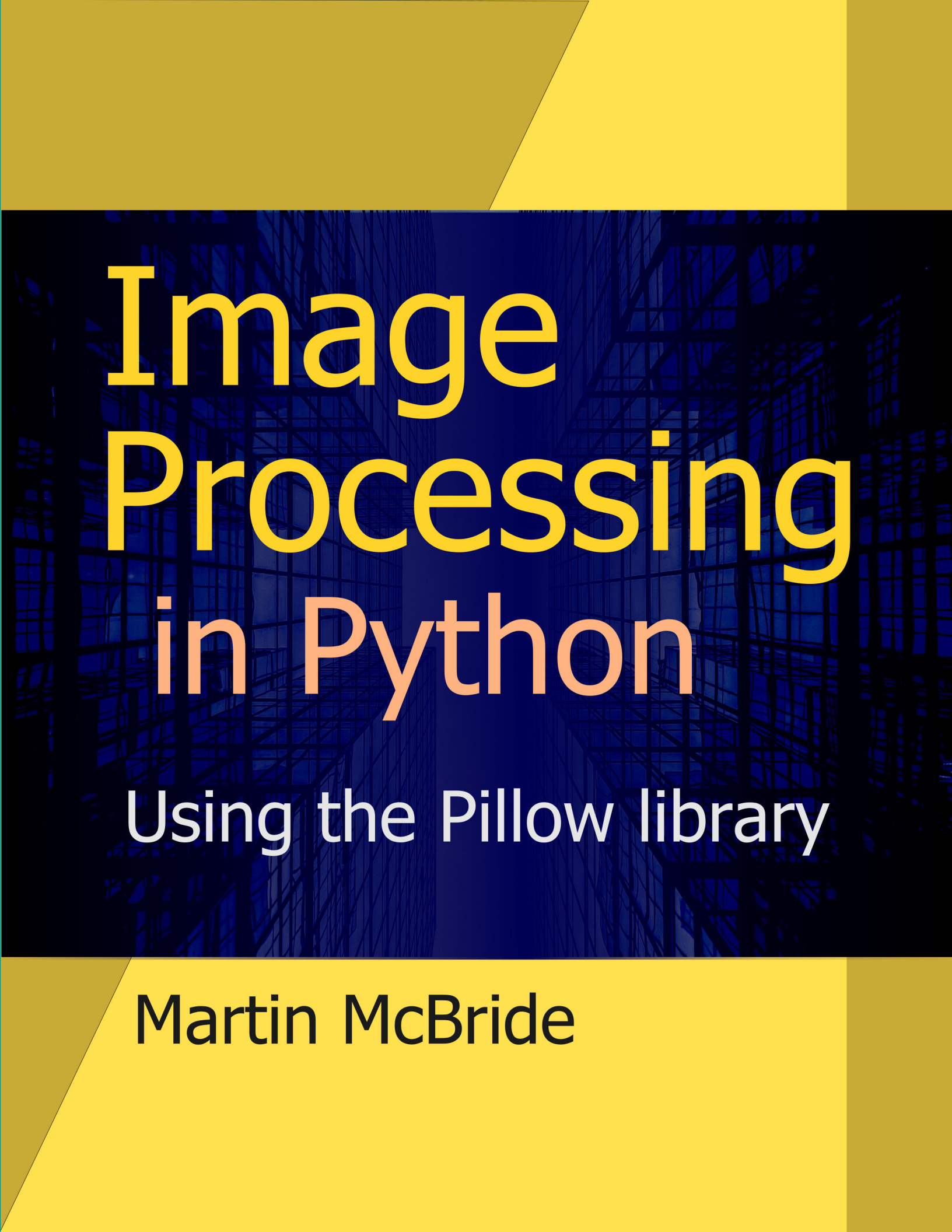


Image Processing in Python
Using the Pillow library
Martin McBride

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

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.

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.

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.

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/

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.

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.

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

1. 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.

1.1 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.

1.2 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.

1.3 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.

1.4 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.

1.5 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*.

2. 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.

2.1 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).

2.1.1 Frequency and wavelength

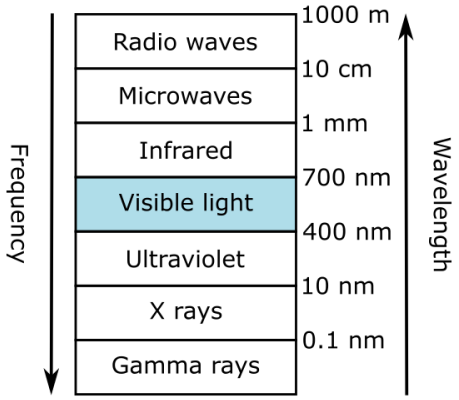
All electromagnetic radiation, including light, travels at a speed of 299 792 458 metres per second. That is the speed of light in a vacuum, often called c (light travels very slightly slower if it passes through materials such as air, glass or water, but the difference is a tiny fraction of 1%).

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.

2.2 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.

2.2.1 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

2.3 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.

2.4 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.

2.4.1 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.

2.4.2 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.

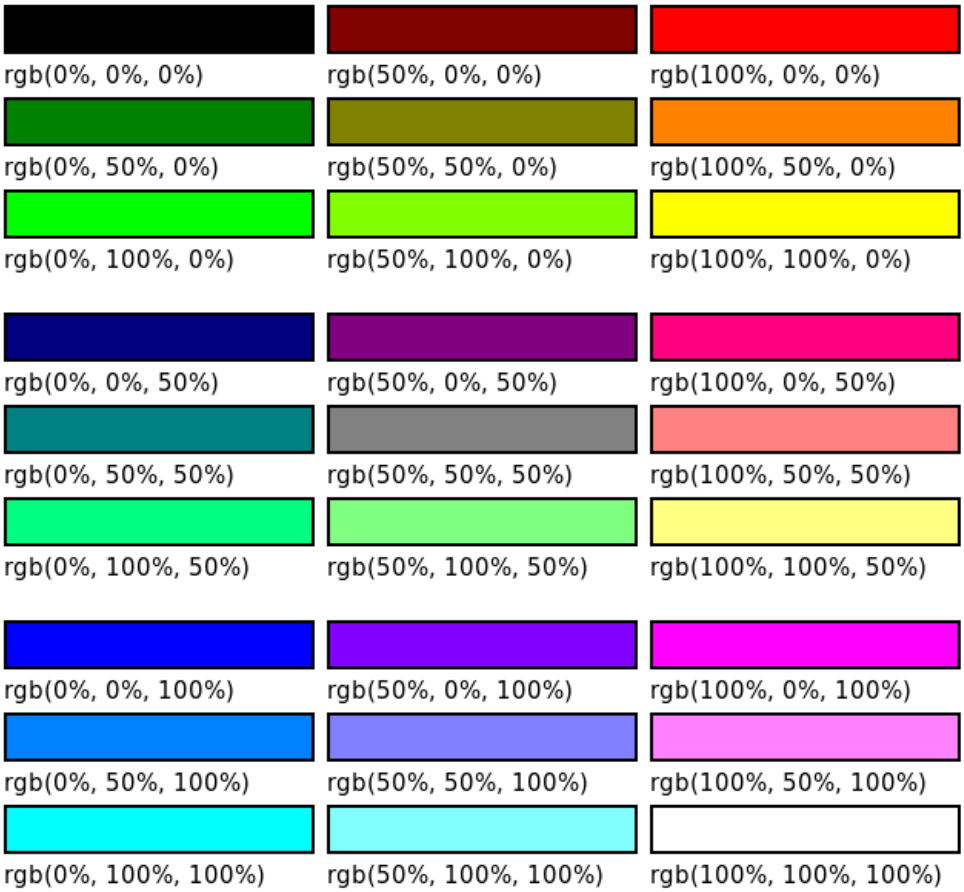
Similar for green and blue.

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%.

2.4.3 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.

2.4.4 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%)

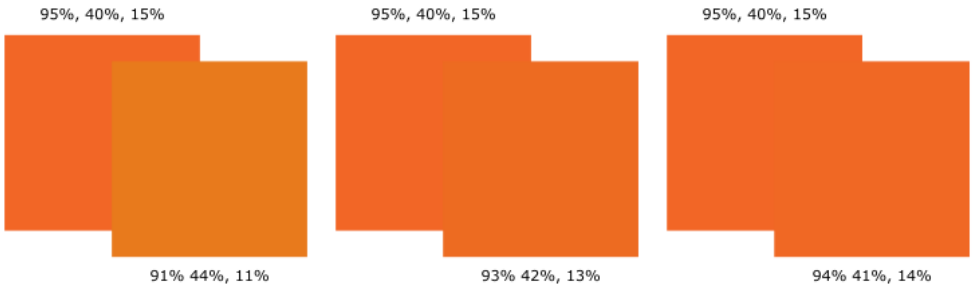
would be represented by three integer values `(255, 128, 0)`.

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.

2.5 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.

2.6 Greyscale colour model

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

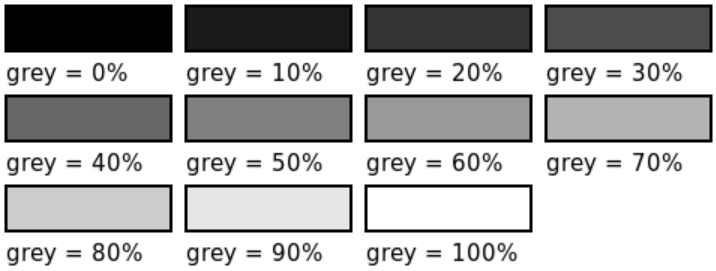
- rgb(0%, 0%, 0%) is black.
- rgb(30%, 30%, 30%) is dark grey.
- rgb(100%, 100%, 100%) is white.

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.

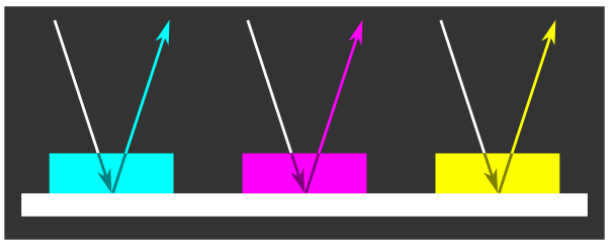
2.7 The CMYK colour model

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:

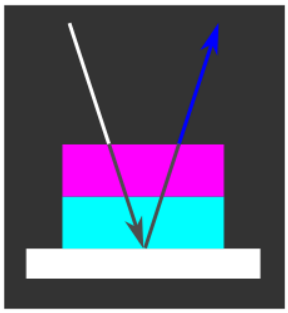


Because the ink is translucent, light passes through the ink and reflects off the white paper underneath. The ink acts as a colour filter, removing certain colours.

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.

2.7.1 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.

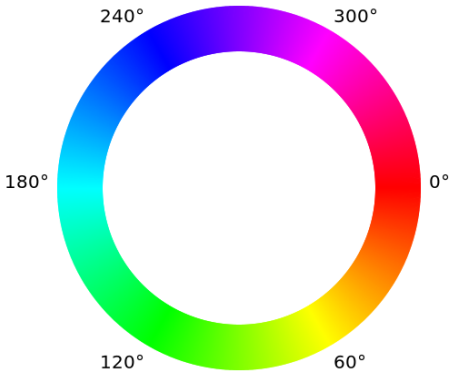
Almost all colour printing processes - from home or office printer, through to newspapers or magazines printing presses - uses a separate black channel, for these reasons. In some cases the printer also adds black ink into darker coloured areas, to reduce the amount of coloured ink that is needed.

2.8 HSL/HSB colour models

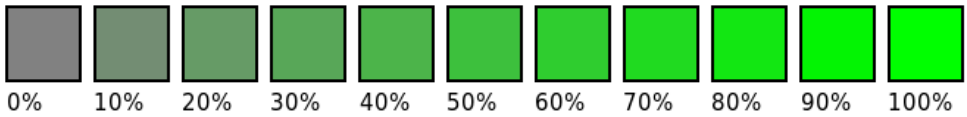
The HSL colour space stores colours as 3 values:

- 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.

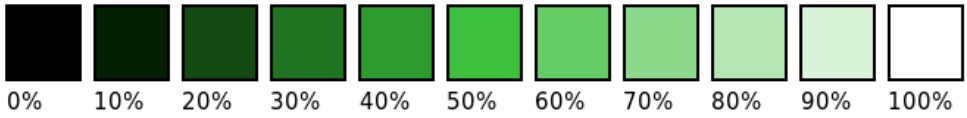
This diagram shows the colour wheel of hue values. The hue can be expressed as an angle between 0 and 360 degrees, where 0 is red, 120 is green, and 240 is blue:



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.

2.9 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.

2.10 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.

2.10.1 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.

2.11 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.

2.11.1 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.

3. 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.

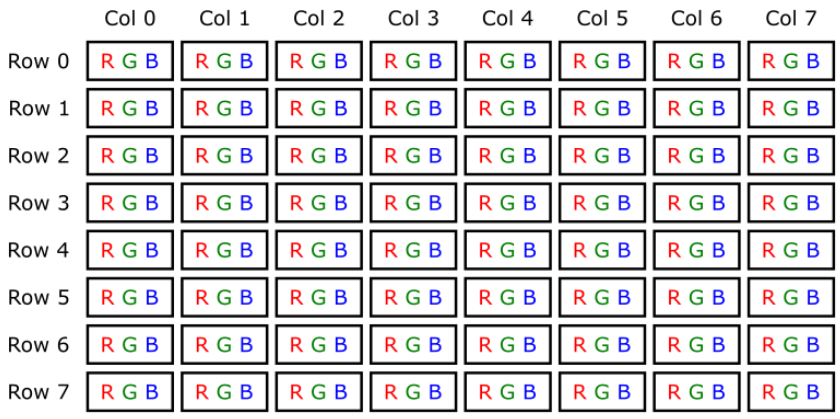
A bitmap image consists of a two-dimensional array of pixels, each with its own colour. The bitmap data for that image consists of an array of colour values, one for each pixel. In this chapter, we will look at how bitmap data is stored.

