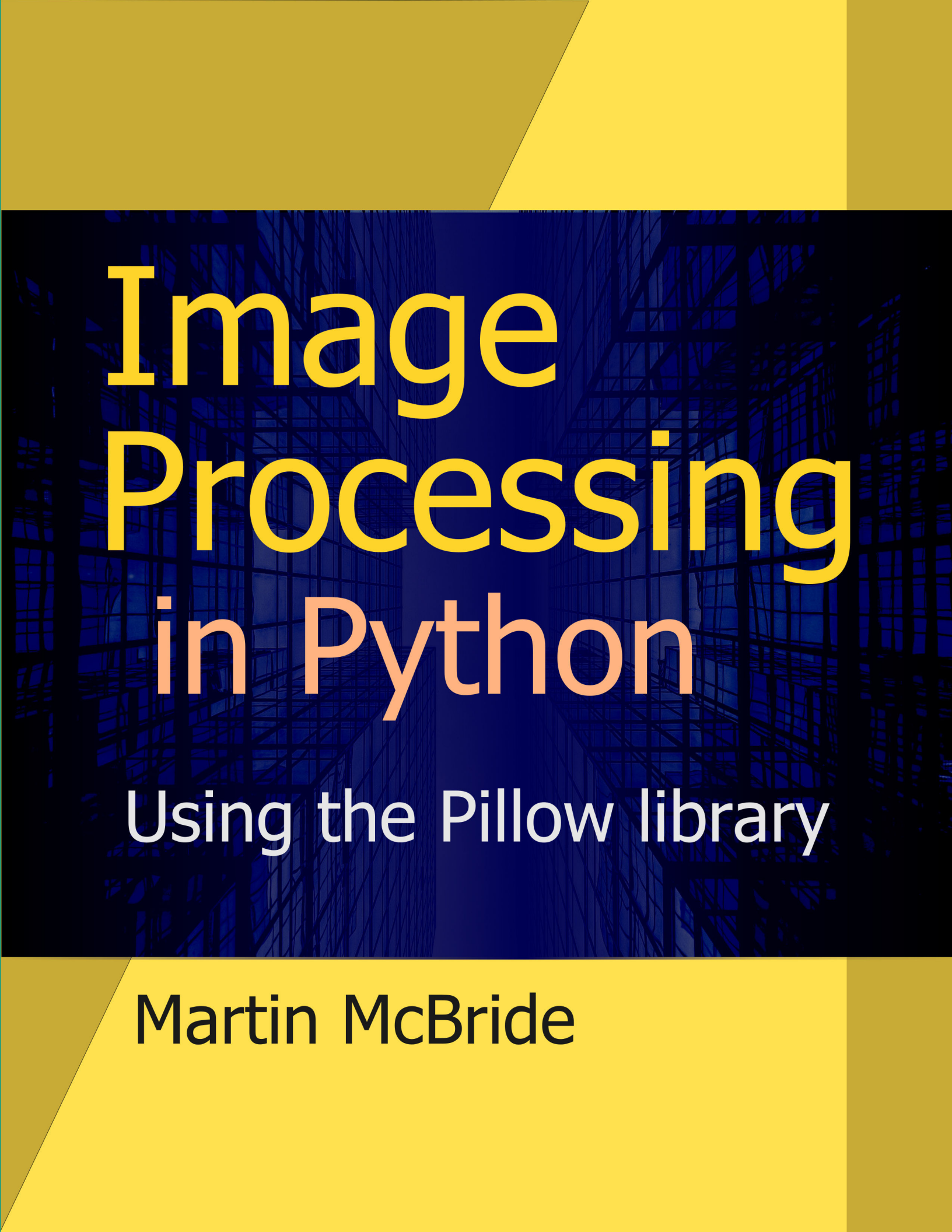


Image Processing in Python
Using the Pillow library
Martin McBride

Image Processing in Python

Processing raster images with the Pillow library

Martin McBride

This book is for sale at http://leanpub.com/imageprocessinginpython

This version was published on 2021-08-22

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Preface

This book provides an introduction to the basics of image processing in Python, using the Pillow imaging library. After reading this book you should be able to create Python programs to read, write and manipulate images.

1.1 Who is this book for?

This book is aimed at anyone wishing to learn about image processing in Python. It doesn't require any prior knowledge of image processing, but it will also be useful if you already have experience working with image data and would like to learn the specifics of the Pillow library.

It will be assumed that you have a basic working knowledge of Python, but all examples are fully explained and don't use any advanced language features.

1.2 About the author

Martin McBride is a software developer, specialising in computer graphics, sound, and mathematical programming. He has been writing code since the 1980s in a wide variety of languages from assembler through to C++, Java and Python. He writes for PythonInformer.com and is the author of several books on Python. He is interested in generative art and works on the generativepy open source project.

1.3 Keep in touch

If you have any comments or questions you can get in touch by any of the following methods:

- Joining the Python Informer forum at http://pythoninformer.boards.net/[1].
- Signing up for the Python Informer newsletter at pythoninformer.com
- Following @pythoninformer on Twitter.
- Contacting me directly by email (info@axlesoft.com).

[1] http://pythoninformer.boards.net/

2. Introduction

This book is about bitmap imaging in Python. It is divided into two sections:

- Bitmap images - introduces some important concepts of bitmap imaging, including colour representation, pixel data models, image compression, file formats and metadata.
- Pillow library - a detailed tutorial on the Python Pillow imaging library, one of the most popular Python imaging libraries.

There is also a Reference section, containing some useful information that you will probably need to refer too, all gathered in one place.

2.1 Versions

This book uses Pillow version 8.2.0 and Python version 3.9.

The examples will work with older versions (Pillow version 5 or later, Python version 3.6 or later).

There is a good chance the examples will also work with newer versions.

2.2 Example sources on github

You can find example images and source files on github, at https://github.com/martinmcbride/python-imaging-book-examples

I Bitmap images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

3. Introduction to bitmap imaging

This part of the book covers *bitmap imaging*.

This chapter will cover the basics of what a bitmap image is. Later chapters will cover:

- How computers represent colour.
- Colour models.
- Colour resolution.
- How image data is stored in memory.
- Transparency.
- Image compression.
- Image file formats.
- Colour management.

3.1 What is a bitmap image?

You most likely already know what a bitmap image is. Almost any image you see on the web will be a bitmap image, and you have probably used your smartphone or digital camera to capture photographs as bitmap images.

You might be more familiar with alternative names - raster image, or pixel image. They mean the same thing as bitmap image. They are sometimes also called JPEG images or PNG images, named after specific image file formats.

You probably also know that a bitmap is made up of *pixels* - they can be thought of as tiny coloured squares that make up the image. They are normally too small to see but become visible if you zoom in too far and the image becomes *pixelated*.

This chapter presents an overview of the characteristics of bitmap images, in preparation for the remaining part of this section that looks at bitmap images in detail.

3.2 Spatial sampling

A real-world scene has an almost infinite amount of detail. If you look out at, for example, a boat by a lakeside, it goes beyond the detail your eye can see. You could walk up to the boat and look at it in virtually unlimited detail.

A bitmap image has a finite amount of detail. If you took a digital photograph of the boat, the camera would convert it into an array of pixels. Each pixel represents a small part of the image.

If the pixels are very small together, we can't distinguish the individual, and the image looks similar to the actual scene. If they are larger, we see the image as a set of pixels rather than a natural image, as this illustration shows:

In practical terms, at a viewing distance of 40 cm, the eye can resolve objects that are about 0.1 mm apart. So for example, imagine a sheet of paper with two thin lines drawn 0.1 mm apart. If you held the paper 40 cm in from of your face, assuming you have normal eyesight, you might just about be able to see that there were two separate lines. Any further away and they would just look like a single line.

This means that if you wanted to print an image on a page so that the eye couldn't see the individual pixels, you should aim for a pixel resolution of 10 pixels per mm (about 250 pixels per inch) or better. So for a printed photograph or 15 cm by 10 cm, you would want an image of at least 1500 by 1000 pixels. That would be a 1.5 megapixel (MP) image.

In reality, you would probably want a higher resolution than that:

- To allow you to print larger photographs.
- To allow you to crop a photograph to show just the subject.

Most modern digital cameras support image sizes of 6 MP or higher.

However, very large pixel sizes are not always as useful as they might seem:

- For printing very large images, such as posters, they are normally viewed from further back than 40 cm, so there is no need to have 0.1 mm spatial resolution.
- At very high resolutions, factors such as camera shake will blur the image, so there is little point in taking a very high-resolution photograph without a very solid tripod.

3.3 Colour representation

Each pixel in a bitmap image has a specific colour. There are various ways we might represent a colour in an image.

The most common way to represent a colour is as three separate components, the amount of red, green, and blue light that make up the colour. As we will see, this is based on the way the human eyes perceives colour. However, we sometimes use alternate methods, including:

- CMYK - used to represent colours for printing.
- HSL - used in art and design as an intuitive way to select related colours.
- Perceptual colour spaces such as CieLAB used to create very accurate colours.

An important consideration is how precisely we need to represent each colour in an image. As rough a rule of thumb, we can detect variations in colour of about 1%. Most modern systems use 8 bits per channel (for example, three integers between 0 and 255 to store the red, green, and blue values). This gives a precision of better than 0.5%, which is adequate for most uses.

However, some image formats store data using fewer or more bits per colour, so it is useful to understand the different formats available.

