops before this limit
17          tol=1e-4,              # stop when error goes this low
18          random_state=123,      # make it deterministic
19          algorithm="lloyd",     # use the standard variant
20      )
```

The most important arguments are `n_clusters` and `init`: the number of clusters and the type of the so-called initialization step. We assumed a fixed palette size so we can just ask for one cluster per palette color. The chosen *k-means++* initialization method is a reliable choice but doesn't scale to large point counts. More about that later.

The `sklearn` module is designed around predictor objects that can be *fit* to a dataset and then used to make *predictions* with new, unseen data. In the k-means' case, fitting is the clustering part and prediction is assigning each data point its closest cluster. We'll call the `fit_predict()` convenience API that does both steps at once and returns per-point cluster indices. Then we'll read the palette colors from the object's `cluster_centers_` variable.

```
22      indices = kmeans.fit_predict(X)    # shape (N,)
23      centers = kmeans.cluster_centers_  # shape (K, C)
```

The centers returned by k-means are still in the $[0, 1]$ range. To turn them into 8-bit RGB colors, we chain integer rounding, clamping or "clipping", and type cast operations together like in the previous section:

```
24      palette = np.round(centers*255).clip(0,255).astype(np.uint8)
```

We could already return the `palette` and the `indices` arrays as is from the function, but let's reconstruct the image so it can be saved or shown. The task is simple: for each pixel, replace the computed cluster index with its respective RGB color in the palette. This would involve a for loop over

every pixel, something we try to avoid due to Python's slowness, so we have to again reach to NumPy for help.

Luckily, we can index arrays with other arrays, so it's enough to plug in `indices` as the first dimension's index. This works even though `indices` is much longer than `palette`, because the same row can be picked multiple times.

```
26      result = palette[indices, :].reshape(H, W, C)
27      return Image.fromarray(result)
```

The resulting shape of `palette[indices, :]` is $N \times C$, so we have to reshape it back to a rectangular array with C channels in order to construct the output image. It's the operation shown in Figure 2.4 but done in reverse. And that's it for the `run()` helper function. Our color quantizer is now ready to be used:

```
33      input_path = "images/spectrum_cone_128.png"
34      img_input = Image.open(input_path)
35      img_result = run(img_input, K)
```

↳ sklearn_kmeans.py ends

See the result in Figure 2.5. It's already pretty good! If we didn't have any higher palette programming ambitions, we could already get off here.

Why do we need anything else?

What's missing? To smoothen out color gradients, we'd need some dithering (Section 11) but that's its own problem and can be done with any palette. It's not a reason to ditch k-means. If the palette is designed in a different color space than sRGB, the color reproduction improves further almost for free (Section 10). I would call this combo already competitive with the state of the art.

Execution time is a more pressing question. The scikit-learn docs say that in practice, k-means is "one of the fastest clustering algorithms available". The thing is, it's also kind of clueless. A clustering algorithm is supposed to find clusters in the data, but k-means can't do anything unless we give it a first guess of cluster centers' locations. This guess is given by an *initialization* routine. The rest of the algorithm then iteratively refines the guess until happy. This is discussed in detail in Section 5.

A reliable choice for initialization is the now-standard k-means++ algorithm that works well both in practice and in theory. The clustering it

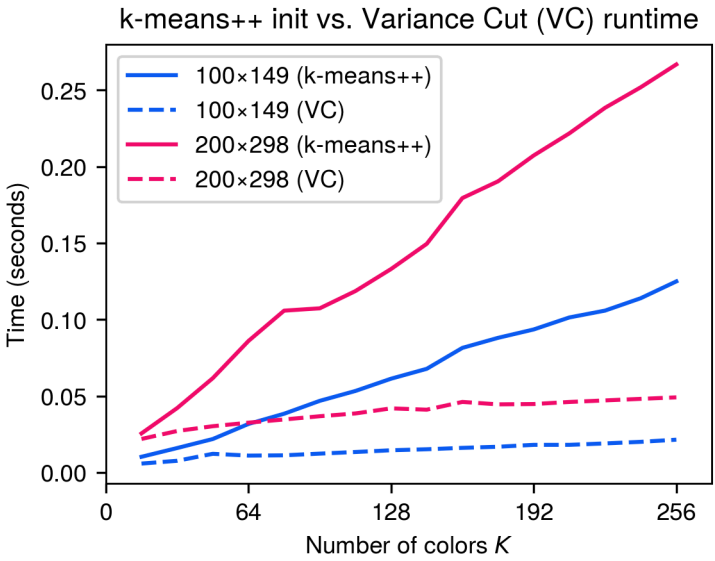

Figure 2.6. Execution time of `sklearn.cluster.kmeans_plusplus` initialization (the default "greedy" variant) compared with the "Variance Cut" top-down clustering algorithm (described in Section 6). The two colors represent benchmark results on two sizes of the MONA LISA test image. Execution times of the two techniques are very close at $K = 16$ but quickly separate when the number of colors increases. Both are written in Python using NumPy and don't include k-means iterations or pixel mapping. Benchmarked on a *Ryzen 7 5700X* processor.

finds has a statistical guarantee of its quality (see references at the end of this chapter) and its execution time scales linearly[4] with point count N. This is what we chose with the `init="k-means++"` constructor argument to `KMeans`.

But if you're planning to use that class, you'll have to accept that just the initialization can take over a quarter of a second for a 200×298 pixel *Mona Lisa*. See Figure 2.6 for a runtime comparison against another technique discussed in this book. Wouldn't you like to know how the faster method works?

[4]k-means++ initialization does an $O(KN)$ pass for each center, so the full time complexity is $O(K^2N)$. Point dimensionality d was assumed constant.

To be honest, there *are* ways to increase k-means' processing speed without leaving the comforts of your favorite machine learning library:

- First of all, you can process only the *unique* image colors, which of course speeds up clustering. But this benefits the alternatives too, and is discussed in Section 9.
- You can change from `KMeans` to the `MiniBatchKMeans` class that scales more gracefully to large point counts.
- To speed up initialization, swap `init="k-means++"` to `init="random"` and set `n_init=10` to regress back to traditional random guessing.
- You can call `kmeans_plusplus()` directly with `n_local_trials=1` and pass its output to `KMeans` as the first guess. This uses a "non-greedy" variant that runs at least two times faster.

But now you're trading image quality for speed. If that's what you're after, why not go all the way? Why not opt for algorithms that give visually identical results but are cleverly programmed to run fast? Code that applies latest research to old problems. Sounds good? Let's do some *palette programming*.

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

The k-means chapter in scikit-learn's user guide links to good further references:

https://scikit-learn.org/1.8/modules/clustering.html#k-means

The original k-means++ paper (Arthur and Vassilvitskii 2007) is also a good read and proves that the expected "cost" of its initial clustering, the sum of squared distances of each point to its cluster center, is at most $8(\ln K + 2)$ times the theoretically optimal solution. That factor was later tightened to $5(\ln K + 2)$ in (Makarychev et al. 2020). The default k-means++ variant that scikit-learn uses is actually a "greedy" one mentioned at the end of the original paper, but its optimality bounds were proven only in (Grunau et al. 2023).