We will cover:

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

3.1 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.

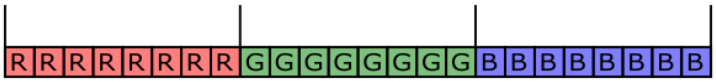
When bitmap data is stored in an image file, it often uses a similar format.

3.2 8-bit per channel images

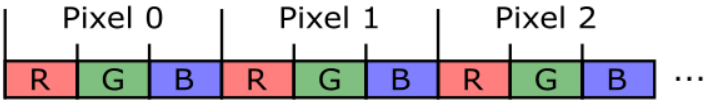
Most bitmaps are stored using 8 bits (ie 1 byte) per colour per pixel. This applies to various colour models.

3.2.1 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:



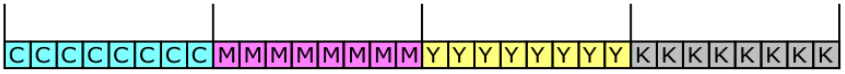
This is a very convenient format for two reasons:

- 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.

3.2.2 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:



3.2.3 8-bit greyscale

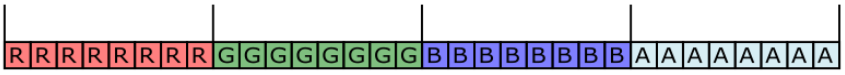
For greyscale data, there is only one channel, so each pixel occupies a single byte:

3.2.4 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.



3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.4 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.

3.5 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.

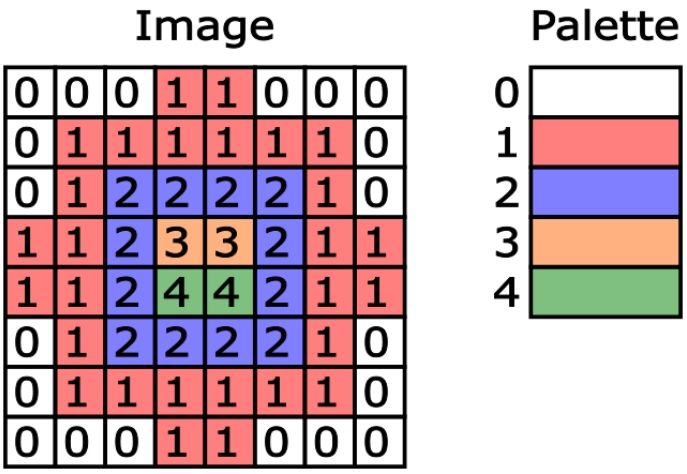
This data is often stored in the following formats:

- 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.

3.6 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.

3.6.1 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.

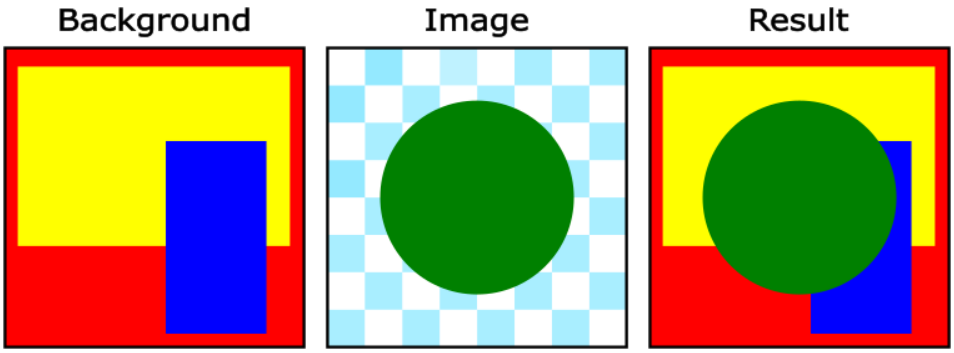
A second method is to use an *adaptive palette*. Before the image is stored, the imaging software analyses the image to see which colours are actually used. It then chooses 256 colours that best represent the image.

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.

3.7 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.

3.7.1 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).

3.7.2 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.

3.7.3 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.

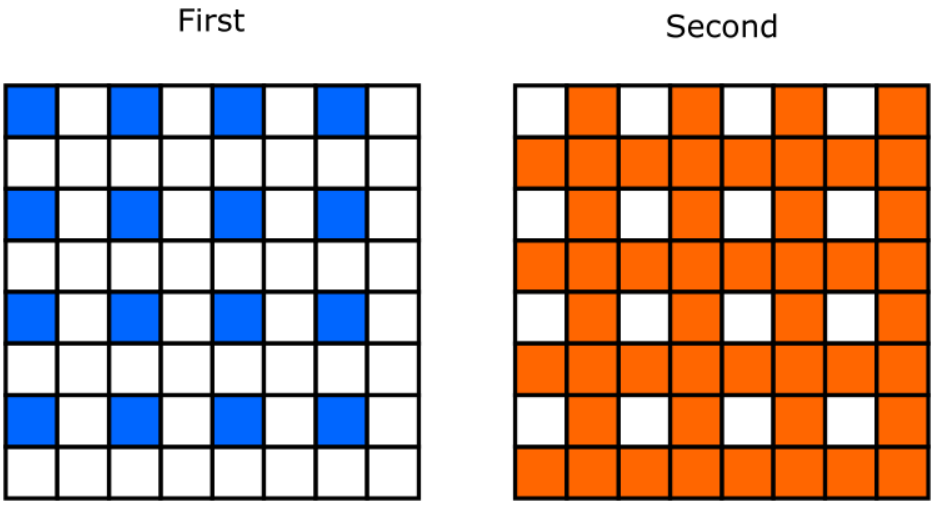
This isn't a technique that is used by any popular image formats, it is really quite hacky. It is mainly used in proprietary image formats.

3.8 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.

4. Image file formats

It is useful to store individual bitmap images as separate files. In this chapter we will look at various aspects of some common image file formats:

- Why are there so many formats?
- Image data and metadata.
- Image compression.
- Common image formats.
- Animation and video.

4.1 Why are there so many formats?

There are a few dozen different image formats that are reasonably well known, and probably hundreds of other formats that are less well known. Why so many?

Many were developed in the days before internet use became widespread and before digital cameras were popular and affordable. At that time, using standard image formats wasn't such a priority because it was comparatively rare for people to exchange images. Every operating system supplier, industry body, or company that needed to store image data was likely to invent their own format rather than using one that already existed.

Many of these formats were superfluous - an existing format could have been used instead, at least technically. There were some good reasons for not using existing formats, for example, some of the formats were proprietary, or even secret, and a company might not want to become dependent on a standard that belonged to a competitor. There were also bad reasons, sometimes software engineers just like to invent their own version of something for the fun of it.

However, there is unlikely to ever be one format to rule them all. Different applications have different requirements, and it is difficult to have a single format that works for all cases. Here are the main genuine reasons different formats are required:

- Efficient lossless compression of logos and diagrams. For example, web images. PNG commonly satisfies this requirement.
- Efficient compression of photographic images, where lossy compression is acceptable. For example, storing photographs or web images. JPEG commonly satisfies this requirement.
- A single format that can store many different types of image data, with different characteristics. For example, newspaper/magazine printing where images might be coming from different sources. TIFF is often used in these cases.

- A really simple format, for storing small images, that can be implemented quickly and easily on low-performance hardware (such as an embedded system). Windows BMP format is an example.
- There are also various formats with unique special features. An example is GIF format, which has the ability to encode animations in a relatively simple file format.

These days, there are half a dozen formats that are very widely used, as we will see later in this chapter.

4.2 Image data and metadata

Image formats generally contain image data and image metadata.

Image data specifies the colour of every pixel in the image. We covered that in the chapter on *Bitmap image data*.

Metadata contains information *about* the image. This is usually stored in a format specific way. There are also some general metadata formats that can be used by different image types, such as EXIF.

Here is the basic information that most image formats provide:

- Image size in pixels.
- Image resolution, usually in pixels per inch or pixels per cm. This can be used to determine the intended physical size of the image when it is printed. For example, if an image has a size of 1800 by 1200 pixels, and a resolution of 300 pixels per inch, it means that it is intended to be printed at a size of 6 by 4 inches (1800/300 and 1200/300). Of course, that doesn't prevent it from being printed at a different size, it is only an indicator.
- The *mode* of the image, which includes the colour space (such as RGB), the number of bits per pixel, and whether the image uses a palette.
- The layout of the data, for example, whether it stored as a sequence of RGB values or three separate planes of R values, G values then B values.
- Parameters relating to the type of image compression used.
- The image palette, if there is one.
- Special information, such as the number of frames in an animated image.

Not every format includes all this information, because sometimes it is implicit. The information is typically stored in an image *header*, which is a data block that precedes the image data, but the format of the header is different for each type of image.

Images often contain additional metadata:

- Date and time information. This is added when the image is created. Some cameras contain GPS hardware and can automatically add the exact location of the photograph.

- A small thumbnail image, which can be displayed without needing to read in the entire image.
- Camera settings. Perhaps the most useful value here is the camera orientation, as this can be used to automatically correct images that have been taken with the camera turned on its side. But cameras often return detailed information about every setting (aperture, shutter speed etc) for each image.
- Description of the image.
- Copyright information for the image.

This information is often encoded using EXIF (Exchangeable Image File Format). This can be attached to JPEG and TIFF files. Many imaging applications allow you to read, edit, or delete EXIF information.

4.3 Image compression

Image data can often be several megabytes in size, or even larger for very high-resolution images. Even with modern disk sizes and network speeds, there are definite benefits to compressing this data. Many images can be compressed to a quarter their original size without any visible loss of quality, but it is not uncommon for an image to be compressed by a factor of 20 or better if a moderate loss of quality is acceptable. This very much depends on the image itself, and the type of compression used.

There are two main classes of image compression, lossless and lossy.

Lossless compression aims to make the image data smaller, but without changing the image in any way. That is to say, if you compress an image and then uncompress it, the resulting image will be identical to the original, every pixel will have exactly the same colour. It works by finding any natural redundancy in the image and encoding it more efficiently.

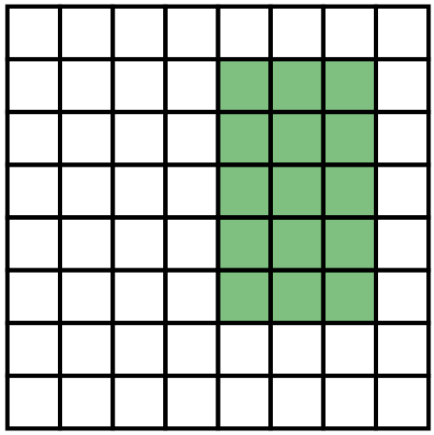
Lossy compression works differently. It will make changes to the image data, to allow it to be coded more efficiently. This means that if you compress an image and then uncompress it, the result will be different to the original. That isn't quite as bad as it sounds because, in most cases, the differences will be so slight that you won't notice. For example, if you compressed a photograph of a tree, the fine details of some of the leaves might change slightly, but they would still look like leaves so there will be no glaring faults with the image.

We won't go into compression algorithms in great detail, it is a very complex subject, and modern image formats like JPEG and PNG provide very efficient algorithms out if the box. It is useful to understand the characteristics of compression algorithms, but not necessary to understand exactly how they work.

4.3.1 Lossless compression

Lossless compression aims to reduce the size of the data by storing it more efficiently, without changing the pixel values themselves. we will look at a couple of popular methods here - run-length coding and Zip style compression.

Here is an example image where each square represents a single pixel. The image is only 8 pixels square to make the description a bit easier:



We could store this image simply by storing the colour of each pixel. So the first two lines of the image would be:

```
white white white white white white white white
white white white white green green green white
```

And so on. Each colour will be stored as a three-byte RGB value. That is fine, but a little repetitive.

Alternatively, we could store the information as a colour plus a count of how many pixels there are with that colour:

```
white:8
white:5 green:3 white:1
```

This tells us that the first line is just 8 white pixels. The second line is 5 white pixels followed by 3 green then 1 white.

This is called run-length coding. We would need to come up with some format for storing the lengths and colours, but it should be quite obvious that we can create a format that is much smaller than the original data, provided the data itself has plenty of long runs of the same colour.

This compression method is simple but very effective for images like diagrams or charts. Quite a lot of image formats use it.

The PNG format uses a slightly different method, called DEFLATE (based on LZ77). This works similarly to ZIP compression. It achieves better results than run-length coding because it is capable of exploiting other repetitive patterns, in addition to straight runs of the same colour.

GIF format also uses a different variant from the same family of algorithms.

Overall these methods work very well on images with blocks of identical colour or other predictable patterns. They work less well with photographic images where the pixel values do not follow regular patterns.

4.3.2 Lossy compression

Lossy compression involves throwing away information to reduce the image size. The main example of a lossy compression algorithm is JPEG.

The algorithm is complex so the description here is greatly simplified. In essence, there are three stages.

First, the image is divided into blocks, usually 8 by 8 pixels. Each block is compressed separately.