Finally, we need to be aware that a computer system cannot capture or display the full range of colours available in the real world. The most obvious aspect of that is colour intensity. In the real world, we might encounter the full glare of the sun or the total darkness of a deep cave. There is no possibility of replicating that range of intensities on a computer monitor, and even less possibility of recreating it on a computer printout. A printer can't create anything brighter than the paper itself, and it can't create anything darker than the black ink (which is probably dark grey at best).

Leaving aside the intensity, many natural colours are too pure and vibrant to be accurately recreated by a screen or printer. We can only approximate them on a computer.

3.4 File formats

Bitmap images are often stored in a file, and there are many different types of file formats in use.

There are three main aspects of an imaging file format that make it what it is:

- All bitmap images must store the bitmap data, of course. Different file types have different capabilities in terms of the colour models and bit depths they support.
- Image data is often very big, so many file formats support data compression. There are many schemes. Some are more efficient than others, and some are more suited to specific types of images.
- Finally, all file formats support *metadata*, see the note.

Metadata means *data about other data*. It is extra information in an image file that describes the image data. This can vary from the most basic information (such as the image dimensions and colour type) right through to highly detailed information about the camera settings used when the image was taken. Each format has a different set of capabilities.

These days the most popular formats are:

- JPEG format for photographic images, mainly because it supports a very efficient form of lossy compression that works well in photographs.
- PNG format for diagrams and logos, mainly because it supports a very efficient form of lossless compression that works well with artificial images.

- GIF format, which isn't used much for normal images, but has a unique feature of supporting simple animations, that makes it useful for certain web pages.
- TIFF format, which is mainly used for professional printing applications. It is a very capable format that supports a huge range of data types, compressions schemes, and metadata, but it is also quite complex so it isn't supported by browsers.

There are many other file types, particularly for small files like icons, because they are small so they don't require sophisticated compression. They are often specific to particular operating systems or types of software.

3.5 Vector images

An alternative way of storing image data is to use a vector format. In a vector image, the image isn't stored as pixels. Instead, it is stored as a set of mathematical definitions of shapes. A vector image is essentially a list of lines, rectangles, circles and other shapes. It defines the exact size, position, and colour of each shape, often as human-readable text.

To view a vector image it must first be *rendered*, that is it is converted to a bitmap by, essentially, drawing each shape. This process is usually highly optimised.

Advantages of vector images are:

- The file size is often much smaller than the equivalent bitmap image.
- The image can be rendered at any resolution because the exact positions of every shape are stored. You can zoom in on the image almost infinitely and it will still have perfect edges.
- Individual shapes in the image can be edited.

Disadvantages are:

- The image must be rendered before it is displayed, which requires more processing power than a simple bitmap.
- There can be compatibility problems. Since the format is typically more complex, there are more ways for things to go wrong.
- It only really works for artificial images (diagrams, text documents etc), you can't efficiently store a natural photograph in a vector format.

Well-known vector formats are SVG, PDF and PostScript. We don't cover vector formats in this book, it is a whole separate topic, but it is covered in my book *Computer Graphics in Python*.

4. Computer colour

In this chapter we will take an in-depth look at how computers represent colour:

- Visible light - what is colour?
- How we see colour.
- The RGB colour model.
- Colour resolution.
- Greyscale colour model.
- Transparency
- The CMYK colour model.
- HSL/HSB colour models.
- Perceptual colour models.
- Colour management.

4.1 Visible light

Visible light - the light our eyes can see - is a form of electromagnetic radiation. Electromagnetic radiation refers to waves in the electromagnetic field that radiated through space, carrying energy.

Radio waves, microwaves, infra-red, visible light, ultraviolet, x-rays, and gamma rays are all different types of electromagnetic radiation. They have different names because some of them were first discovered and investigated by scientists before anyone realised they were linked.

Despite these phenomena appearing to be very different, they are in fact all exactly the same thing - oscillations in the electromagnetic field that travel through space at the speed of light. The difference is the oscillation frequency.

For example, analogue radio waves used by public broadcasters have frequencies of between 300 kHz (300 thousand oscillations per second) for medium wave AM, through to 300 MHz (300 million oscillations per second) for FM radio. On an analogue radio, you will tune to a particular frequency to listen to a particular station (eg 95 FM means 95 MHz).

4.1.1 Frequency and wavelength

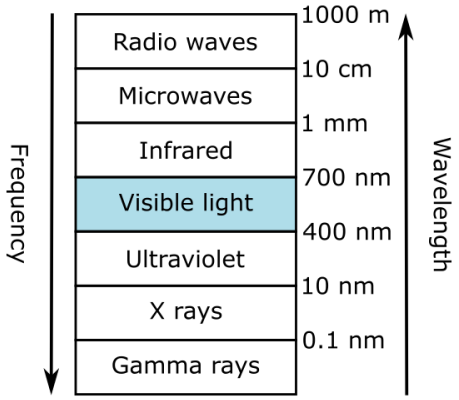
All electromagnetic radiation, including light, travels at a speed of 299 792 458 metres per second. That is the speed of light in a vacuum, often called c (light travels very slightly slower if it passes through materials such as air, glass or water, but the difference is a tiny fraction of 1%).

For a particular frequency f, electromagnetic radiation has a wavelength λ given by:

```
λ = c / f      where c is speed of light
```

For example a 100 MHz radio signal has a wavelength of 3 metres.

This diagram shows some well-known types of electromagnetic radiation, listed in order of decreasing wavelength:



Visible light occupies a very small range of the spectrum, between infrared and ultraviolet, with wavelengths between about 400nm and 700nm. *nm* stands for nanometre, and 1nm is equal to a millionth of a millimetre.

4.2 What is colour?

We can see electromagnetic radiation in the range 400 nm to 700 nm, but why do we see different colours?

A simple explanation is that different wavelengths appear as different colours. Here is an illustration:

Light with a wavelength of around 700 nm looks red, light with a wavelength of about 610 nm looks orange, and so on. You may recognise these as being the seven colours of the rainbow.

The last three colours of the rainbow are classically called blue, indigo, violet. If you look at an actual spectrum it is more accurate to describe the last three bands as cyan, blue, violet. It is possible that when Newton first described the colours, what he meant by blue was something closer to what we now know as cyan (a light sky blue).

But there are far, far more than 7 colours in the spectrum. Here is an illustration of the spectrum between red and orange:

As you can see there is a whole range of subtly different colours present, that mix different amounts of red and orange. The same is true for each of the other adjacent colours. In fact, there is an almost infinite range of different colours in the visible spectrum, although some of them are so similar that the human eye can't even tell them apart.

These are known as spectral colours. They are colours that correspond to light of a single wavelength, and they are also the colours that appear in a rainbow.

4.2.1 Non-spectral colours

Most of the light we see doesn't contain just a single wavelength of light. That is because most light sources (such as sunlight or many forms of artificial light) contain a mixture of many different wavelengths. When that light bounces off a surface, that surface will absorb or reflect different wavelengths to differing degrees.

This means that most of the light the enters our eyes contains a mixture of different wavelengths. But we perceive it as being a single colour.

Here are some colours that don't exist on the spectrum:

There is no single wavelength that looks hot pink or white. Those are colours that we can only see if a certain combination of different wavelengths is present

4.3 How we see colour

If you think about the infinite number of colours in the spectrum and then think about the number of ways you could mix every combination of those colours in different amounts, it might seem like an impossible task to replicate that on a computer screen.

Fortunately, the way we perceive colour a little simpler than that.

The human eye contains three types of colour detecting cells, called *cones*, that measure the intensity of light across different, broad parts of the spectrum:

- L-cones detect light towards the yellow/orange/red end of the spectrum.
- M-cones detect light in the mid-range green/yellow area of the spectrum.
- S-cones detect light at the blue/violet end of the spectrum.

Each type of cone measures the average amount of light in the part of the spectrum it can detect. These detection bands overlap, and by measuring the relative amount of light in each of these bands, our brain can recreate every colour that we see.

The important fact here is that colour *as humans perceive it* is a 3-dimensional quantity - the amount of light detected by the three types of cones. If the light coming from a computer monitor stimulates those cones in the same way as the light from a real object, then the colour will appear very similar.

These cones do not exactly correspond to red, green and blue, but if we mix different amounts of red, green and blue light we can simulate many of the colours we see.

The eye also contains a second type of cell, called *rod* cells. Rods sense light and dark, but they don't see colour. They are more sensitive than cones, so they provide your ability to see in dark conditions (that is why you don't see colours when it is dark). The eye contains more rods than cones, so they can detect finer detail than cones.

4.4 The RGB colour model

The most natural, and most common, way to represent colour on a computer system is to model the way we see colour, that is to use threes values, red, green and blue. Each unique combination of red, green and blue creates a unique colour. We call this the RGB colour model.

We say that red, green and blue are the three *components* of the RGB model. Alternatively, they are sometimes called the three *channels*, it means the same thing.

4.4.1 Displaying colour

On a computer screen, each pixel has three tiny elements. Typically these are LCD cells whose transparency can be controlled by an electrical signal. These act as a variable source of red, green and blue light. By allowing the correct amount of light from each component, we can set each pixel to any colour.

We store the required colour of each pixel as an array in the computers video memory, with each pixel represented by 3 elements in the array. The video hardware controls the colour of each pixel on the screen.

4.4.2 Representing RGB colours as a percentage

There are several ways to represent an RGB colour, but they all amount to the same thing: colour is specified by three numbers, that represent the amounts of red, green and blue that make up that colour.