Next, the block is transformed into the 2D frequency domain. What does that mean? You may be familiar with the frequency displays you often see on fancy audio systems or recording equipment. As the music plays, the display shows you the amount of low, mid and high-frequency sound there is in the signal.

A frequency transform on an image block is a similar idea, but it works of spatial frequency rather than audio frequency. And it works in two dimensions. There are still 8 by 8 values in the transformed block, but instead of each value representing the colour of a single pixel, each value represents the amount of variation at a particular spatial frequency..

We can transform an image block into the frequency domain, then transform it back into the spatial domain, and we will get the same image back. But that alone doesn't provide any compression.

The third step is to *quantise* the values in the frequency domain. Our eyes aren't very sensitive to high frequency variation, so we can encode it less accurately. So for the higher frequencies we don't need to store a value with range 0 to 255. We could, for example, encode it as a value 0 to 3 (represent the 4 values 0, 85, 190, 255). We round the actual value to the nearest quantised value, so we can store it as a 2 bit quantity rather than an 8 bit quantity.

Under this scheme it is common for quite a lot of the values to round down to zero. A block of zeros can be encoded very efficiently, giving a very good compression ratio.

When we decompress this data, the individual pixel value will be different, but the overall character of the block will be similar.

It is also possible to vary the amount of quantisation. The more quantisation we apply, the better the compression (but at the cost of less accuracy).

4.4 Some common file formats

In this section, we will look at some common file formats and their capabilities. There are many other formats, but these are the most widely used.

4.4.1 PNG format

PNG (Portable Network Graphics) was introduced in 1996, mainly aimed at providing a suitable modern image format for the internet.

It supports the following colour modes:

- RGB, using 24 or 48 bits per pixel (ie 8 or 16 bits per channel).
- RGBA, using 32 or 64 bits per pixel (ie 8 or 16 bits per channel).
- Greyscale, using 1, 2, 4, 8 or 16 bits per pixel.
- Greyscale and alpha, using 16 or 32 bits per pixel (ie 8 or 16 bits per channel).
- Palette based RGB.
- Palette based RGBA.

For palette-based images, the colours in the palette are always stored as 8 bits per channel. The palette can contain up to 256 colours.

PNG uses a variant of the compression scheme used by Zip, which provides reasonable lossless compression for most images. It is particularly suited to artificial images containing regions of flat colour, such as diagrams, charts, text and similar. It will also compress photographs, but it does not perform quite as well as JPEG.

PNG optionally allows interleaving, using the Adam7 algorithm, as described in the *Bitmap data* chapter.

4.4.2 JPEG format

JPEG (named after the group that created it, Joint Photographic Experts Group) was introduced in 1992. With the introduction of digital cameras, a format was required which compressed digital photographs very efficiently.

It supports the following colour modes:

- RGB using 24 bits per pixel (ie 8 bits per channel).
- Greyscale and CMYK images are possible, but many applications don't support this.
- 12 bits per channel is possible, but many applications don't support this.
- Does **not** support transparency.
- Does **not** support less than 8 bits per channel.

JPEG uses lossy compression. The image you get back isn't precisely the same as the original, it has some blurring and noise, but it is generally difficult to spot if the image is a photograph of a natural scene. It has quality settings, which can make the compressed file smaller, at the expense of image quality.

JPEG distorts hard edges quite badly, so it isn't suitable for images of diagrams or text. It is generally better to use PNG for those types of image.

4.4.3 GIF format

GIF (Graphics Interchange Format) was the first image format to be widely supported by the World Wide Web. The Web needed an image format that was independent and offered good compression. GIF seemed to be the best solution, so most browsers added support for it. GIF also allowed for transparency and simple animation.

It supports the following colour modes:

- Palette based RGB, allowing a maximum of 256 separate colours in the image. Each individual colour is 24-bit RGB.
- Greyscale, by using only grey values in the palette.
- Transparency, by specifying a transparent palette entry.

Despite its initial popularity, the fact that a GIF can only display 256 different colours means GIF images are often low quality. Additionally, GIF pixels can either be totally transparent or totally opaque. This doesn't allow for graphics that fade out at the edges, which again gives a poor quality result. PNG and JPEG are generally better choices.

The one unique feature of GIF images is that they can store multiple images that can be played back as an animation. This makes them useful for creating animated logos and even short video clips.

4.4.4 BMP format

Windows BMP (Bitmap) format is a very simple format for storing RGB or Monochrome images.

It supports the following colour spaces:

- RGB, using 24 bits per pixel (ie 8 per channel).
- RGBA, using 16 or 32 per pixel (ie 4 or 8 bits per channel).
- Palette based RGB.
- Greyscale, by using only grey values in the palette.

Images can be uncompressed, or compressed using a form of runlength compression.

BMP was the default file format for Microsoft Windows images, but in many cases PNG is now used instead.

4.5 Animation

GIF provides simple animation. The only difference between a GIF animation and any other image is that:

- A GIF animation contains a sequence of images.
- It also specifies the time delay between displaying the images.

When a browser or image viewer opens a GIF (assuming the viewer supports animation), it will show the images one after another. If the delay is quite long (half a second or more) the images will appear one after the other like a slide show. If the delay between images is short (a tenth of a second) it will look like a slightly jerky video.

GIF is still the only widely supported image format that provides animation. Variants of PNG format (called MPNG and APNG) do something similar, but GIF animation has been around for a long time and is supported on every browser and many image viewers, so it is still the most popular way of creating simple animations for the web. There are also a lot more tools available for creating animated GIFs than animated PNGs.

Animated GIFs are not generally as good quality as actual video. Video requires at least 24 good quality images per second. Due to the sheer amount of data, they need to use far more sophisticated compression than image formats. They typically use compression schemes that work across multiple frames, taking advantage of the fact that successive frames are often very similar, to achieve far greater compression ratios.

II Pillow library

5. 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.

5.1 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.

5.2 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.

5.3 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.

6. 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

6.1 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.

6.2 Creating and displaying an image

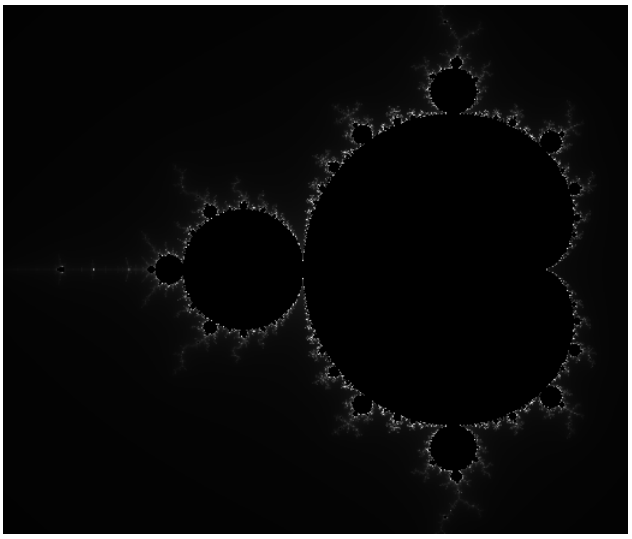
So, let's get straight on and create an image with Pillow. Fortunately Pillow has a couple of features to help us get started quickly:

- 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.

6.3 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.

6.4 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.

6.4.1 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.

6.5 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:

6.6 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:

6.7 Image processing

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

6.8 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:

6.9 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.

6.10 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.

7. Image class

The Image module is a core part of the Pillow library. It is the main class that is used to store images of all types, and it also contains a large number of functions methods for processing images in various ways.

In this chapter we will look in detail at several important areas of functionality provided by the Image class:

- Creating images.
- Saving images.
- Image generators.
- Working with image bands.

Some other key parts of the Image module will be covered in separate chapters later in the book:

- Image properties and statistics.
- Pixel access - how to read and write pixels efficiently.
- Integration with other libraries - how to exchange image data efficiently.

The Image module also contains some functions that overlap with other modules. Again these will be covered in later chapters:

- Image operations (rotate, transpose etc) - see the *ImageOps module* chapter.
- Image compositing (blending etc) - see the *Image compositing* chapter.
- Image filtering (filter function) - see the *Enhancing and filtering images* chapter.

The Image module also contains functions for adding extensions, for example to handle special encoding schemes in certain image formats. That is a very specialised area that we won't be covering in this book, not least because extensions are usually written in C and this is a Python book.

7.1 Example code

For brevity, in the examples below we will assume the variable `image` has already been initialised with image data (for example, by reading in an image file).

7.2 Creating images

An image in Pillow is represented as an `Image` object. There are various ways to create image objects.

You would never normally need to create an Image object directly, you should use one of the functions below to create an Image that is correctly initialised with data.

7.2.1 Image.new

`Image.new` creates a new image with a given mode, size, and colour. We used this in the previous chapter.

7.2.2 Image.open

`Image.open` reads an image in from a file, again as we saw in the previous chapter. `open` also accepts a `formats` parameter:

```
boat_image = Image.open("boat-small.jpg", formats=['PNG', 'JPEG'])
```

`open` will normally try to open any image file, in any supported format, but if you supply a list of formats it will only attempt to open images in those formats. This is useful if you want to avoid accidentally opening some obscure format that might have features your application doesn't support. In the above case, we will only open PNG and JPEG files, so it will work because we have supplied a JPEG file.

`open` determines the file type by examining the file. It pays no attention to the file extension. So, for example, if a PNG file had been accidentally saved with a .jpg extension, the code above would open the PNG file without error.

This code only allows JPEG images:

```
boat_image = Image.open("boat-small.jpg", formats=['JPEG'])
```

This code will open the file if it really is a JPEG file, but will raise an error if the file is in a different format.

Rather than a file name, you can also supply a file object that has been opened in read mode.

7.2.3 copy

The `copy` method creates a new copy of an image:

```
image2 = image.copy()
```

`image` is an existing Image object (for example read in from a file). `image2` will be a complete copy of the original image. You can then change `image2` without affecting `image`.

7.2.4 Other methods

You can create an image from data, in the form of a memory array (such as a NumPy array), you can use the `fromarray`, `frombuffer`, or `frombytes` methods to create an image from the data. This is covered in the chapter *Integration with other libraries*.

7.3 Saving images

You can save an image using the `save` method, like this:

```
image.save('boat.png')
```

The function will use the file extension to decide which file format to use, so in the example above it will create a PNG file.

For more certainty, you can specify a format:

```
image.save('boat.png', format=PNG)
```

This will store the image in PNG format regardless of the file extension. This is useful if you wish to store the image with a non-standard extension (eg .dat rather than .png).

Rather than a file name, you can also supply a file object that has been opened in write mode. In that case tyou should supply a `format` parameter because Pillow won't know the destination file name.

7.4 Image generators

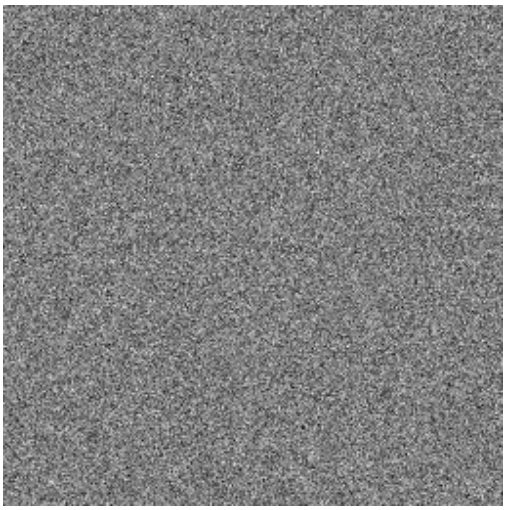
Pillow comes with a slightly random set of image generator functions.

We looked at the `effect_mandelbrot` function in the previous chapter. It is a good way to quickly generate a non-boring image for test purposes.

`effect_noise` generates a noise image:

```
noise = Image.effect_noise(size=(256, 256), sigma=32)
```

The image size is set to 256 by 256 pixels, and the image is filled with *Gaussian noise*, which sets each pixel to a random value between 0 and 255 according to a Gaussian distribution, centred on the value 128. The `sigma` value controls how spread out the values are, 32 is a reasonable value for the sake of illustration. Here is the result:



`linear_gradient` creates an image with a gradient that goes from 0 to 255 (black to white). The image size is fixed at 256 pixel square, so the only parameter is the mode. It must be a greyscale mode, so the obvious choice is `'L'`. You can also use `'P'` to create a pallet based greyscale image:

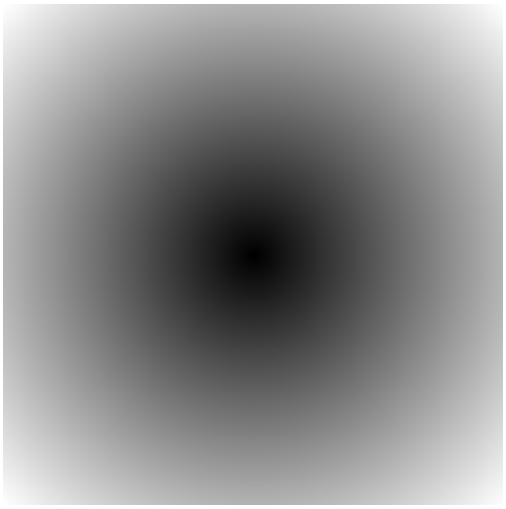
```
image = Image.linear_gradient('L')
```

Here is the result:

A similar function is available for creating a radial gradient:

```
image = Image.radial_gradient('L')
```

And again, here is the result:



7.5 Working with image bands

In an RGB image, each pixel contains a red, green, and blue component. In Pillow, these components are called *bands*.

Pillow contains various functions for working with bands, which we will look at in this section.

7.5.1 getbands

The `getbands` method returns a tuple containing the single-letter names of each band in the image:

```
bands = image.getbands()
```

For an RGB image, the function return `('R', 'G', 'B')`. For an RGBA image it returns `('R', 'G', 'B', 'A')`, and so on.

7.5.2 split

The `split` method separates an image into multiple images, with each image containing a single band of the original image:

```
red_image, green_image, blue_image = image.split()
```

In this case, the original image is split into 3 components because it is an RGB image. `red_image` is a greyscale image that contains just the red component of the image, similar for `green_image` and `blue_image`. Here is the red image:

In this image, dark areas are parts of the image that contain very little red, bright areas are parts of the image that contain a lot of red.

7.5.3 merge

The `merge` function does the opposite of `split`. It takes several single band images and merges them to create a colour image:

```
mixed_image = Image.merge('RGB', [red_image, green_image, blue_image])
```

The function takes an image mode (`'RGB'` in this case) and a sequence of single-band images. The number of band images must match the number of bands in the mode, that is 3 in the case of an RGB image.

The code above merges the three bands we created earlier, so it will result in an image that is identical to the original.

Here is something a bit more interesting:

```
blank = Image.new('L', image.size)
red_sep = Image.merge('RGB', [red_image, blank, blank])
red_sep.save('red_sep.png')
```

In this case, `blank` is an empty band image (a greyscale image, the same size as the original image but filled with black).

If we merge `red_image` with two blank images, we will get an RGB image where:

- The red band is the same as `image`.

- The green and blue bands are empty.

So it shows just the red band, but coloured red:

7.5.4 getchannel

`getchannel` is an alternative to `split` if you only want one band:

```
red_image = image.getchannel('R')
```

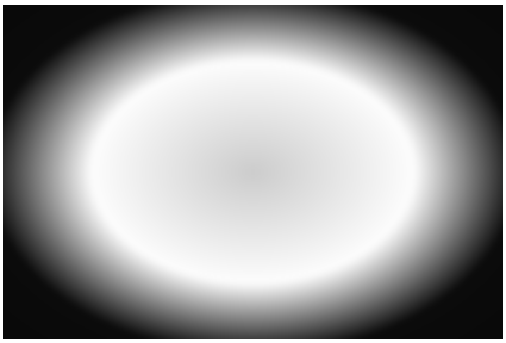
This extracts the red band from the image, same as `split`, but it doesn't extract green or blue.

Bands can be identified by name (such as `'R'`, `'G'`, `'B'`, as they appear in `getbands`) or index number (0, 1, 2…).

7.5.5 putalpha

The `putalpha` method can be used to add an alpha band to an image (or replace the current alpha band if there is one already). It is a good way to add transparency to an existing image.

In this example we will use a vignette (a radial gradient) as our alpha channel:



Here is the code to apply this image as an alpha channel to out boat image:

```
vignette_image = Image.open('vignette.png').getchannel(0)
image.putalpha(vignette_image)
```

The vignette image is an RGB image, but we need a single band image for `putalpha`. That is no problem, we can simply use `getchannel(0)` to extract the first band.

`putalpha` adds the vignette as the alpha channel to `image`. This converts `image` from RGB to RGBA. Here is the resultant image:

The image fades out around the edges - that is because it is increasingly transparent, so the white page shows through.

8. ImageOps module

The `ImageOps` module contains a collection of functions that perform various common image operations in a single call. These include some of the most common things you might typically do with an imaging application such as GIMP or Photoshop.

Operations fall into several groups:

- Image resizing - expand, crop, scale, pad, fit.
- Image transformation - flip, mirror, exif_transpose.
- Colour effects - colorize, grayscale, invert, posterize, solarize.
- Colour adjustment - autocontrast, equalize.
- Image deforming - deform.

This module is described as "experimental" in the Pillow documentation. It is pretty reliable, although it should only be used with 24-bit RGB and 8-bit greyscale images. There are other ways of performing these operations using standard functions, but that would usually require multiple function calls, so this module is worth knowing about.

We will use a smaller version of our *boat* image, 420 by 280 pixels, to illustrate these effects:

8.1 Image resizing functions

These functions all resize the image, in various ways.

8.1.1 expand

The `expand` function adds a solid colour border around the image, like this:

In this case, we have added a yellow border that is 40 pixels wide. This means that the width and height of the new image are both increased by 80 pixels, making it 500 by 360 pixels.

The image content is not rescaled by this function.

`expand` accepts a single parameter for the border width, so the four borders will always have equal widths.

The image inside the borders is pixel-for-pixel identical to the original.

Here is the code:

```
from PIL import Image, ImageOps

image = Image.open('boat-small.jpg')
result_image = ImageOps.expand(image, 40, 'yellow')
result_image.save('imageops-expand-40.jpg')
```

`expand` has the following signature:

```
ImageOps.expand(image, border=0, fill=0)  # returns a new image
```

- `image` is the original image.
- `border` gives the width of the border in pixels, and the same width is used on all 4 sides of the image.
- `fill` gives the colour of the border (defaults to black if not supplied)

All the examples in this chapter have similar import, open and save code. To avoid repetition for the remaining examples we will just show the `ImageOps` function call.

8.1.2 crop

`crop` removes a band of pixels from each of the 4 sides of the image, leaving just the central part, like this:

In this case, we have removed a 20-pixel band from each edge. This means that the width and height of the new image are both decreased by 40 pixels (380 by 240 pixels).

This diagram shows the original image, and the red rectangle indicates the boundary that is cropped:

`crop` accepts a single parameter for the crop width, so the top, bottom, left and right will all be cropped by the same amount.

The remaining image corresponds to the central part of the original image.

Here is the code:

```
result_image = ImageOps.crop(image, 20)
```

`crop` has the following signature:

```
ImageOps.crop(image, border=0)  # returns a new image
```

- `image` is the original image.
- `border` gives the width of the area to be removed, in pixels. The same width is used on all 4 sides of the image.

You can think of `expand` and `crop` and being almost like opposites, `expand` adds a band of pixels around the edge of an image, `crop` removes them. If you expand an image then crop it by the same amount you will end up with the original image.

8.1.3 scale

`scale` increases or decreases the size of the image by a scale factor. Here is an example of scaling the image by a factor of 2. The new image has twice the width and twice the height, and the image content is scaled to match:

Here is the code:

```
result_image = ImageOps.scale(image, 2)
```

Here is another example, again using the original image, with a scale factor of 0.5. This creates an image that is half the width and height:

The code is the same as the previous example, but with a scale factor of 0.5 passed into the `scale` function.

`scale` has the following signature:

```
ImageOps.scale(image, factor, resample=3)  # returns a new image
```

- `image` is the original image.
- `factor` is the scaling factor to be applied.
- `resample` specifies the resampling method to be applied, as described below.

Whenever we scale, rotate or apply other transformations to a pixel image, problems such as jagged edges and other unwanted effects can occur. We can apply a *filter* to reduce these effects. By default,

Pillow uses a *bicubic* filter, which usually does a pretty good job. The `resample` parameter allows us to choose one of several other filters instead. See the *Pillow resampling filters* chapter for more details.

When you increase the size of an image by scaling you are adding new pixels, but of course, you can't magically create extra pixel data. Duplicating pixels creates a blocky effect. Applying a filter smooths out the blockiness and creates a blurred effect instead. Either way, a resized image is never going to have the same quality as an image that was originally created at the larger size.

8.1.4 pad

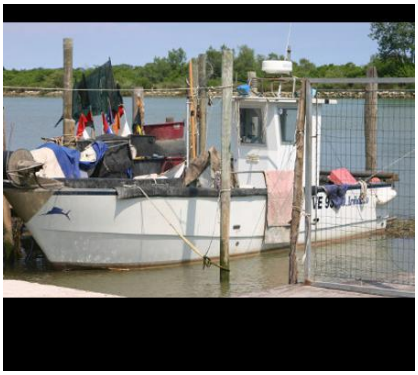
The `pad` function allows you to change the shape of an image, without stretching or distorting the image. It does this by scaling the image so it fits the required size, and then adding padding to the image as necessary.

Specifically, `pad` scales the image to be as big as it can be while still fitting in the target size. Depending on the image size and the target size it might add padding to the top and bottom of the image, or add padding to the left and right of the image.

Here is the first example. Our original image is 420 by 280 pixels, and we want to pad it to make it 450 by 400 pixels. Here is the code to do that:

```
result_image = ImageOps.pad(image, (450, 400), centering=(0.5, 0.2))
```

And here is the result:



The target size has a *taller aspect ratio* than the original image, so `pad` does the following:

- First it scales the image to fit the required width. This makes an image that is 450 by 300 pixels in size.
- The image is the correct width, but it isn't tall enough, so the image is padded (with black pixels) to make it 400 pixels high.

By default, `pad` would add 50 pixels at the top and bottom of the image, so the image is centred in the area. However we have provides a `centering` value of `(0.5, 0.2)` to adjust this.

The `centering` parameter has a value `(x, y)`, which can be a little confusing. In this case, we are centring the image in the y-direction so the `x` value is completely ignored. The value, 0.2, means that 20% of the extra space is added above the image and 80% below. That is why the top band of black is smaller than the bottom band in the image above.

In this second example, the same original image is padded to a size of 250 by 400 pixels:

```
result_image = ImageOps.pad(image, (400, 200), color='tomato', centering=(1, 0.5))
```

Giving this result:

This time, the target size has a *wider aspect ratio* than the original image, so `pad` does the following:

- First it scales the image to fit the required height. This makes an image that is 300 by 200 pixels in size.
- The image is the correct height, but it isn't wide enough, so the image is padded to make it 400 pixels wide.

We have provides a `centering` value of `(1.0, 0.5)`. Since we are now padding the width, we use the `x` value of 1.0 (the `y` value is ignored). This means that 100% of the padding is placed to the left of the image. Also, notice that we have set a `color` of tomato, so the padding is a red colour.

In summary:

- If the original aspect ratio is less than the target aspect ratio, the original is scaled to fit the target height, then padded to fit the target width.
- If the original aspect ratio is greater than the target aspect ratio, the original is scaled to fit the target width, then padded to fit the target height.
- If the original aspect ratio is equal to the target aspect ratio, the original is the same shape as the target, so it can be scaled to fit with no padding required.

Notice `pad` never applies padding to both the width and height.

`pad` has the following signature:

```
ImageOps.pad(image, size, method=3, color=None, centering=(0.5, 0.5))  # returns a n\
ew image
```

- `image` is the original image.
- `size` is the target size, a tuple of two integers (width, height).
- `method` specifies the resampling method to be applied.
- `color` is the colour to use as the background when the image is padded, default is black.
- `centering` controls the position of the image, a tuple of two numbers (x, y). Defaults to (0.5, 0.5).

8.1.5 fit

The 'fit' function allows you to change the shape of an image, without stretching or distorting the image. It does this by scaling the image so it fills the required size, and then cropping the image as necessary.

Specifically, `fit` scales the image to be the smallest size that completely fills the target size. Depending on the image size and the target size it will either:

- crop the top and bottom of the image or
- crop the left and right of the image

It will never do both because the image will always be scaled so that it either fits the width or fits the height.

This function is similar to `pad` in that it changes an image into a new size. But whereas `pad` shows the entire image and pads any gaps, `fit` just fills the space with as much of the image as possible, cropping rather than padding.

Here is an example. Our original image is 420 by 280 pixels, and we want to pad it to make it 450 by 400 pixels. Here is the code to do that:

```
result_image = ImageOps.fit(image, (450, 400))
```

And here is the result:

The target size has a *taller aspect ratio* than the original image, so `fit` does the following:

- First it scales the image to fit the required height. This makes an image that is 600 by 400 pixels in size.
- The image is the correct height, but it is too wide, so the image is cropped to 450 pixels wide.

In the case where the target size has a wider aspect ratio, the image is scaled to the correct width, then has its height cropped.

`fit` has the following signature:

```
ImageOps.fit(image, size, method=3, centering=(0.5, 0.5))  # returns a new image
```

- `image` is the original image.
- `size` is the target size, a tuple (or other sequence) of two integers (width, height).
- `method` specifies the resampling method to be applied.
- `centering` controls the position of the image, a tuple of two numbers (x, y). Defaults to (0.5, 0.5).

There is no `color` parameter for `fit` because it never adds any padding.

`centering` works in the same way as for `pad`, but it is a little less obvious because there is no padding visible.

8.2 Image transformation functions

These functions perform various simple transformations on images. Also, remember that the `Image` class contains several methods for transforming an image.

8.2.1 flip

`flip` flips an image vertically:

The code to do this is:

```
result_image = ImageOps.flip(image)
```

8.2.2 mirror

`mirror` flips an image horizontally:

The code to do this is:

```
result_image = ImageOps.mirror(image)
```

8.2.3 exif-transpose

Photographers often turn their camera on its side to frame a picture in portrait view (tall rather than wide).

However, when you do that, the camera still creates an image in landscape view, so the subject appears to be rotated through 90 degrees. This can be fixed easily by rotating the image in the opposite direction, but that would require you to look at each image individually to decide if it needed rotating.

Fortunately, many cameras store EXIF data in the image that makes this a lot easier. EXIF is a special type of metadata that can be attached to an image, usually by the camera. It gives details of how the image was obtained (camera model, exposure settings, time and location etc). One setting is the Orientation tag, which typically indicates if the shot was taken in landscape view, or with the camera turned on its side in portrait view.