The first way is to use a percentage for each component. So for the red value:

- 0% means that the colour has no red component. For example, if the colour is pure blue then it has no red component.
- 100% means that the maximum possible amount of that component is present. For example, the brightest pure red contains the maximum possible amount of red.
- 50% means that the colour contains half the maximum amount of red.

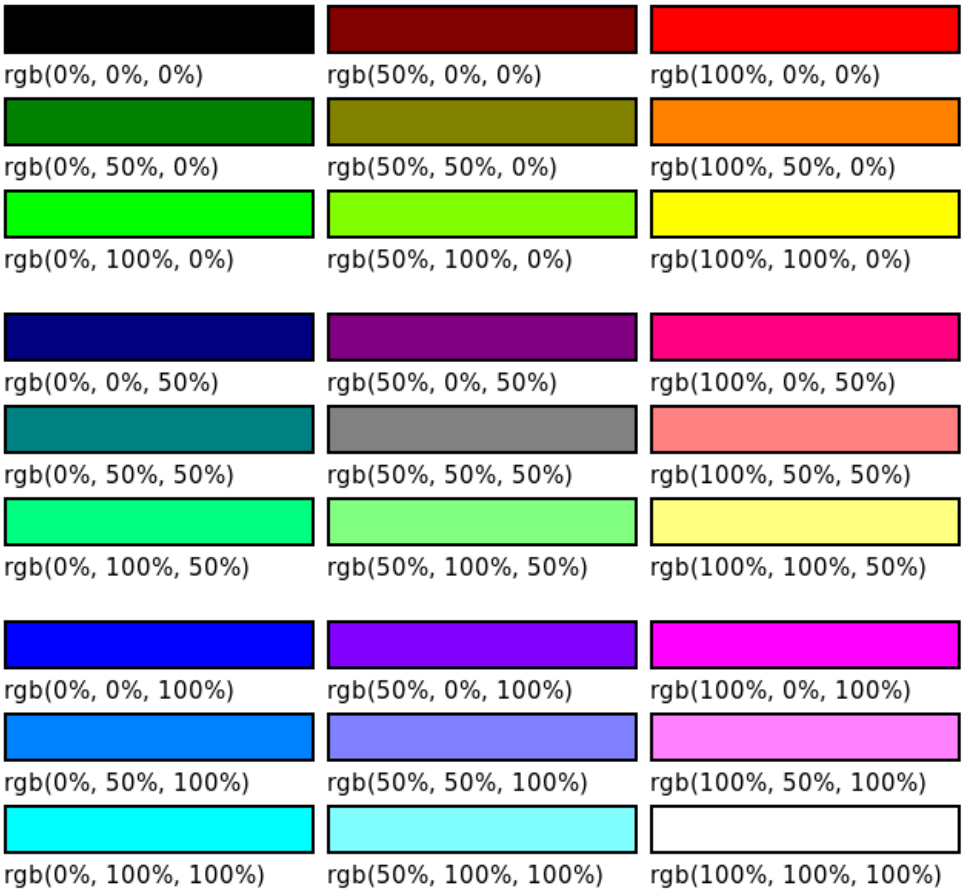
Similar for green and blue.

We can specify any RGB colour using percentages, like this:

rgb(100%, 50%, 0%)

This indicates a colour that contains the maximum amount of red, 50% of the maximum green, and no blue. That would give an orange colour.

Here are a selection of example colours:



This table shows every combination of red, green and blue values of 0%, 50% and 100%.

4.4.3 Floating point representation

An alternative way to represent an RGB colour is to use numbers in the range 0.0 to 1.0. This works in the same way as percentages, but with fractions instead of percentages:

- A value of 0.0 corresponds to 0%.
- A value of 1.0 corresponds to 100%.
- A value of 0.5 corresponds to 50% and so on.

So the value:

rgb(100%, 50%, 0%)

would be represented by three floating-point values `(1.0, 0.5, 0.0)`.

The Python vector graphics library Pycairo represents colours in this way. The numerical processing library NumPy sometimes uses this method to store images.

4.4.4 Byte value representation

The final method is to represent each colour channel as an integer value between 0 and 255:

- A value of 0 corresponds to 0%.
- A value of 255 corresponds to 100%.
- A value of 128 corresponds to (approximately) 50% and so on.

So the value:

rgb(100%, 50%, 0%)

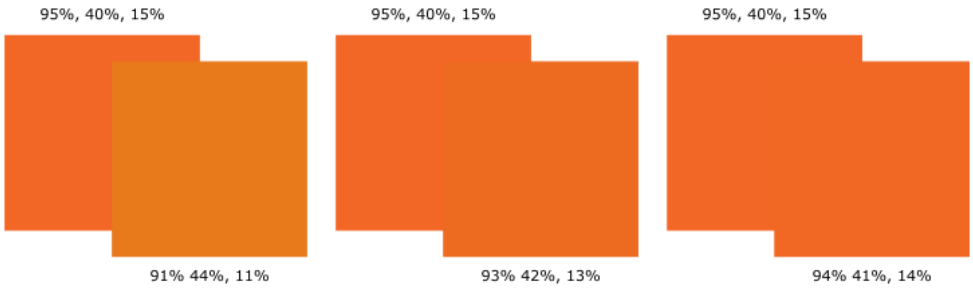
would be represented by three integer values `(255, 128, 0)`.

There is a reason we choose the range 0 to 255. It is the range that can be stored in an unsigned byte. This means that a colour can be stored in exactly 3 bytes of memory, so an image can be stored using 3 bytes per pixel.

It turns out that 256 levels of each colour are enough to be able to represent colours precisely enough for most purposes - it is good enough for photographs and videos, for example. We will look at this next.

4.5 Colour resolution

The human eye can see many different colours, but there are limits to our perception. When two colours are very similar, they appear to be identical - we can't see the difference, even if they are side by side. Here is an example:



This shows three pairs of overlapping orange squares.

The first pair (on the left) have colours `rgb(95%, 40%, 15%)` and `rgb(91%, 44%, 11%)`. The red, green and blue values of the two squares differ by 4%. Although they are both similar shades of orange, you can probably see that they are different colours.

The second pair (in the middle) have colours `rgb(95%, 40%, 15%)` and `rgb(93%, 42%, 13%)`. The colour values of the two squares differ by 2%. The two colours are very similar, but you might just be able to see that they aren't exactly the same.

The final pair (on the right) have colours that differ by just 1%. It is very difficult to see the difference between them. To the eye, they are effectively identical.

We normally store RGB images using 1 byte (8 bits) per colour per pixel. A byte can store integer values 0 to 255, which is 256 distinct values. This means each colour channel can be represented with a precision of better than 0.5%. Since we can't see any difference between two colours that differ by 1%, this precision is perfectly adequate for most uses.

4.6 Greyscale colour model

Any RGB colour that has equal amounts of red, green and blue, will display as a shade of grey, for example:

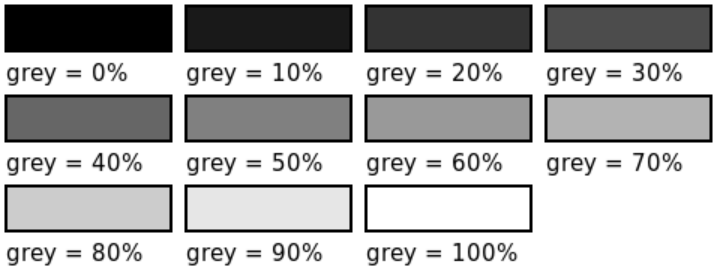
- rgb(0%, 0%, 0%) is black.
- rgb(30%, 30%, 30%) is dark grey.
- rgb(100%, 100%, 100%) is white.

In the greyscale colour space, the colour is specified by a single value. The colour is displayed as if the red, green and blue values were all equal. You can think of greyscale as a subset of RGB that includes only the pure grey colours:

- Grey value 0% is black.
- Grey value 30% is dark grey.
- Grey value 100% is white.

Unlike RGB, the greyscale colour model only has one component, the grey value. This will typically be stored as a single byte value.

This image shows various grey values from 0% to 100%



Greyscale is useful for images that don't include any colour, for example:

- Text only documents.
- Charts and diagrams that don't use colour.
- Scans of black and white photographic images.

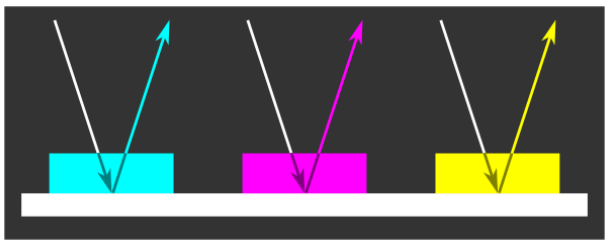
4.7 The CMYK colour model

When we print an image, we normally apply different coloured inks to white paper to create the colours we require.

A printed page works quite differently from a computer screen. A screen starts is dark by default and adds red, green and blue light to create colours. We call this the additive model

A printed page doesn't generate light, it simply reflects light. A blank page starts off white (assuming it is white paper viewed under white light), and the ink on the page absorbs different amounts of red, green and blue light, to leave the required colour. We call this a subtractive model.