`exif-transpose` reads the Orientation tag (if it is present), and rotates the image so that it appears in the correct orientation:

- If the tag indicates that the image was taken in portrait mode, it will be rotated.
- If the tag indicates that the image was taken in landscape mode, or if there is no tag present, a copy of the original image is returned.

Here is the code to do this:

```
result_image = ImageOps.exif_transpose(image)
```

If this function is applied to a set of images, it will automatically rotate only those images that need rotating.

8.3 Colour effects

These functions provide various colour effects.

8.3.1 grayscale

`grayscale` accepts an RGB image and converts it to a greyscale image (similar to a black and white photograph). Here is how the original boat image looks after conversion:

Here is the code:

```
result_image = ImageOps.grayscale(image)
```

`grayscale` doesn't take any additional parameters.

8.3.2 colorize

`colorise` takes a greyscale image and adds colour to it. It converts the grey values into a blend of two or three colours, for some interesting effects.

In its simplest form, it works by specifying two colours. Black pixels are set to the first colour, white pixels are set to the second colour, and grey pixels are set to an intermediate colour (depending on the grey value). Here is an example:

In this case, we have used dark blue for the first colour and white for the second colour. This is like a blue-tinged version of a black and white image. Here is the code:

```
result_image = ImageOps.colorize(image, 'darkblue', 'white')
```

In the next example, the colours are dark blue and yellow. The code is the same as above, but with the second colour changed to `"yellow"`. The intermediate greys are replaced by different shades of green:

In this case, we have used a fairly dark colour to replace black, and a fairly light colour to replace white, so the image still looks quite realistic.

It is also possible to specify a third, mid-tone colour. Grey values between 0 and 127 will vary between the black colour and the mid colour. Grey values between 127 and 255 will vary between the mid colour and the white colour. In this case, we have used purple to blue to white, which gives a quite subtle effect, but you could use clashing colours to create a more dramatic effect:

Here is the code to use 3 colours:

```
result_image = ImageOps.colorize(image, 'purple', 'white',
                                        mid='mediumslateblue')
```

By default, the black colour applies at grey level 0 (the `blackpoint`), and the white colour applies at grey level 255 (the `whitepoint`). We can change this. In the image below we set the `blackpoint` to 64, which means any grey level of 64 or less will be set to the black colour. We also set the `whitepoint` to 192, which means that any grey level of 192 or greater will be set to the white colour. Grey values between 64 and 192 will vary smoothly from the black colour to the white colour.

The effect of this is to increase the contrast of the final image. It is mainly intended to correct for a low contrast input image. For example, if you know that the input image only had grey values between 64 and 192, you could use this feature to correct for it.

Here is the code:

```
result_image = ImageOps.colorize(image, 'darkblue', 'white',
                                          blackpoint=64, whitepoint=192)
```

There is also a `midpoint` that can be used to specify where the mid colour applies. It defaults to 127.

`colorize` has the following signature:

```
ImageOps.colorize(image, black, white, mid=None, blackpoint=0,
                         whitepoint=255, midpoint=127)  # returns a new image
```

- `image` is the original image.
- `black` is the colour to use for black pixels.
- `white` is the colour to use for white pixels.
- `mid` is the colour to use for mid-point pixels.
- `blackpoint` is the value that should be mapped to the `black` colour.
- `whitepoint` is the value that should be mapped to the `white` colour.
- `midpoint` is the value that should be mapped to the `mid` colour.

The mapping parameters must be in ascending order, that is:

```
blackpoint <= midpoint <= whitepoint
```

However, if `mid` is not used, the `midpoint` value is ignored.

8.3.3 invert

`invert` inverts all the colour values in an image, creating something that looks like a photographic negative. Here is the result for a colour image:

In the colour image, the red, green and blue channels are inverted individually. This means that each colour is reflected across the colour wheel (for example, red colours become cyan, etc), as well as dark and light being inverted.

Here is the code:

```
result_image = ImageOps.invert(image)
```

`invert` doesn't take any additional parameters.

8.3.4 posterize

Posterization is a process where the number of colours in an image is significantly reduced. This causes areas of the image that have similar colours to be replaced by areas of a single flat colour. This gives an effect similar to old fashioned posters, where the printing technique used only allowed for a limited number of flat colours.

Here is an example:

The `posterize` function gives a fairly crude effect. It reduces the number of bits used for each colour. We normally use 8 bits per colour, which gives 256 different levels of each colour, which means we don't see any quantisation effects. If we reduce the number of bits to 4, then each colour has only 16 possible levels, so in a gradually changing colour, we see distinct bands where the colour looks the same.

The image above only uses 2 bits per colour per pixel, which means that there are only 64 different colours in the image (red, green and blue can each have a level of 0 to 3).

Here is the code:

```
result_image = ImageOps.posterize(image, 2)
```

This effect was quite popular in the early days of computer imaging because it is very quick to compute. But the result is very noisy, and you don't have any control over which colours are used. A much better effect can be obtained but segmenting the image (dividing it into areas of similar colour) and then setting each region to a chosen colour.

`posterize` has the following signature:

```
ImageOps.posterize(image, bits)  # returns a new image
```

- `image` is the original image.
- `bits` is the number of bits per colour to use, an integer from 1 to 8.

8.3.5 solarize

Solarizing is an effect that inverts the brightest parts of the image.

Here is an example:

Solarize works by calculating the grey level of each pixel. If the grey level for a particular pixel is greater than a threshold (set to the default of 128 in the example) that pixel is inverted. Otherwise, it is left unchanged.

Looking at the image, you can see that the sky, and the white hull of the boat, are inverted, but the trees in the background are unchanged. This gives a surreal effect.

Here is the code to solarize an image:

```
result_image = ImageOps.solarize(image)
```

`solarize` has the following signature:

```
ImageOps.solarize(image, threshold=128)  # returns a new image
```

- `image` is the original image.
- `threshold` the grey value threshold. All pixels that are lighter than this are inverted.

8.4 Image adjustment

Pillow allows us to apply certain automatic adjustments to images.

8.4.1 autocontrast

The contrast of an image describes the range of lightnesses of all the different pixels in the image.

If an image has some very dark pixels and some very bright pixels, we say it has high contrast.

If all the pixels in an image have similar levels of brightness, we say it has low contrast. This could be due to all the pixels being quite dark, or all the pixels being quite bright, or maybe all the pixels being a similar mid-level of brightness.

In general, images often look better if they have good contrast. It means they are taking full advantage of the range of colour values available on the screen or printer, and it typically allows you to see more detail in the image. It is possible to have too much contrast (see below) but a decent amount of contrast is usually a good thing.

`autocontrast` modifies the contrast of an image automatically. It measures the darkest and lightest parts of the image, then it adjusts the brightness of every pixel so they occupy the whole available range. Here is how it is used:

```
result_image = ImageOps.autocontrast(image)
```

For many natural images, this will create a more pleasing result.

`autocontrast` has a few optional parameters:

- `cutoff` ignores a certain percentage of the lightest and darkest pixels. This can be useful, for example, if you have a low contrast image that contains a few noise pixels. If those noise pixels are very dark or very bright that could throw off the histogram calculation. Pillow might think the image already has very dark and light pixels so it doesn't need adjusting. A cutoff of `2` will ignore the highest and lowest `2%` of values when calculating the adjustment. A cutoff of `(3, 5)` will ignore the lowest `3%` and highest `5%` of values.
- `ignore` will ignore a specific colour. This is useful if your image has a specific background that you don't want to include in the contrast calculation. For example, if the image was a diagram on a white background, you might want to correct the contrast of the diagram without taking the background into account. You could do this by setting the ignore colour to white.
- `mask` is a mask image, the same pixel size as the main image. If a mask is provided, it determines which pixels should be included in the calculation. For example, if there was a particularly important area of the image you could select that with a mask. `autocontrast` would then adjust the image to ensure that area of the image had good contrast. The same adjustment would be applied to the whole image, but it would not necessarily give good results in other parts of

the image.

There are times when autocontrast won't work well. These are usually cases where the image is genuinely low contrast. For example, a photograph taken on an overcast day might not contain any strong light or dark areas. If you use autocontrast to try to force some areas to be very light and dark, it would look very unnatural.

8.4.2 equalize

`equalize` does a similar job to `autocontrast`, in that it adjusts the distribution of intensities. However, instead of just spreading the range of brightness values, it tries to ensure that each part of the histogram contains the same number of pixels.

This is a stronger effect than autocontrast. The benefit is that it maximises the visible detail in the light, mid-tone, and dark areas of the image. The downside is that it is often less visually appealing.

It is called like this:

```
result_image = ImageOps.equalize(image)
```

It has one optional parameter, `mask`, that works as described for `autocontrast`.

8.5 Deforming images

You can use the ImageOps `deform` function to apply general deformations to an image. Typical deformations include:

- Barrel distortion. This happens when an image appears more magnified at its centre than its edges. It is sometimes called a fisheye effect.
- Pincushion distortion, which is the opposite of barrel distortion.
- Perspective distortion. For example, if you take a photograph looking up at a tall building, the building will appear narrower at the top due to perspective.

You can use the `deform` function to correct for these types of distortion in a photograph. You can also use it to add these types of distortion to a photograph for artistic effect. There are many other types of deformation you can add too.

We will look at how to add this simple wave deformation to an image:

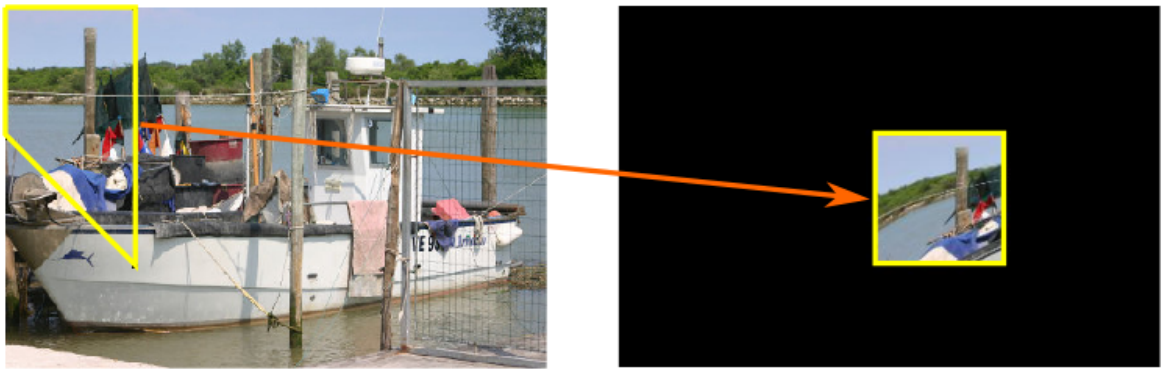
8.5.1 How deform works

Deformation takes an existing image (the source image) and creates a new image of the same size and mode (the target image).

The deformation is controlled by a mesh:

- The mesh defines one or more rectangular regions on the target image. Each rectangle is aligned with the x and y axes of the image.
- For each target region, the mesh defines a quadrilateral region in the source image.
- `deform` copies each region from the source to the target. The source region is deformed to fit the target rectangle.

Here is a simple deformation that maps one region of the source image onto one region of the destination image:



Here the non-rectangular source region is translated to a new position in the target image and also squeezed into a square shape. This distorts the pixel image data as shown.

It is possible to scale, rotate, mirror, skew, or deform the region, depending on the shape of the source quadrilateral and the order of the vertices.

This example shows a single region. The code is described below. In most cases, you will divide the image into many regions to apply a deformation to the whole image.

8.5.2 getmesh

The `deform` function needs to know how to map the target image onto the required regions of the source image. To do that, it requires a deformer object.

The deformer object must implement a `getmesh` function which:

- Accepts the image as a parameter.
- Returns a list of mappings.

Here is a simple deformer that maps a single region:

```
class SingleDeformer:

    def getmesh(self, img):
        #Map a target rectangle onto a source quad
        return [(
                # target rectangle
                (200, 100, 300, 200),
                # corresponding source quadrilateral
                (0, 0, 0, 100, 100, 200, 100, 0)
                )]
```

A mapping takes the form:

```
((200, 100, 300, 200), (0, 0, 0, 100, 100, 200, 100, 0))
```

This mapping contains:

- The target rectangle, defined by the two points `(200, 100)` and `(300, 200)`.
- The source quad, represented by the four points `(0, 0)`, `(0, 100)`, `(100, 200)`, `and (100, 0)`‘.

Notice that `getmesh` is required to return a list of mappings. Since there is only one mapping, it returns a single element list:

```
[((200, 100, 300, 200), (0, 0, 0, 100, 100, 200, 100, 0))]
```

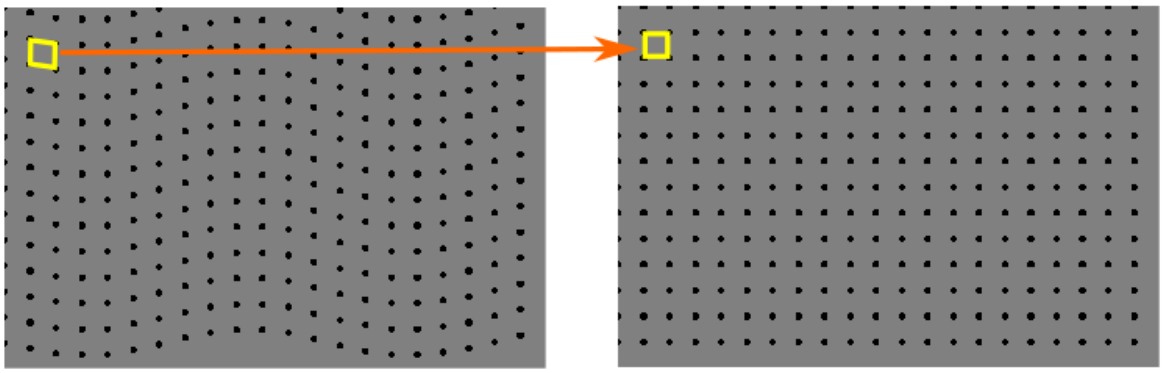
Here is how we use this deformer on an image:

```
image = Image.open('boat-small.jpg')
result_image = ImageOps.deform(image, SingleDeformer())
result_image.save('imageops-deform.jpg')
```

The result of this mapping was shown in the image above.