Many printing processes work by adding several layers of different coloured *translucent* inks. Here is an illustration:

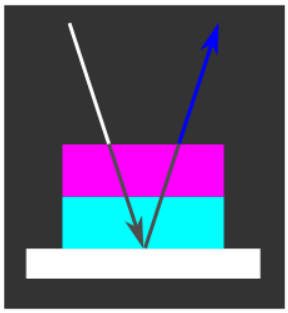


Because the ink is translucent, light passes through the ink and reflects off the white paper underneath. The ink acts as a colour filter, removing certain colours.

The left-hand part of the image shows what happens when we place cyan on a white page. It might seem counter-intuitive at first, but cyan ink works by absorbing red light. More precisely it absorbs red light but allows green and blue light to pass through, so the resulting colour is cyan (green plus blue).

Magenta ink filters out green light, letting red and blue light pass through (magenta is red plus blue). Yellow ink filters out blue light, letting red and green light pass through (yellow is a mixture of red and green).

If we put one layer of ink on top of another, light passes through both layers and reflects off the page. For example:



In this case, magenta ink has been layered over cyan ink. The cyan ink removes the red light, the magenta ink removes the green light, so we end up with a blue colour on the page. In fact, by layering different amounts of cyan, magenta and yellow ink on the page, it is possible to remove different amounts of red, green and blue from the reflected light, which allows you to print any colour.

4.7.1 The K component

If you own a colour inkjet printer or similar, you will probably know that it has 4 colours, cyan, magenta, yellow and black. The K part of CMYK stands for *Key*, which is an old printing term for the black plate in a traditional printing press.

In theory, we shouldn't need a separate black ink, because mixing cyan, magenta and yellow ink together should create black. But in reality, there are several advantages to using black ink:

- Equal quantities of cyan, magenta and yellow ink don't create a perfect black. They usually create a very dark brown.
- It is very difficult to place the three colours in exactly the same place on the page. Any slight misalignment will make black shapes slightly blurred, which particularly affects black text.
- Black ink is a lot cheaper than coloured ink.
- A lower total amount of ink is used, which can help the ink to dry faster and avoid paper stretching, particularly on traditional printing presses.

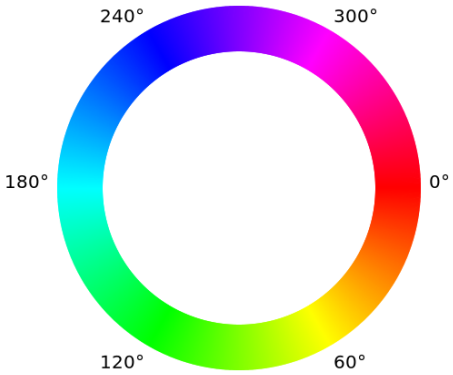
Almost all colour printing processes - from home or office printer, through to newspapers or magazines printing presses - uses a separate black channel, for these reasons. In some cases the printer also adds black ink into darker coloured areas, to reduce the amount of coloured ink that is needed.

4.8 HSL/HSB colour models

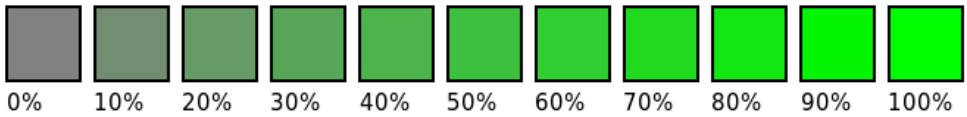
The HSL colour space stores colours as 3 values:

- The Hue (H) indicates the basic colour, essentially the position of the colour on the colour wheel that goes from red to green to blue then back to red.
- The Saturation (S) controls how saturated the colour is. A value of 100% indicates a pure colour, 50% represents the same colour but mixed with grey, and at 0% the colour is pure grey.
- The Lightness (L) controls how light or dark the colour is. Lightness of 50% shows the basic colour at a medium level of lightness. If the L component increases towards 100%, the colour gets lighter and lighter until it eventually becomes white. If the L component decreases towards 0%. the colour gets darker and darker until it eventually becomes black.

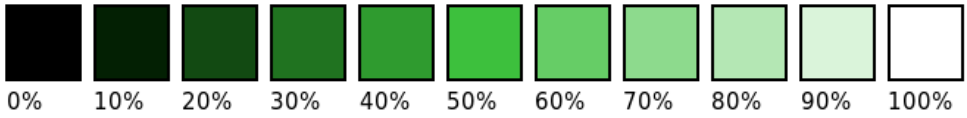
This diagram shows the colour wheel of hue values. The hue can be expressed as an angle between 0 and 360 degrees, where 0 is red, 120 is green, and 240 is blue:



Here is the effect of changing the saturation between 0% and 100% (with a hue of green and lightness of 50%). Notice that the basic colour remains the same:



Finally, here is the effect of changing the lightness between 0% and 100% (with a hue of green and saturation of 50%). Again, the basic colour remains the same:



HSL colours are useful in art and design applications because they allow sets of related colours to be created very easily.

4.9 HSL variants

The HSB (hue, saturation, brightness) colour space uses the same definition of hue as HSL. Saturation and brightness behave differently but can achieve the same range of colours as HSL.

The HSL model works well with the additive model (as used by RGB). HSB is based on a subtractive colour model, but CMYK is more useful for subtractive colours. HSB tends not to be used as often as HSL, so we will not cover it in detail.

You may also see the term HSV (hue, saturation, value). This is an alternative name for HSB.

4.10 Perceptual colour models

RGB is quite a crude attempt to model how the eye perceives colour, although is remarkably useful for many day-to-day uses. You can watch a movie or view your holiday snaps on an ordinary computer screen without necessarily feeling that there is anything seriously wrong with the colours.

But for more demanding applications, RGB is not good enough. For example, suppose you were in the business of making fine art prints. You need to scan an original painting, then create prints that have *exactly* the same colours. So if you hold the print up next to the original artwork, you want them to look identical.

If you were to use an ordinary desktop scanner and an ordinary desktop printer to do this, the result might look very nice, but it wouldn't look like the original. If you placed them side by side, the colours would not be the same. It would be fine as a cheap poster, but not as an expensive print that is meant to show the exact colours in a Turner seascape.

The basic problem is that red, green and blue aren't defined very well. If you display a 100% pure red circle on your monitor, it will certainly look red, but there is no what of knowing *exactly* what red colour it will be. Indeed you can change the appearance of the colour by adjusting your monitor settings.

Similarly, if you print that red circle, the exact result can vary greatly - it will depend on the type of printer, the type of ink and indeed the type of paper. It will depend on whether the ink cartridges are full or nearly empty. It will be slightly different for two different printers of the same type, and it will be slightly different for the same printer on a different day. How can you hope to match the colours of the original scene or painting?

The first thing we need to do is to establish a standard, defined colour space, so we can define exactly what "red" means. In fact, we don't usually use RGB for this, because RGB isn't a linear space. For example, if you look at 3 blocks of colour, 100% each of red, green and blue, the green block will look brighter. The same "amount" of green appears brighter because our eyes are more sensitive to green.

As a first step towards standardising colours, we usually adopt a *perceptual colour model* that assigns colour values that appear more linear to the eye.

4.10.1 CIE spaces

We commonly use CIE colour spaces. These are based on the work of the International Commission on Illumination (CIE), which did many experiments in the 1930s to establish a standard model of human vision.

A commonly used standard space is CIELAB. This space has three components:

- L represents the lightness of the colour (roughly equivalent to the L component of HSL).
- A represents the position of the colour on a scale from red to green.
- B represents the position of the colour on a scale from blue to yellow.

This is not as easy to imagine as RGB values but is it a better way to represent what we perceive. If we express a colour in CIELAB we have a precise definition of what that colour is.

CIELAB isn't the only standard. CIEXYZ is another commonly used perceptual colour space that works similarly.

4.11 Colour management

A standard way of representing colours is half the solution. We still have the problem that our input devices (camera, scanner etc), computer monitor, and printer all use RGB, and each has its own idea of what RGB means.

Colour management solves this problem by using a *profile* for each device, that allows us to convert between the device's RGB values and standard CIELAB.

Profiles are determined by measuring the device. For example, to profile a scanner we might scan a page that has lost of different colours, each of which has a known CIELAB colour. We can look at the RGB values the scanner produces for each different CIELAB colour, and derive a *transformation* to convert between RGB and CIELAB. We can do the same for monitors and printers, by printing different RGB colours and measuring the result with a colour meter.

Most users don't make these measurements themselves, they simply install profiles supplied by the manufacturers that convert colours based on the typical characteristics of the model of scanner, printer or monitor.

So in principle, we can scan an image and convert it from the scanner's RGB to CIELAB. To display it we convert the CIELAB to the monitor's RGB, and to print it we convert the CIELAB to the printer's RGB.

In reality, the image would normally be stored in RGB format, but with the scanner's profile attached to the image. When we display the image, we apply a combined conversion, from scanner RGB to CIELAB then back to monitor RGB. This achieves the same result, but it has the advantage that the image stored in the system is in RGB format. If you wanted to use the scanned image without colour management (for example, if you wanted to put the image on a website) you can just ignore the attached profile and use the image as a normal RGB image.

4.11.1 Gamuts

There is one more topic that needs to be considered in colour management - the *colour gamut*.

If you take a photograph of a real-life scene, the camera might not be able to capture all the colours exactly. Some might be too bright to register properly. Some colours might be too vibrant or intense. We call the set of colours that the camera can capture its *gamut*.

If we view that image on a monitor, we will find that colours are limited even further. The white produced by a monitor isn't very bright (nor would you want it to be) and the black is equivalent to the monitor when it is switched off, which isn't very dark. The intensity of different colours is limited too. In general, the gamut of a computer screen is smaller than the gamut of an input device.

Most printers have an even smaller gamut than a screen. Compare a white sheet of paper (the brightest colour a printer can produce) with a white area on your computer monitor. The paper is nowhere near as bright.

There is no way around this, images on your screen or printer will never have the contrast and vibrancy of a real-world scene. But there are choices in how this limitation is handled. You can select a *rendering intent*:

- Perceptual - this adjusts the colours so that the image looks natural. It tries to achieve an overall natural appearance even if that means the colours are not completely accurate. It is suitable for photographs.
- Relative colorimetric - this tries to preserve the correct colours in the image. This means that colours that are in gamut will be displayed correctly, but colours that are out of gamut will be converted to the nearest available colour. This makes the image as accurate as possible, but out of gamut areas may appear blocky. In this mode, all colours are scaled to take account of the white point (essentially, the brightest white available on the monitor/printer).
- Saturation - this attempts to preserve saturated colours, even if it means non-saturated colours are less accurate. This is useful for things like business documents, where black text and bright colours in graphs and charts need to be displayed as fully saturated.
- Absolute colorimetric - similar to relative colorimetric, but it doesn't scale for the white point. It attempts to make every in gamut colour as accurate as possible in absolute terms, but that might result in a lot of out of gamut colours looking incorrect and blocky.

5. Bitmap image data

In the previous chapter, we saw how computers represent colour.

A particular colour is usually represented by several values. For example, an RGB colour is represented by three values, the amount of red, green and blue in the colour. Each value is normally stored as an integer of a certain range, that maps on to a percentage value for that colour.

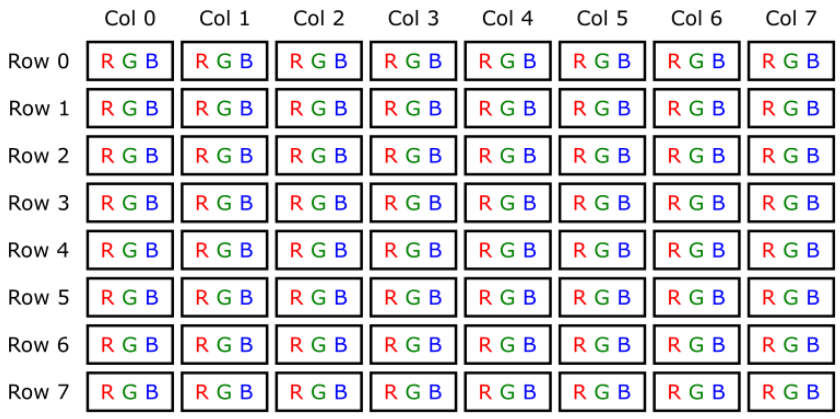
A bitmap image consists of a two-dimensional array of pixels, each with its own colour. The bitmap data for that image consists of an array of colour values, one for each pixel. In this chapter, we will look at how bitmap data is stored.

We will cover:

- How bitmap data is stored in memory or files.
- Pixel formats and colour depths.
- Palette based images.
- Ways of handling transparency.

5.1 Data layout

Consider a very small image (say an 8 by 8 pixel icon) has a total of 64 pixels. We could visualise it like this:



However, computer memory is one-dimensional, so pixel data has to be stored sequentially. Typically, the data is stored in *scanline order* - that is, the first line, followed by the second line, and so on:

This isn't the only possible ordering, but it is very common. The data stores the colour information for the 8 pixels of the first row, followed by the colour information for the 8 pixels of the second row, etc.

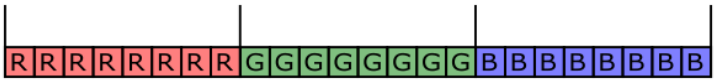
When bitmap data is stored in an image file, it often uses a similar format.

5.2 8-bit per channel images

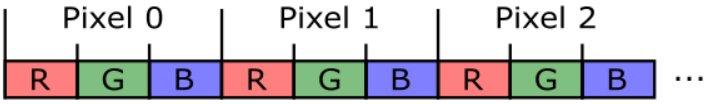
Most bitmaps are stored using 8 bits (ie 1 byte) per colour per pixel. This applies to various colour models.

5.2.1 24-bit RGB

For RGB images, the most common way to store data is to use 1 byte (or 8 bits) per colour per pixel. This means that each pixel occupies 3 bytes (or 24 bits), like this:



In this case, the first byte represents the red component of the pixel, the second represents the green, and the third represents the blue. Successive pixels are stored one after the other in memory, like this:



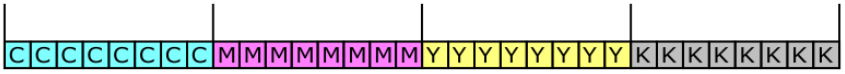
This is a very convenient format for two reasons:

- One byte provides 256 possible brightness levels for each colour. As we saw earlier, this number of levels is enough to provide good image quality.
- Having each pixel channel in a separate byte means it can be read/written straight from the memory buffer without needing to rearrange bit ordering (unlike other formats described later).

We have assumed a colour order of red, green, blue. This is the most common ordering, but some formats may use a different ordering, for example blue, green, red. You should check the specific format of data you are using to make sure you have the correct order. This same consideration applies to all the formats described here.

5.2.2 32-bit CMYK

A similar format can be used to store CMYK bitmap data. This time each pixel requires 4 bytes (32 bits) to accommodate the cyan, magenta, yellow and black components:



5.2.3 8-bit greyscale

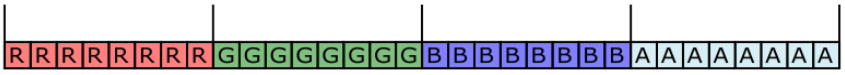
For greyscale data, there is only one channel, so each pixel occupies a single byte:

5.2.4 32-bit RGBA

RGB images are sometimes stored with an alpha channel for transparency. This means that the transparency of every pixel can be controlled. It can be:

- Fully opaque, that is it completely hides anything behind it.
- Fully transparent, so the pixel is completely invisible.
- Partially transparent, so that the pixel behind is partly visible.

There are 256 levels of transparency available.



5.3 Bitmap data with fewer levels

When personal computers first started to become popular, from the late 1970s, computer memory was very expensive, especially the faster memory required for video hardware. In addition, disk drives hard far lower capacity, and networks were generally a lot slower.

To make PCs affordable, most early video systems used only 1 byte or 2 bytes per pixel, which meant that they couldn't display as many colours.

These days, when memory, large disks, and fast networks are readily available, this is no longer a consideration. However, some of the image file formats we still use today support these modes, so we will describe them here.

Try to avoid using these modes if you possibly can, the quality ranges from poor to terrible.

5.3.1 8-bit RGB

In 8 bit RGB, the whole RGB colour is stored in a single byte. One possible scheme looks like this:

Here, each colour is stored as a 2-bit quantity. This means that there are only 4 possible levels of each colour:

- 0 corresponds to 0%
- 1 corresponds to 33.3%
- 2 corresponds to 66.7%
- 3 corresponds to 100%

This has the obvious advantage that it uses a third of the amount of memory that full RGB uses. But it has a major disadvantage that it can only reproduce a very limited set of colours. This creates a very poor quality image, like this (the original 24-bit version is on the left, the 8-bit version on the right):

One further disadvantage of this scheme is that you need to perform bitwise operations to separate the red, green and blue components. Video hardware that supports this encoding usually has

hardware support for these operations, which makes the design of the board marginally more complex.

This scheme only uses 6 of the available 8 bits in each byte. Sometimes the extra bits were used to signify other qualities, such as transparency, or inverse colour, or even flashing pixels. Sometimes a different coding scheme was used, like this:

In this case, the red and green channels are coded using 3 bits instead of 2, so there are 8 possible levels of red and green. Blue still has 2 bits, because there are only 8 bits available in a byte. It is done this way because the eye is less sensitive to blue than the other colours.

This method makes the image quality slightly better, but it still isn't particularly good.

5.3.2 16-bit RGB

Another encoding scheme uses 2 bytes (16 bits) to store RGB values. This allows 5 bits per channel, like this:

5 bits gives 32 different levels of each colour, which gives a marked quality improvement. Here is the result:

The result is far better, although there is still some banding in areas of slowly changing colour (such as the sky, which looks a little blocky). It uses twice as much storage as the 8-bit case but still uses a third less than the full 24-bit RGB.

Using 5 bits per colour means that there is still one unused bit. As n the 8-bit RGB case, some schemes use this extra bit for special meaning (such as transparency). It is also quite common to use 6 bits for green, 5 bits for red and 5 bits for blue. Green gets the extra bit because our eyes are more sensitive to colour variations in the green part of the spectrum. The extra bit gives a very slight improvement.

5.3.3 Dithering

One of the problems with 8-bit and 16-bit RGB images is that the lack of available colours leads to "banding" in parts of the image where colours are flat and change slowly. For example, the sky might gradually vary between subtly different shades of blue, but in a 16-bit image, you will just see large areas of the exact same blue that then suddenly changes to a totally different blue.

Dithering attempts to fix this by mixing some random noise into the original image. This means that in any part of the image where pixels were originally the same value, the pixels now all have slightly different values but they average out to the original value.

When we reduce the colour depth, instead of all the pixels being placed in the same band, some get placed in different bands. Your eye will tend to average this variation out, and you will see something resembling a smooth variation (but you will also see a bit of noise).