8.5.3 A wave transform

Now we will take a look at the wave-transform from the beginning of this section. This requires us to divide the image up into lots of small regions, and map each one separately. This diagram shows the regions we will use:



The right-hand side shows the target mesh. It is a grid of 20 by 20 pixel squares.

The left-hand side shows the source quads. Each square of the target is taken from a parallelogram-shaped area of the source image, that has been displaced and sheared relative to the target grid. The overall effect of this is to make a wavey image.

Here is the wave deformer:

```
class WaveDeformer:

    def transform(self, x, y):
        y = y + 10*math.sin(x/40)
        return x, y

    def transform_rectangle(self, x0, y0, x1, y1):
        return (*self.transform(x0, y0),
                *self.transform(x0, y1),
                *self.transform(x1, y1),
                *self.transform(x1, y0),
```

```
            )

    def getmesh(self, img):
        self.w, self.h = img.size
        gridspace = 20

        target_grid = []
        for x in range(0, self.w, gridspace):
            for y in range(0, self.h, gridspace):
                target_grid.append((x, y, x + gridspace, y + gridspace))

        source_grid = [self.transform_rectangle(*rect) for rect in target_grid]

        return [t for t in zip(target_grid, source_grid)]
```

There are lots of squares in the mesh, so we will create them programmatically inside `getmesh`. First, the target mesh:

```
        target_grid = []
        for x in range(0, self.w, gridspace):
            for y in range(0, self.h, gridspace):
                target_grid.append((x, y, x + gridspace, y + gridspace))
```

This simply creates a set of squares, of size `gridspace` by `gridspace`, that completely cover the image.

Next we create an equivalent set of source quads:

```
        source_grid = [self.transform_rectangle(*rect) for rect in target_grid]
```

`transform_rectangle` is called once for each square `(x0, y0, x1, y1)`. It expands this square into a quad with corners `(x0, y0)`, `(x0, y1)`, `(x1, y1)`, and `(x1, y0)`. It then calls `transform` to transform the position of each corner.

It is the `transform` function that controls what happens to the image. The function leaves the x coordinate unchanged but displaces the y component by a function related to the sine of x. This causes the image to be displaced vertically as you move along the image in the horizontal direction.

8.5.4 Other deformations

The `WaveDeformer` class provides a general method of creating a set of target squares and corresponding source quads.

You can create a variety of other deformations simply by altering the `transform` method.

9. Image attributes and statistics

Pillow supports several different types of image data and uses a variety of attributes to describe these different types.

Several image attributes are stored directly in the `Image` object:

- Size - the width and height of the image in pixels.
- Filename - the name of the file containing the image.
- File format - the format of the file containing the image (JPEG, PNG etc).
- Mode - the colour mode of the image data (RGB etc).
- Bands - the colour bands in the image (closely related to the mode).
- Palette - the colour palette used by the image, if it is palette-based.
- Info - extra information about the image. The type of data depends on the file format and is often optional.
- Animated - a flag that indicates if the image is animated (for example, if it is an animated GIF).
- Frames - for animated formats, the number of frames in the animation
- EXIF data

Finally, Pillow can be used to calculate various image statistics, such as the histogram. This is mainly done via the `ImageStats` module, which we will also cover in this chapter.

9.1 Attributes

For the examples below we will use three image objects:

```
from PIL import Image

new_image = Image.new('L', (400, 300), 'darkgrey')
jpeg_image = Image.open('boat.jpg')
gif_image = Image.open('spinning-cube-ani.gif')
```

`new_image` is a new, blank image (ie it has been created by Pillow, not read from file) with mode `L`, which is a greyscale image. This is created with a size of 400by 300 pixels.

`jpeg_image` is the familiar *boat.jpg* image, read in from a JPEG file. It is 600 by 400 pixels.

`gif_image` is an image read in from a GIF animation of a spinning cube, containing 80 frames.

All these images are available from the Github repo for this book.

9.1.1 File size

The `size` property of an `Image` object gives width and height of the image, in pixels. The size is returned as a tuple of two integers `(width, height)`:

```
print("Size:")
print(" new_image", new_image.size)                    # (400, 300)
print(" jpeg_image", jpeg_image.size)                  # (600, 400)
print(" gif_image", gif_image.size)                    # (400, 400)
```

This code prints the value of the property (the result is also shown in the comment in the code above):

```
Size:
 new_image (400, 300)
 jpeg_image (600, 400)
 gif_image (400, 400)
```

There is also a `width` property, which returns the width as an integer, and a `height` property, which returns the height as an integer.

9.1.2 File name

The `filename` property of an `Image` object gives the name of the file it was originally read from.

For a new image, the property doesn't exist, so if you called `new_image.filename` it would raise an exception. To avoid this, we can use `getattr` to get the property value. `getattr` accepts a default value, so if the property doesn't exist it will return that default value. In the code below, the default value is `None`:

```
print("File name:")
print(" new_image", getattr(new_image, "filename", None))    # None
print(" jpeg_image", getattr(jpeg_image, "filename", None))  # 'boat.jpg'
print(" gif_image", getattr(gif_image, "filename", None)) # 'spinning-cube-ani.gif'
```

For brevity, we won't show the result of the print statements, the expected result is included in the code comments. As you would expect there is no filename for the new file.

9.1.3 File format

The `format` property of an `Image` object gives the format of the file it was originally read from. As with the `filename`, it isn't present for an image that wasn't originally read from file, so again we use `getattr`. Here is the code and result:

```
print("File format:")
print(" new_image", getattr(new_image, "format", None))      # None
print(" jpeg_image", getattr(jpeg_image, "format", None))    # 'JPEG'
print(" gif_image", getattr(gif_image, "format", None))      # 'GIF'
```

The file format is returned as a string. Pillow supports a large number of formats. Some of the more common ones are:

- JPEG, PNG, GIF, and TIFF, discussed in the earlier chapter on image file formats.
- EPS, a vector format based on PostScript.
- BMP, DIB, WMF, ICO, and CUR, formats typically used by the Windows operating system.
- ICNS format used by macOS.
- Targa TGA format.

9.1.4 Mode and bands

The `mode` property of an `Image` object gives the colour mode that the data is stored in.

```
print("Mode:")
print(" new_image", new_image.mode)                          # 'L'
print(" jpeg_image", jpeg_image.mode)                        # 'RGB'
print(" gif_image", gif_image.mode)                          # 'P'
```

The mode is defined by a string. In the cases above, the new image is an 8 bit per pixel greyscale image (denoted by L). That is no surprise, we specified the L colour model when we created the image.

The JPEG image is an RGB image.

The GIF image is palette-based, represented by P. Each pixel in a GIF image is a single byte value. This is used as an index into a palette of up to 256 RGB values.

A full list of the possible colour modes is included in the *Reference* section.

The `getbands` function provides similar information, but it returns a tuple of single character names for the image bands. For example,and RGB image has bands `('R', 'G', 'B')`:

```
print("Bands:")
print(" new_image", new_image.getbands())                    # ('L',)
print(" jpeg_image", jpeg_image.getbands())                  # ('R', 'G', 'B')
print(" gif_image", gif_image.getbands())                    # ('P',)
```

9.1.5 Palette

The property `palette` returns an `ImagePalette` object if the image has a palette, or `None` if it does not.

```
print("Palette:")
print(" new_image", new_image.palette)                    # None
print(" jpeg_image", jpeg_image.palette)                  # None
print(" gif_image", gif_image.palette)                    # <ImagePalette>
```

The `ImagePalette` object is not well defined in the Pillow documentation. An alternative way to access the pixel data is to use the `getpalette` method of the Image object:

```
gif_image = Image.open('spinning-cube-ani.gif')
palette = gif_image.getpalette()
print(len(palette))
print(palette)
```

`getpalette` returns a list containing the RGB values of the palette. In the particular image used, the palette length is 768. Since each palette entry has three elements (R, G, and B) this means that the palette has 256 entries in total. Here is the palette data (only the first and last few values are shown):

```
[255, 255, 255, 64, 64, 64 ... 127, 126, 24, 114, 24, 144]
```

This data is in the format `R, G, B, R, G, B ...` so:

- Palette entry 0 has RGB value (255, 255, 255).
- Palette entry 1 has RGB value (64, 64, 64).
- ...
- Palette entry 254 has RGB value (127, 126, 24).
- Palette entry 255 has RGB value (114, 24, 144).

9.1.6 Info

The property `info` returns a dictionary object containing extra information stored in the image file. This information is format-specific, and many fields are optional, so if you are looking for specific information your code should be prepared to handle the case where the information is missing.

```
print("Info:")
print(" new_image", new_image.info)                       # {}
print(" jpeg_image", jpeg_image.info)                     # {various}
print(" gif_image", gif_image.info)                       # {various}
```

`new_image` has no info because it was not read from a file. It still has an `info` dictionary but it is empty.

`jpeg_image` returns the following dictionary (this may be different for other images):

```
{'jfif': 257,
 'jfif_version': (1, 1),
 'dpi': (180, 180),
 'jfif_unit': 1,
 'jfif_density': (180, 180),
 'exif': <binary data> }
```

The `exif` entry in this dictionary (which may or may not be present) contains binary data. EXIF is covered later in this chapter.

`gif_image` returns the following (again it might be different for other GIF files):

```
{'version': b'GIF89a',
 'background': 0,
 'duration': 50,
 'extension': (b'NETSCAPE2.0', 795),
 'loop': 65535}
```

If you need to know what the individual info items mean for any particular format, it is best to refer to the format specification. There are far too many to list here.

9.1.7 Animation

Most image formats don't support animation. The most common one that does is GIF format. The example we are using is a small animated GIF.

The `is_animated` property is true of the image is animated. For formats that don't support animation at all, this property will most likely not exist, so again we wil use `getattr`:

```
print("Animation:")
print(" new_image", getattr(new_image, "is_animated", None))    # None
print(" jpeg_image", getattr(jpeg_image, "is_animated", None))  # None
print(" gif_image", getattr(gif_image, "is_animated", None))    # True
```

In this case, only the GIF file has this flag present and true. If the flag is true we can use the `n_frames` property to find out how many frames there are:

```
print("Frames:")
print(" new_image", getattr(new_image, "n_frames", None))    # None
print(" jpeg_image", getattr(jpeg_image, "n_frames", None))  # None
print(" gif_image", getattr(gif_image, "n_frames", None))    # 80
```

9.1.8 EXIF tags

We saw earlier how to get the EXIF data of an image (if it exists) from the `exif` entry in the `info` dictionary. However, it is usually easier to use the `getexif` function:

```
from PIL import Image, ExifTags

jpeg_image = Image.open('boat.jpg')
exif = jpeg_image.getexif()

for tag in exif:
    tagname = ExifTags.TAGS[tag]
    value = exif[tag]
    print('{}: {}'.format(tagname, value))
```

EXIF is a binary format that contains numbered tags. Each tag has some associated binary data that might represent a number, string, timestamp or other information.

`getexif` returns a dictionary-like object, `exif`, containing the EXIF tags and values. In the code above, we loop over every tag in the `exif` structure. For each tag:

- We look up the name of the tag in the `ExifTags.TAGS` dictionary. This converts an integer value into a tag name. For example, tag number 256 is the tag that holds the image width. The `ExifTags.TAGS` dictionary maps 256 onto the string `'ImageWidth'`.
- We look up the tag value using the `exif` object. In this case, tag 256 has a value of 600, the width of the image in pixels.

Here is what the code prints out (the InterColorProfile is a binary object, not shown):

```
ImageWidth: 600
ImageLength: 400
ResolutionUnit: 2
ExifOffset: 170
Make: Canon
Model: Canon EOS 300D DIGITAL
Software: PaintShop Pro 20.00
Orientation: 1
YCbCrSubSampling: 1
DateTime: 2007:05:31 14:04:43
InterColorProfile: b'...'
XResolution: 180.0
YResolution: 180.0
```

9.2 Image statistics

The Image module allows us to calculate an image histogram and a few other statistics. The ImageStat module provides more detailed statistics.

In this section we will use the `jpeg_image` from the examples above, and also a greyscale version of this image, calculated like this:

```
greyscale_image = jpeg_image.convert('L')
```

9.2.1 Image histogram

We can calculate the histogram of a greyscale image like this:

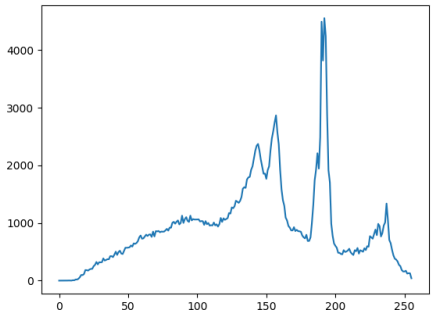
```
histogram = greyscale_image.histogram()
print("Histogram length:", len(histogram))  # 256
print("Histogram:", histogram)              # [0, 0, 0, 0, 1, 1, 1, 2, ... 41]
```

This returns a list of length 256 integers representing the greyscale pixel values 0 to 255. Each number in the list tells us how many pixels with that value exist in the entire image. So:

- Entry 0 has value 0. This tells us that there are no pixels in the image that have a grey value of 0.
- Entry 1 has value 0. This tells us that there are no pixels in the image that have grey value 1.
- Skipping ahead a little, entry 7 has value 2, so that tells us there are exactly 2 pixels in the image that have a grey value of 7.