In this image, the left-hand side shows the original smooth green gradient at 8 bits per colour. The middle section shows the same gradient reduced to 5 bits per colour - the banding is very clear to see. The right-hand section shows the same thing but with noise added:

The thing to note about the right-hand image is that every individual pixel has one of the same colours that are in the middle image. But because the pixels are mixed up, there appear to be extra colours.

While the right-hand image is clearly better than the middle image in terms of banding, the noise is quite unpleasant, so the left-hand image (full 8 bits per pixel) is the best, as you might expect.

Another point to bear in mind is that most modern image formats use compression. Noise tends to reduce the effectiveness of compression so, ironically, using 8 bits per pixel plus dithering often results in a larger file size than just using 24 bits per pixel. So you get the worst of both worlds, a poorer quality image and a bigger file.

5.4 Bilevel images

There are certain types of images that don't require colour or even shades of grey. For example:

- Black text on a white background.
- A black line drawing on a white background.

In these cases, every pixel in the image is either black or white. These are sometimes called bilevel images because pixels only have two possible levels or states.

We can represent each pixel by a single bit, which means we can store 8 pixels in one byte of data.

Bilevel images are inherently much smaller than other types of image (one bit per pixel, rather than 24 bits in an RGB image). There are also special compression methods that only work with bilevel data that achieve very good results.

Examples of systems that use bi-level images are:

- Old-style fax machines, that were used to send text documents between two locations over an analogue phone line (rarely seen outside of a museum these days).
- High-speed laser printers that are only used to print black text.
- Traditional printing presses (offset lithography) that are still used to print very large runs of identical documents (eg newspapers).

We won't cover these types of images much in this book. They are quite specialised, and most of the techniques described here don't work well with bi-level images.

5.5 Bitmap data with more levels

Modern digital cameras usually capture between 10 and 12 bits per component (some cameras go up to 14 bits). 10 bits corresponds to 1024 levels, 14 bits corresponds to 16,384 levels.

For normal use, this data is reduced to 8 bits per component and saved in JPEG format, within the camera itself. After all, the eye can't distinguish 256 different levels, let alone 16,384.

However, the extra levels can come in handy if you want to edit your photographs, especially if you want to adjust the brightness and contrast. For example, if you have taken a photograph but part of it is too dark. There may be extra detail that the camera has picked up but can't be seen in an 8-bit

image. Working with a 14-bit image you might be able to brighten those areas to reveal the details, which can then be stored as an 8-bit image afterwards.

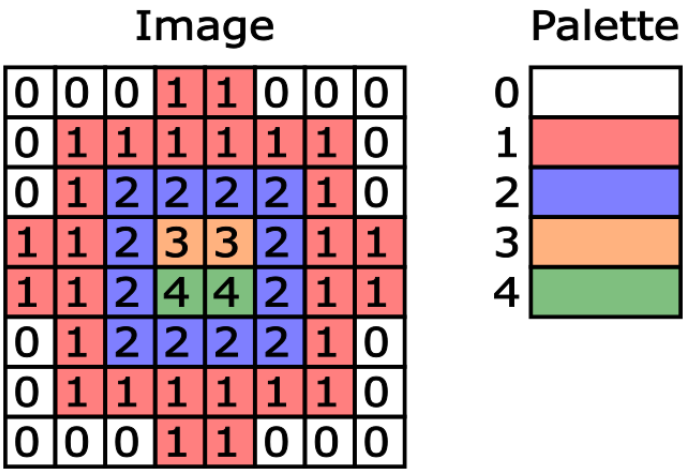
This data is often stored in the following formats:

- 10 bits per component, 30 bits in total, which is stored as 4 bytes.
- 16 bits per component, 48 bits in total, which is stored as 6 bytes. 12-bit or 14-bit data can also be stored in this format.

5.6 Palette based images

Palette based images are similar to 8-bit RGB images, but they take a slightly different approach which can give better quality.

Palette based images typically use 8 bits per pixel. Each pixel contains a number between 0 and 255, which provides an index into a colour table that represents the colour of that pixel:



In this case, our image is a simple 8 by 8 pixel icon. The image only has 5 different colours, which are stored in a separate table (the palette).

For pixels with a value 0, we take element 0 (the first element) of the palette, which is white.

For pixels with a value of 1, we take element 1 (the second element) of the palette, which is a shade of red. And so on.

Even though each pixel is stored as a single byte, the colour in the palette table is a 24-bit RGB value. So, for example, the red we use for pixels with index 1 can be the *exact* shade of red we want.

5.6.1 Images with more than 256 colours

Provided the image has 256 or less distinct colours, each distinct colour can have its own entry in the palette table, so we can give each pixel the exact right colour.

This works well for logos, diagrams, etc that have a limited number of different colours. But what if we try to store a photograph as a palette based image?

Well, a photographic image is quite likely to contain more than 256 different colours, so a palette cannot represent all the possible colours. Our image will have to use the closest available colour from the palette. There are two different ways to do this.

The first way is to use a fixed palette. In that case, the palette contains a range of different colours. The set of colours is fixed, which means the set of colours is always the same, no matter what the image contains. This is quite similar to the case for an 8-bit RGB image. If a pixel has a colour that doesn't appear in the palette, we just have to choose the closest colour that is present in the palette, and use that.

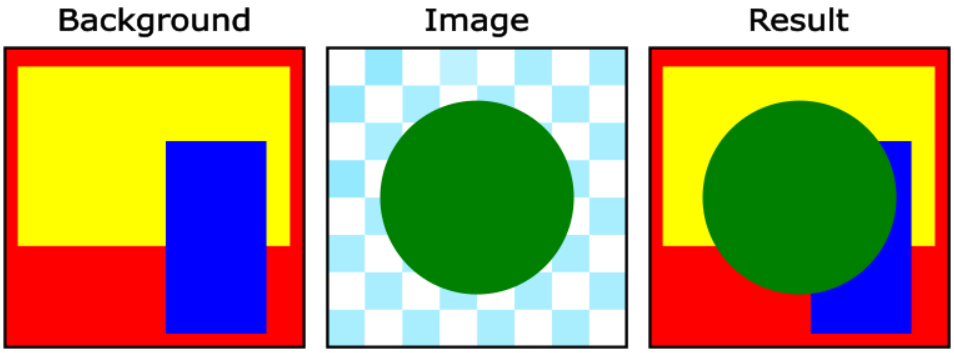
A second method is to use an *adaptive palette*. Before the image is stored, the imaging software analyses the image to see which colours are actually used. It then chooses 256 colours that best represent the image.

So, for example, imagine you have a photograph of some green grassy hills and a bright blue sky. If you save that using a fixed palette, you will only have a limited number of light blues and greens to use in your image. The standard palette will also include bright reds, oranges, purples, and lots of dark colours, that simply aren't present in the image.

If you use an adaptive palette, you can devote most of the 256 available colours to mainly light greens and blues, so you could have twice as many different colours that the image uses, and no reds, oranges or purples at all. You might still not be able to pick the exact colour for every pixel, but you will be able to choose a closer colour, so the overall image quality will be better.

5.7 Handling transparency

We discussed transparency in the *Computer colour* chapter. Transparency also applies to images. If we make certain pixels of the image transparent, that allows the background to show through, so we can achieve effects like this:



Of course, the image itself isn't really transparent. It is just that certain pixels are flagged as being transparent. It is up to the software that renders that image to decide whether to show the image or the background. If there is a transparent image on a website, for example, it is the web browser that implements the transparency.

There are three different ways that images provide transparency:

- Using an alpha channel.
- Using a transparent palette entry.
- Defining a particular colour as being transparent.

PNG format, probably the most popular format used to support transparency, uses an extra alpha channel.

5.7.1 Alpha channel

One way to implement transparency is to use an extra colour channel. For example, instead of an RGB image, we would use an RGBA image (A stands for alpha, which means transparency).

This allows the degree of transparency of each pixel to be controlled. For example:

- An alpha value of 255 makes the pixel opaque.
- An alpha value of 0 makes the pixel fully transparent.
- An alpha value of 128 makes the pixel half-transparent, so the final colour is a blend of the image pixel and the background.

This gives us 256 levels of transparency. The benefit of that is that we can use anti-aliasing on the edges of the foreground image, by making the boundary pixels partly transparent. This makes the edges appear nice and smooth.

The downside of this technique is that an RGB image now has 4 channels rather than 3, so it is larger (although image compression can reduce this effect).

5.7.2 Transparent palette entry

If we have a palette based image, another possibility is to define one particular palette entry as being transparent. For example, we might define palette entry 0 as being the transparency indicator.

When the image is rendered, any pixels that have an index of 0 will not be painted, the background will be allowed to show through.

The advantage of this technique is that it doesn't increase the image size. The disadvantage is the pixels can either be fully transparent or fully opaque, so it isn't possible to smooth the edges of the transparent image. Also, the technique only works with palette-based image formats.

GIF format uses this technique.

5.7.3 Transparent colour

A final technique that is sometimes used is to take a normal RGB format but to define a particular colour as being transparent.

For example, we might define the RGB colour (1, 1, 1) to be transparent. So if we need a pixel to be transparent, we just set the colour to (1, 1, 1).

This presents a slight problem - what if we have a normal pixel that isn't meant to be transparent, but happens to have the colour (1, 1, 1)? Well, that colour is a very, very dark grey, almost indistinguishable from black, so we can just set the colour to (0, 0, 0). It will make no visible difference.

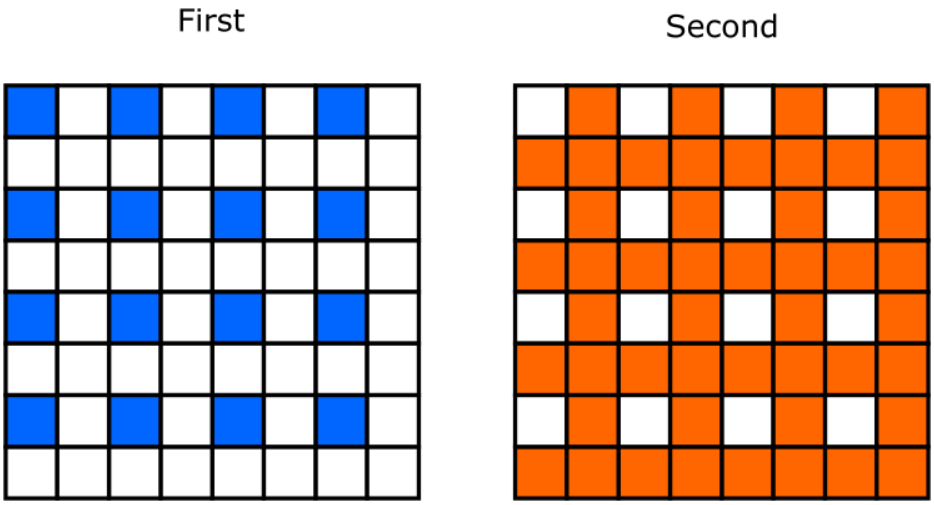
This isn't a technique that is used by any popular image formats, it is really quite hacky. It is mainly used in proprietary image formats.

5.8 Interlacing and alternate pixel ordering

All the examples above assume that bitmap data is stored in *scanline* order (all the pixels in the first line of the image, followed by all the pixels in the second line of the image, and so on.). That is what normally happens.

However, some formats have the option to *interlace* the data, which means storing the pixels in a different order. This has the advantage that it is possible to display a low resolution version of the image before all the data is available. So, for example, if you are viewing a web page over a slow connection, and the page contains a large image, the browser can display a reduced quality version the full image before it has finished downloading. It can then replace the poor quality image with the final image when it has arrived.

Here is a simple example of interlacing:



The pixels are stored in the following order:

- First, every second pixel of every second line is stored, in scanline order.
- Second, all the remaining pixels are stored, again in scanline order.

Each pixel is only stored once, but because they are stored in a different order, it means that as soon as the first set of pixels is available (when the image has only been 25% downloaded) it is possible to display a reasonable (slightly blurred) representation of the final image.

Some interleaving schemes involve more than two stages. For example, the Adam7 algorithm used by PNG has 7 stages. The first stage stores the 8th pixel of ever 8th line, which means a blocky version of the full image cab be displayed after just one 64th of the image data has been downloaded. The image then improves over a further 6 stages until the full image is displayed.

Some formats (TIFF for example) permit pixel data to be stored in a different order. For example, TIFF files can be ordered as a set of tiles, where each tile is a rectangular area of the complete image. This allows imaging software to load each section of the image independently. This is because TIFF images are often very large, and in the early days of TIFF most computers didn't have enough memory to load the entire image at once. This tends to be less of an issue with modern computers.

6. Image file formats

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.1 Why are there so many formats?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.2 Image data and metadata

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.3 Image compression

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.3.1 Lossless compression

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.3.2 Lossy compression

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.4 Some common file formats

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.4.1 PNG format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.4.2 JPEG format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.4.3 GIF format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.4.4 BMP format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

6.5 Animation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

II Pillow library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

7. Introduction to Pillow

Pillow is a Python library for image processing. It provides many image processing features similar to those might find in an imaging application like GIMP or Photoshop, but they are invoked using Python code rather than a user interface. This is ideal for automating image manipulation tasks, or adding imaging features to your own applications.

In this chapter we will cover:

- Pillow versus PIL.
- How to install Pillow.
- A review of the main features of Pillow.

7.1 Pillow and PIL

Pillow is a *fork* of an older imaging library called PIL. This means that it is based the original code in PIL, but new features and bug fixes have been added over time.

At the time of writing, PIL itself hasn't been updated since 2009. A main motivation for Pillow was that PIL was not compatible with Python 3, and also PIL was not compatible with setuptools (meaning that it couldn't be installed easily using pip). In addition, PIL had known bugs that were not being fixed.

Since then, Pillow had also added new features of its own.

It now seems quite unlikely the PIL will be updated in the future, although it has never officially been declared dead. Pillow has become the *de facto* replacement for PIL.

One thing to note. Pillow uses the PIL namespace. You should not try to install PIL and Pillow on the same system.

7.2 Installing Pillow

Pillow can be installed using pip, as follows:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

This should work on Windows, macOS, and most flavours of Linux.

7.3 Main features of Pillow

Here are the main features of Pillow:

- Open, save, and convert between many different image formats.
- Perform a variety of image processing tasks automatically from Python code.
- Extract metadata, statistics, and other information from an image.
- Integrate with system features - image viewer, screen grab, printing.
- Integrate with various UI systems - Qt, Tk, Windows.
- Access pixel data efficiently, for implementing your own image processing.
- Convert between different pixel representations - colour spaces, colour depths, palette based images.
- Exchange data with NumPy and other Python libraries.
- Draw on images.
- Apply colour management.
- Handle image sequences.

8. Basic imaging

This chapter looks at basic image handling using the Pillow library. This includes:

- A first look at the `Image` class.
- How to create images.
- How to display images.
- Simple examples of image processing.
- Image modes

8.1 The Image class

The `Image` class is used by Pillow to store images. It is probably the most fundamental of all the classes in Pillow.

It is also quite a large class, with many different methods and associated functions. For that reason, we will cover it over several chapters.

This chapter will mainly look at:

- Some *factory* functions (function that create new `Image` objects in various ways).
- A few basic image processing functions.
- Some of the most important `Image` object methods.

The next chapter will cover some of the more specialised `Image` object methods.

8.2 Creating and displaying an image

So, let's get straight on and create an image with Pillow. Fortunately Pillow has a couple of features to help us get started quickly:

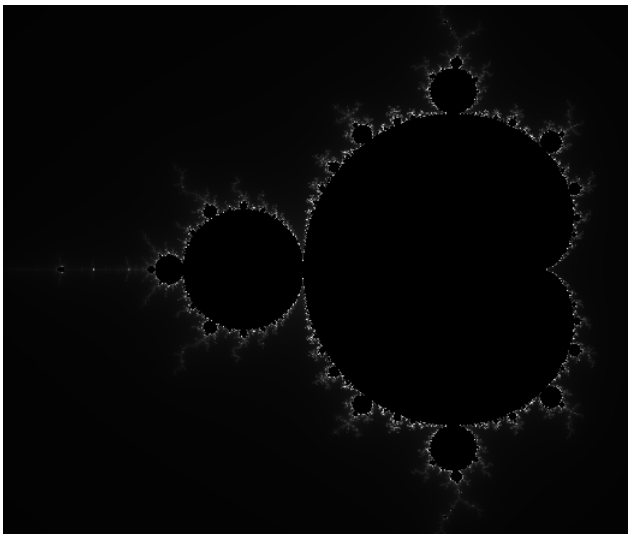
- A function that creates an image of the famous Mandelbrot Set. This just makes things slightly more interesting than a blank screen.
- A method that creates a window and displays an image.

Here is the necessary code:

```
from PIL import Image

image = Image.effect_mandelbrot((520, 440), (-2, -1.1, 0.6, 1.1), 256)
image.show()
```

And here is what it creates:



The `Image.effect_mandelbrot` function takes 3 parameters:

- The `size`. `(520, 440)` gives an image that is 520 by 440 pixels.
- The `extent`. `(-2, -1.1, 0.6, 1.1)` means that the image will display the Mandelbrot Set in the region `-2 < x < 0.6` and `-1.1 , y , 1.1`. This area covers the main part of the fractal.
- The `quality`. This sets the maximum number of times the algorithm loops to create the image. For a small image like this, 256 is fine.

If you are wondering why we have chosen the oddly specific image size of 520 by 440, it is because the Mandelbrot image covers a region that 2.6 by 2.2 units. The image pixel size should be the same shape as the selected region, to avoid the image being distorted. Both have the same aspect ratio of 13:11.

The function returns an `Image` object containing the image that has been created.

Calling `image.show()` causes the image to be displayed in a separate window. This is mainly intended for testing and debugging purposes - it saves you the hassle of saving the image to file and then opening the image in a viewer.

8.3 Saving an image

Of course, most of the time you will want to save your image. This is very easy with Pillow:

```
from PIL import Image

image = Image.effect_mandelbrot((520, 440), (-2, -1.1, 0.6, 1.1), 256)
image.save('mandelbrot.png')
```

The `save` method saves the image object using the supplied filename. The file type is controlled by the extension. For example ".png" to create a PNG file, or ".jpg" to create a JPEG file.

There are ways to exercise more control over the type of file you create, and we will cover them in a later chapter, but a lot of the time this simple method is good enough.