There are 256 entries, but we have missed most of them out to save space. The final entry has a value of 41, this tells us that there are 41 pixels in the image with a grey value of 255.

Histograms are useful to show us how the pixels are distributed among the grey levels. Here is a plot of the values in the histogram above:



This histogram shows us that the image is fairly well spread over the range 0 to 255. Most parts of the range are used. There are some peaks in the graph, which indicate a lot of pixels with similar grey levels. That is to be expected. The sky, for example, has lots of pixels that are a similar colour, so that probably accounts for one of the peaks. The white area of the boat will also create a peak, and the water of the river another one. We wouldn't usually expect the histogram of an image to be completely flat.

We can also calculate the histogram of an RGB image:

```
histogram = jpeg_image.histogram()
print("Histogram length:", len(histogram))  # 768
print("Histogram:", histogram[:256])        # [0, 0, 2, ... 1057]
print("Histogram:", histogram[256:512])     # [5, 2, 3, ... 91]
print("Histogram:", histogram[512:])        # [45, 12, 8, ... 129]
```

This time the histogram length is 768. It is three separate histograms. The first 256 elements are a histogram of the values of the red channel, the next 256 are a histogram of the green channel, and the final 256 represent the blue channel.

9.2.2 Masking

The `histogram` function can accept an optional `mask` parameter:

```
histogram = jpeg_image.histogram(mask=mask_image)
```

If this parameter is supplied it be must be an Image object that is the same size as the main image. It must have a mode of `L` (greyscale) or `1` (bilevel).

When a mask is supplied, the histogram calculation will only take account of pixels where the corresponding mask image pixel has a non-zero value.

9.2.3 Other Image statistics

Here are a couple of other useful statistics methods of Image objects:

```
print("Extrema:", greyscale_image.getextrema())  # (4, 255)
print("Extrema:", jpeg_image.getextrema())       # ((2, 255), (0, 255), (0, 255))
print("Entropy:", jpeg_image.entropy())          # 9.21637999047619
```

`getextrema` gets the min and max values of all the pixels.

For the greyscale image, it returns a single tuple (4, 255), showing the minimum value is 4 and the maximum value is 255. Notice that this agrees with the previous histogram for `greyscale_image`, which showed that the first 4 values of the histogram are zero, meaning that there are no pixels with values 0, 1, 2, or 3. The smallest value is 4.

For an image with more than one colour channel, the function returns a tuple of pairs, one for each channel. In the case of `jpeg_image`, this shows that the red channel has a min of 2 and a max of 255. The green and blue channels each have values between 0 and 255. Again, this agrees with the previous histogram.

The `getextrema` function can accept a `mask` parameter that works in a similar way to the histogram `mask` described earlier.

You can also find the entropy of the image using the `entropy` method. This provides a measure of how disordered the image is (the higher the entropy, the more disordered the image is). Typically images that appear noisy have higher entropy.

9.2.4 ImageStat module

The ImageStat module can be used to find additional statistical information about an image. Here is how to do it:

```
stat = ImageStat.Stat(jpeg_image)
print("stat.extrema", stat.extrema) # [(2, 255), (0, 255), (0, 255)]
print("stat.count", stat.count)     # [240000, 240000, 240000]
print("stat.sum", stat.sum)         # [34365232.0, 34592832.0, 33760460.0]
print("stat.sum2", stat.sum2)       # [5623187660.0, 5659892442.0, 5688740626.0]
print("stat.mean", stat.mean)       # [143.18846666666667, 144.1368, 140.66858333333\
334]
print("stat.median", stat.median)   # [145, 148, 144]
print("stat.rms", stat.rms)         # [153.0684441135185, 153.56720084379998, 153.95\
806552976256]
print("stat.var", stat.var)         # [2927.011596982221, 2807.468060760001, 3915.43\
56046597204]
print("stat.stddev", stat.stddev)   # [54.101863156292694, 52.98554577203109, 62.573\
44168782568]
```

Essentially, we create an `ImageStat.Stat` object, passing in the image object. We can then read the statistics as properties of that object.

For single-channel images (such as greyscale images) the property will be a single value. For multichannel images (such as RGB images) the property will be a list of values, one per channel. Here is a description of the various statistics:

- `extrema` gives the minimum and maximum pixel values. For 8 bit per channel images, this will give the same result as the `Image.getextrema` method. For other image types (such as images with floating-point values) the result will not be quite correct because it is calculated from the histogram rather than the raw image values. The `Image.getextrema` method will always give the correct result, so you might consider using that instead.
- `count` gives the total number of pixels in the image. This is simply the width times the height. For a multi-channel image, of course, all the values will be the same.
- `sum` gives the sum of all the pixel values in the image. For each channel, it just adds the value of every pixel
- `sum2` gives the sum of the squares all the pixel values in the image. This is useful for calculating the RMS, below.
- `mean` calculates the mean average. For each channel, this is the `sum` divided by the `count`.
- `median` calculates the median average. For each channel, if all the pixels were sorted in ascending order, the median would be the middle pixel. Essentially, half the pixels have a value that is less than the median, half the pixels have a value that is greater than the median.

- `rms` calculates the root mean square average. For each channel, it calculates `sum2` divided by the `count`, then takes the square root.
- `var` calculates the variance. For each channel, it calculates the average of the squared difference between each pixel value and the mean. The variance is mainly used to calculate the standard deviation, below.
- `stddev` calculates the standard deviation. This is a measure of the spread of the pixel values. It is calculated as the square root of the variance. A `stddev` of zero indicates that every pixel has the same value. A `stddev` of 5 would indicate that quite a lot of the pixels have values that are within +/- 5 of the mean, so many of the pixels have very similar values. Our boat image has a `stddev` of around 50, which indicates a good spread of different values.

10. Enhancing and filtering images

In this chapter we will look at two modules that provide some commonly used improvements and effects:

- The `ImageEnhance` module that provides contrast, brightness, colour and sharpness enhancements.
- The `ImageFilter` module that a large range of filters for smoothing, embossing, edge enhancement and more. It also allows you to create your own filters.

The `ImageFilter` module could be thought of as being like the *expert level* of the `ImageEnhance` module. It provides more functions, and more control of each function. However, `ImageEnhance` is easier to use if you just need the basic filters it provides.

10.1 ImageEnhance

The `ImageEnhance` module provides a few easy to use enhancements, each controlled by a single parameter. There are the basic enhancements that you would expect to find in any image processing software:

- Brightness - makes the image brighter or darker.
- Contrast - increases or decreases the range of tonal values.
- Colour - makes the image more or less colourful.
- Sharpness - makes the image sharper or more blurred.

Each feature is controlled by a single value, the `factor`. In general:

- A factor of `1.0` leaves the image unchanged.
- A factor of less than `1.0` decreases the effect, and `0.0` removes it altogether.
- A factor of greater than `1.0` increases the effect. There is no upper limit.

To use `ImageEnhance`, you first create an enhancer object, then call the `enhance` method passing in the factor. We will illustrate this with the `Brightness` enhancer.

10.1.1 Brightness

Here is how we alter the brightness of an image:

```
from PIL import Image, ImageEnhance

image = Image.open('boat-small.jpg')

enhancer = ImageEnhance.Brightness(image)
less_image = enhancer.enhance(0.5)
more_image = enhancer.enhance(1.5)
```

After the import statement, we open the boat image as `image`.

We then create an `enhancer` object by calling `ImageEnhance.Brightness`, passing the `image` in as a parameter:

```
enhancer = ImageEnhance.Brightness(image)
```

We then call the `enhance` method of the enhancer, passing in a value of 0.5. This creates a new image, based on the original image, but with less brightness (ie darker). We store this as `less_image`:

```
less_image = enhancer.enhance(0.5)
```

If we were to reduce the factor to zero, the image would be solid black.

We also create a `more_image` that is brighter than the original (using a factor of 1.5). Here are the three images, side by side (the less image, the original image, and the more image):

10.1.2 Contrast

Changing the contrast of an image works in a very similar way to changing the brightness, we just use `ImageEnhance.Contrast` instead. All the surrounding code is identical:

```
enhancer = ImageEnhance.Contrast(image)
less_image = enhancer.enhance(0.5)
more_image = enhancer.enhance(1.5)
```

In this case, a factor of 0.5 gives an image with less contrast, that is the range of brightnesses is reduced, so light and dark colours are pushed towards a midrange grey colour. If the contrast were to be reduced to 0.0, the image just becomes a flat grey rectangle.

A factor of 1.5 increases the contrast, so light colours become lighter and dark colours become darker.

Here is the result, again the less image, the original image and the more image:

10.1.3 Color

Colour enhancement makes the image more or less colourful:

```
enhancer = ImageEnhance.Color(image)
less_image = enhancer.enhance(0.5)
more_image = enhancer.enhance(1.5)
```

In this case, a factor of 0.5 gives an image with less colour. The colour if each pixel is made less vibrant, moving towards a grey colour. If the factor were to be reduced to 0.0, the result would be a greyscale image.

A factor of 1.5 increases the colour, making each pixel more vibrant.

Here is the result:

10.1.4 Sharpness

Sharpness enhancement makes the image appear either sharper or more blurred:

```
enhancer = ImageEnhance.Sharpness(image)
less_image = enhancer.enhance(0.5)
more_image = enhancer.enhance(1.5)
```

In this case, a factor of 0.5 gives an image with less sharpness, in other words slightly blurred. The colour if each pixel is made less vibrant, moving towards a grey colour. If the factor were to be reduced to 0.0, the result would be a greyscale image.

A factor of 1.5 makes the image appear sharper. It does this by enhancing edges in the image.

Here is the result:

10.2 ImageFilter

In the rest of this chapter we will look at the `ImageFilter` module.

A filter object doesn't process images directly. Instead, you must pass the filter object into the `filter` method of `Image`, and it will then be applied to the image. This is shown in the examples below.

We have divided the filters into three groups:

- Predefined filters - these are filters that just apply one effect, with no parameters.
- Parameterised filters - these are filters that take one or more simple parameters to vary the effect.
- User-defined filters - these are filters that offer far more control, but take a bit more understanding to use effectively.

10.3 Predefined filters

These built-in filters take no parameters. Here is how to use the blur filter:

```
from PIL import Image, ImageFilter

image = Image.open('boat-small.jpg')
blur_image = image.filter(ImageFilter.BLUR)
```

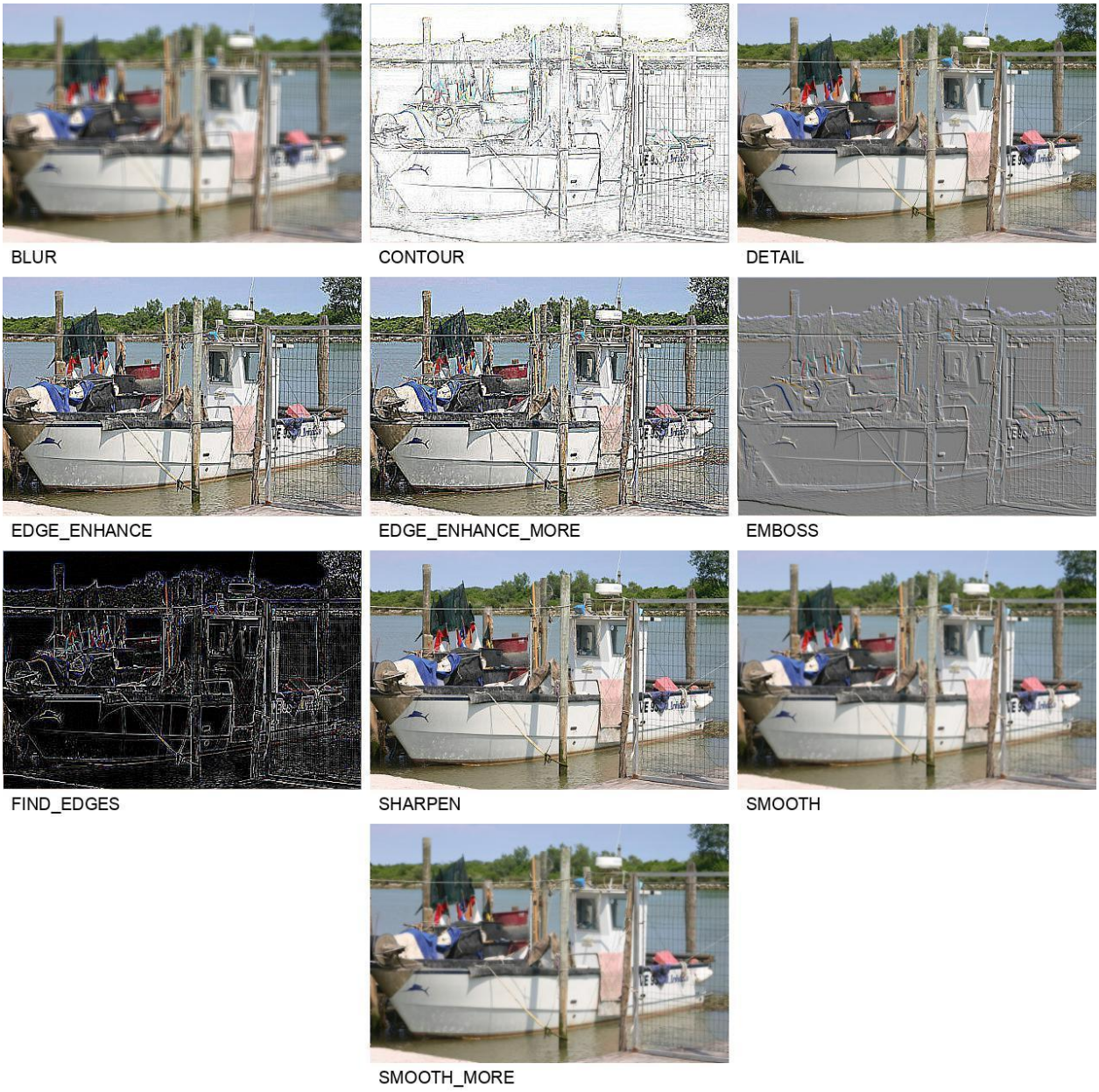
In this example, `ImageFilter.BLUR` is the built in blur filter.