8.4 Handling colours

Pillow supports numerous colour spaces, including:

- RGB colour, which it stores as a tuple of 3 values in the range 0 to 255. This is equivalent to 24-bit RGB as described in the *Bitmap data* chapter. So, for example, a tuple of `(255, 0, 0)` represents pure red (that is, 100% red, 0% green and 0% blue).
- RGBA colour (RGB plus transparency). This is stored as a tuple of 4 values, where the fourth value represents transparency. This is equivalent to 32-bit RGBA.
- Grey colour, which is stored as a tuple of 1 value representing the grey value (equivalent to 8-bit greyscale).

It is sometimes easier to use a string representation of colour. The 'ImageColor' module can convert various string types to a colour tuple.

8.4.1 Converting strings to colours

You can use strings to represent colours. There are various formats, for example:

- `'#FF0080'` represents an RGB value of 100% red, 0% green, 50% blue.
- `'rgb(100%, 0%, 50%)'` also represents an RGB value of 100% red, 0% green, 50% blue.
- `'magenta'` represents an RGB value of 100% red, 0% green, 100% blue.

When you pass as string into a Pillow function that requires a colour, it will automatically be converted to a colour. See the *Reference* section for more information on the various supported formats.

8.5 Creating images

You can create a new, blank image like this:

```
from PIL import Image, ImageColor

image = Image.new('RGB', (400, 300), 'gold')
image.show()
```

`new` takes 3 parameters:

- `mode` - a string that determines the colour model and depth of the image. The string RGB creates a 24-but RGB image. We will cover other modes later in the book.
- `size` - a tuple giving the image size in pixels.
- `color` - the colour of the image. A new image is completely filled with a single colour. You can just use a tuple, or you can use a string value as we saw earlier.

Here is the image that we have created with a gold background colour:

8.6 Opening an image

You can also open an existing image from a file, like this:

```
from PIL import Image

image = Image.open('boat.jpg')
image.show()
```

This assumes you have an image called *boat.png* in your working folder. The image below in on github, along with the source files for this book, see the introduction for the link. Here is what you should see:

8.7 Image processing

Pillow provides may different ways to process images. Here are a couple of very simple examples.

8.8 Rotating an image

In this code we open an image and rotate it by 180 degrees:

```
from PIL import Image

image = Image.open('boat.jpg')
rotated = image.rotate(180)
rotated.show()
```

`rotate` is a method of the `image` object. It creates a new image that has been rotated. `rotate` takes 1 parameter:

- `angle` - the angle of rotation, in degrees.

`rotate` has some extra optional parameters that we will look at in a later chapter. Here is the result:

8.9 Creating a thumbnail

A thumbnail image is a small version of image, such as you might see in a file explorer.

The `thumbnail` method creates an image that fits within a specified maximum size:

```
from PIL import Image

image = Image.open('boat.jpg')
image.thumbnail((200, 200))
image.show()
```

`thumbnail` takes 1 parameter:

- `size` - tuple giving the maximum width and height of the image.

In this case the original image is 600 by 400 pixels. We have asked for the image to be scaled down to fit in within 200 by 200 pixels. `thumbnail` reduces the image to 200 by 133 pixels - that is the largest image that will fit the required size, but maintaining the original width to height ratio.

8.10 Image modes

In Pillow, the image *mode* describes both the colour space and the number of bits per pixel. It is represented by a string quantity.

The most common modes are:

- `'RGB'` - 24-bit RGB (8 bits per colour).
- `'RGBA'` - 32-bit RGB plus alpha (8 bits per colour and 8 bit alpha).
- `'L'` - 8-bit greyscale.
- `'1'` - bilevel data (each pixel is either fully black or fully white).
- `'P'` - each pixel is an 8-bit index into a palette that maps onto a colour of some other.

These are not the only modes. See the *Reference* chapter for more details.

9. Image class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.1 Example code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.2 Creating images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.2.1 Image.new

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.2.2 Image.open

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.2.3 copy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.2.4 Other methods

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.3 Saving images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.4 Image generators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5 Working with image bands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5.1 getbands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5.2 split

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5.3 merge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5.4 getchannel

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

9.5.5 putalpha

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10. ImageOps module

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1 Image resizing functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1.1 expand

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1.2 crop

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1.3 scale

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1.4 pad

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.1.5 fit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.2 Image transformation functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.2.1 flip

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.2.2 mirror

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.2.3 exif-transpose

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3 Colour effects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3.1 grayscale

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3.2 colorize

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3.3 invert

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3.4 posterize

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.3.5 solarize

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.4 Image adjustment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.4.1 autocontrast

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.4.2 equalize

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.5 Deforming images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.5.1 How deform works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.5.2 getmesh

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.5.3 A wave transform

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

10.5.4 Other deformations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11. Image attributes and statistics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1 Attributes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.1 File size

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.2 File name

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.3 File format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.4 Mode and bands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.5 Palette

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.6 Info

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.7 Animation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.1.8 EXIF tags

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.2 Image statistics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.2.1 Image histogram

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.2.2 Masking

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.2.3 Other Image statistics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

11.2.4 ImageStat module

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12. Enhancing and filtering images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.1 ImageEnhance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.1.1 Brightness

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.1.2 Contrast

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.1.3 Color

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.1.4 Sharpness

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.2 ImageFilter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.3 Predefined filters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.4 Parameterised filters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.4.1 Blurring functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.4.2 Unsharp masking

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.4.3 Ranking and averaging filters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

12.5 Defining your own filters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13. Image compositing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.1 Simple blending

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.1.1 Image transparency

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.1.2 ImageChops blend function

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.1.3 ImageChops composite function

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2 Blend modes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2.1 Addition

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2.2 Subtraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2.3 Lighter and darker

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2.4 Multiply and screen

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.2.5 Other blend modes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

13.3 Logical combinations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14. Drawing on images

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.1 Coordinate system

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.2 Drawing shapes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.2.1 Drawing rectangles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.2.2 Drawing other shapes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.2.3 Points

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.3 Handling text

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.3.1 Drawing simple text

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.3.2 Font and text metrics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.3.3 Anchoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.3.4 Drawing multiline text

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.4 Paths

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.4.1 Drawing a path

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.4.2 Transforming paths

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

14.4.3 Mapping points

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

15. Accessing pixel data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

15.1 Processing an image

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

15.2 Creating an image

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

15.3 Performance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16. Integrating Pillow with other libraries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16.1 NumPy integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16.1.1 Converting a Pillow image to Numpy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16.1.2 Image data in a NumPy array

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16.1.3 Modifying the NumPy image

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

16.1.4 Converting a NumPy array to a Pillow image

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

III Reference

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17. Pillow colour representation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.1 Hexadecimal colour specifiers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.2 RGB functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.3 HSL functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.4 HSV functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.5 Named colours

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

17.6 Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

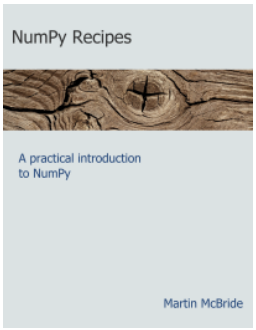
17.7 Image modes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/imageprocessinginpython.

18. More books from this author

I have several other Python books available. See https://pythoninformer.com/books/ for more details.

18.1 Numpy Recipes

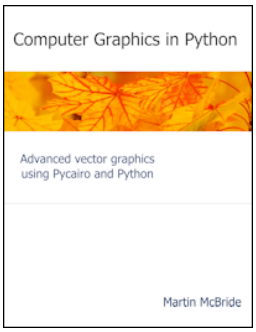


NumPy Recipes takes practical approach to the basics of NumPy

This book is primarily aimed at developers who have at least a small amount of Python experience, who wish to use the NumPy library for data analysis, machine learning, image or sound processing, or any other mathematical or scientific application. It only requires a basic understanding of Python programming.

Detailed examples show how to create arrays to optimise storage different types of information, and how to use universal functions, vectorisation, broadcasting and slicing to process data efficiently. Also contains an introduction to file i/o and data visualisation with Matplotlib.

18.2 Computer Graphics in Python with Pycairo



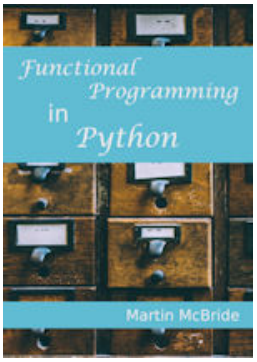
The Pycairo library is a Python graphics library. This book covers the library in detail, with lots of practical code examples.

PyCairo is an efficient, fully-featured, high-quality graphics library, with similar drawing capabilities to other vector libraries and languages such as SVG, PDF, HTML canvas and Java graphics.

Typical use cases include: standalone Python scripts to create an image, chart, or diagram; server-side image creation for the web (for example a graph of share prices that updates hourly); desktop applications, particularly those that involve interactive images or diagrams.

The power of Pycairo, with the expressiveness of Python, is also a great combination for making procedural images such as mathematical illustrations and generative art. It is also quite simple to generate image sequences that can be converted to video or animated gifs.

18.3 Functional Programming in Python



Python's best-kept secret is its built-in support for functional programming. Even better, it allows functional programming to be blended seamlessly with procedural and object-oriented coding styles. This book explains what functional programming is, how Python supports it, and how you can use it to write clean, efficient and reliable code.

The book covers the basics of functional programming including function objects, immutability, recursion, iterables, comprehensions and generators. It also covers more advanced topics such as closures, memoization, partial functions, currying, functors and monads. No prior knowledge of functional programming is required, just a working knowledge of Python.