When we call `image.filter`, it applies the filter to `image`, and returns the filtered image.

There are 10 predefined filters:

- BLUR
- CONTOUR
- DETAIL
- EDGE_ENHANCE
- EDGE_ENHANCE_MORE
- EMBOSS
- FIND_EDGES
- SHARPEN
- SMOOTH
- SMOOTH_MORE

They are fairly standard filters that you will find in most image processing applications. Here is what each one does:



10.4 Parameterised filters

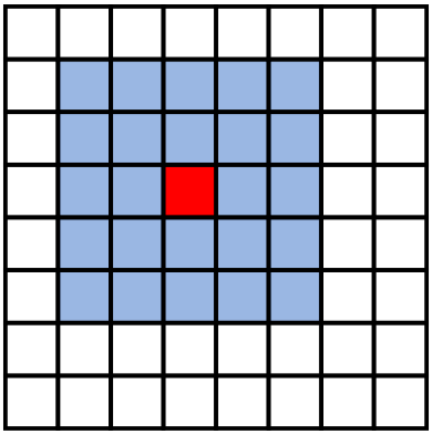
There are several more filters that each take one or more simple parameters that control their behaviour. These filters are mainly concerned with blurring, sharpening and noise removal.

10.4.1 Blurring functions

There are two blurring:

- `BoxBlur`
- `GaussianBlur`

These functions make the image blurred, an effect similar to being out of focus. They generally work by replacing each pixel with a new value obtained by averaging the surrounding pixels. For example, in the image below, we show a blur with a *radius* of 2:



For the red pixel, the value is replaced by an average that takes account of every surrounding pixel that is 2 or less pixels away in any direction. So that will be the average of al the blue pixels and the red pixel (a total of 5 by 5, or 25 pixels).

For box blur, the calculation uses a simple mean average of all 25 pixels.

For Gaussian blur, the calculation is a weighted average using the Gaussian function, which basically means that the pixels that are closest to the centre have more influence than the pixels that are further away. This usually gives a better result, so we will use it for the example below.

Here is the effect of the `GaussianBlur` filter with a radius of 4:

Here is the code:

```
from PIL import Image, ImageFilter

image = Image.open('boat-small.jpg')
result_image = image.filter(ImageFilter.GaussianBlur(4))
```

Blurring is useful for various things:

- If an image is noisy or grainy, a small amount of blurring can improve the appearance.
- Blurring is often used as an artistic effect, similar to using soft-focus filters in photography.
- Blurring can be used to redact parts of an image, for example an email address, car number plate, or face in an image that needs to be hidden for legal reasons. Blurring is less intrusive than a solid black bar.

`GaussianBlur` has this signature:

```
GaussianBlur(radius)
```

- `radius` is the number of pixels that are taken into account.

A radius of 1 takes account of every pixel within 1 pixel of the target pixel, so it uses a 3 by 3 square. A radius of 2 takes account of every pixel within 2 pixels of the target pixel, so it uses a 5 by 5 square (as shown in the diagram above). Radius 3 uses a 7 by 7 square, and so on.

A larger radius creates a more blurred image.

`BoxBlur` is used in exactly the same way as `GaussianBlur`, but it usually doesn't give quite as good a result. It was originally used because the calculation is simpler, but with a modern computer there is no real performance advantage.

10.4.2 Unsharp masking

Unsharp masking is a technique for making images appear sharper. Here is the code:

```
from PIL import Image, ImageFilter

image = Image.open('boat-small.jpg')
result_image = image.filter(ImageFilter.UnsharpMask(4))
```

Here is the resulting image:

Here is how it works. In the image below:

- **A** is the original image.
- **B** is the original image with gaussian blur applied.
- **C** is the *difference* between **A** and **B**. That is, for each colour of each pixel, we set the value in **C** equal to **A** - **B**.

Unsharp masking works by adding image **C** to image **A**. This creates a sharper version of image **A**.

A simple explanation is this. **B** is a blurred image. If we add **C** to **B** we get **A** (because **C** is **A** - **B**). So adding **C** to **B** makes it sharper.

But what if we add **C** to the original image, **A**? That would make it even sharper, right? As unconvincing as that might sound, it does actually work very well for a wide variety of images.

`UnsharpMask` has the following signature:

```
UnsharpMask(radius=2, percent=150, threshold=3)  # creates a filter object
```

- `radius` the blur radius, as for `GaussianBlur`. Controls the extent of the sharpening.
- `percent` the amount of image **C** to add to image **A**, as a percentage. 150 means that the values are multiplied by 1.5 before being added. This controls the strength of the sharpening.
- `threshold` - the minimum **C** value that will be sharpened.

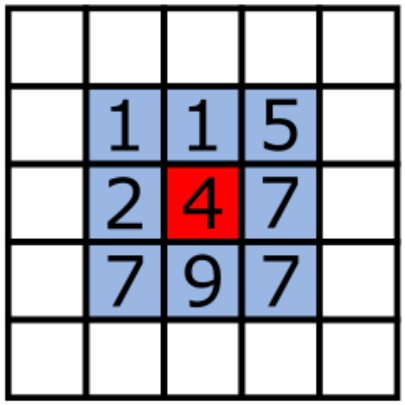
If too much sharpening is applied it can look unnatural, and can even result in "halos" appearing around edges. Reduce either the `radius` or `percent` to avoid this.

The `threshold` means that sharpening is only applied if there is a significant difference between **A** and **B**. This helps to reduce noise in parts of the image that don't contain edges that need sharpening.

10.4.3 Ranking and averaging filters

There are a group of filters that work by ranking the nearest pixels in order.

These filters operate on a square area of pixel called a *kernel*. Typically a kernel is 3 by 3 pixels, or 5 by 5 pixels. Here is a 3 by 3 example:



The new value of the centre pixel (marked in red, with a value 5) depends on the values of all 9 pixels in the kernel. The filter is applied to each pixel in the image, based on the kernel of surrounding pixels

We first arrange the 9 values in order:

```
1 1 2 4 5 7 7 7 9
```

A `MedianFilter` takes the middle value from the ordered list. So in this case the new value of the red pixel will be 5. Here is the code:

```
image = Image.open('boat-small.jpg')
median_image = image.filter(ImageFilter.MedianFilter())
```

A median filter is good for removing noise, because it will reject outliers. For example, imagine there was a glitch the largest pixel value was 100 instead of 9:

```
1 1 2 4 5 7 7 7 100
```

The median value is still 5, despite one of the pixels being very high. In fact, if all the other pixels in the area had low values, the one glitch pixel would be completely ignored.

`MedianFilter` has the following signature:

```
MedianFilter(size=3)  # creates a filter object
```

- `size` the size of the kernel. 3 for a 3 by 3 kernel, 5 for a 5 by 5 kernel, and so on.

Notice that `size` is different to the `radius` used by `GaussianBlur`:

- A `radius` of 1 corresponds to a `size` of 3 (they both give a 3 by 3 kernel).
- A `radius` of 2 corresponds to a `size` of 5 (they both give a 5 by 5 kernel).
- etc

A `MinFilter` works in the same way, but instead of the medial value it always returns the minimum value, so in the example above it would return 1. This is good if you have a dark image and you are specifically trying to remove bright pixel noise. `MaxFilter` does the opposite, it always returns the maximum value. It can be used if you have a light image with dark noise pixels.

A `MeanFilter` takes the average of all the pixels in the kernel. Using the data above, the mean value is about 4.8, so 5 will be used. `MedianFilter` is essentially the same as `BoxBlur`, except that it uses a `size` rather than a `radius`.

A `ModeFilter` takes the most frequent of the pixels in the kernel. With the data above, the filter would return 7, because there are 3 pixels with the value 7. This filter will only select a value if it occurs at least 3 times in the kernel (otherwise it will just return the original pixel value). If you apply a mode filter to an image that has blocks of similar colours pixels, it will tend to filter out any different values, leaving blocks of identical colour. It is a primitive form of *segmentation.*

All the filters above have the same signature as the `MedianFilter`.

A `RankFilter` can select any of the pixels, depending on the `rank` parameter. If you set the `rank` to 0 is will pick the lowest value, like a `MinFilter`, if you set it to 8 it will pick the highest value like `MaxFilter` (assuming the kernel is 3 by 3). If you set `rank` to 2 it will pick the third from lowest value. This can be used for very specific filters.

`RankFilter` has the following signature:

```
MedianFilter(size, rank)  # creates a filter object
```

- `size` the size of the kernel, as above.
- `rank` the value to use, an index into the ordered list of pixel values.

10.5 Defining your own filters

You can define your own filters using the `Color3DLUT`, `Kernel`, `Filter`, and `MultibandFilter` objects in the ImageFilter module. That is quite an advanced topic that we will not cover here.

11. Image compositing

Compositing images means taking elements from two or more different images and making them appear on the same image. These are mainly included in the ImageChops module. *Chops* is short for *channel operations*, because the functions work on the colour channels in the image.

Examples include:

- Adding a frame to an image.
- Combining two images so that elements of both appear in the same image.
- Combining an image with a background texture.
- Using a mask image to make the selected area of the main images brighter, darker etc.

There are quite a lot of different ways of compositing images. This chapter will give visual representations of example uses of each one. However, this is just a starting point, it is a good idea to experiment with them to see what effects you can achieve.

The chapter covers:

- Simple blending operations using alpha blending.
- More advanced blending modes that mix images in different ways. These are similar to the blending modes available with programs such a Gimp or Photoshop (although Photoshop has some more advanced modes that Pillow doesn't have).
- Logical blends, normally used for 1-bit images. They can be used to combine image masks in various ways.

ImageChops only supports images of type 'L' or 'RGB' (that is, 8-bit grey or 24-bit RGB images). It is generally best to ensure that the two images have the same pixel size, otherwise the larger image will be cropped.

11.1 Simple blending

In this first section we will look at three simple ways to combine two images:

- Using image transparency.
- Blending.
- Compositing.

11.1.1 Image transparency

In this example, we are going to take our boat image and combine it with a simple "picture frame" image, like this:

Image Frame Composite

This technique is very simple, and in fact it doesn't use the ImageChops module at all. It relies on the fact that the frame image is partly transparent. More precisely, the outer frame part is fully opaque, but the area inside the frame is fully transparent. When we place this image on top of the boat image, it looks like the boat is in a frame.

Here is the code:

```
from PIL import Image

image = Image.open('boat-small.jpg')
frame_image = Image.open('frame.png')
framed_image = image.copy()
framed_image.paste(frame_image, mask=frame_image)
```

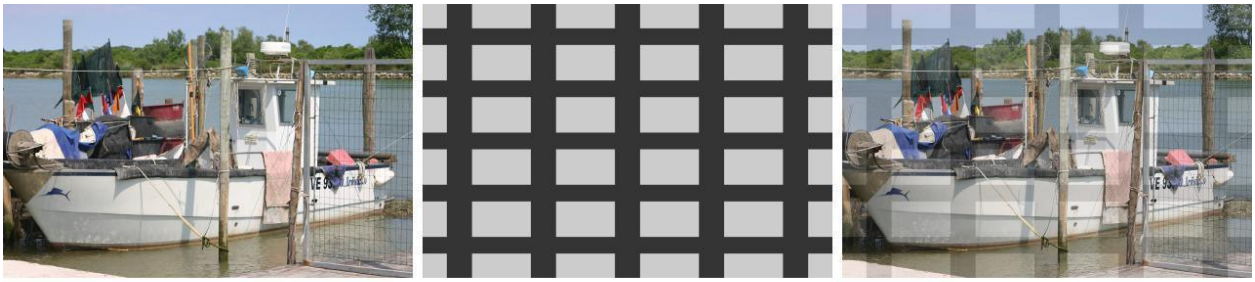
We open the boat image and the frame image. We create the output (`framed_image`) by duplicating the boat image, then pasting the frame image on top.

For this to work properly, the two images must be the same size. Of course, we can resize the images in code if necessary.

By default, `paste` will not take account of transparency. We need to pass the `frame_image` as a `mask` when we paste. This will force the paste operation to take account of the alpha channel.

11.1.2 ImageChops blend function

The ImageChops `blend` function performs a simple alpha blending between two images, like this:



In this example, we apply a blend factor of 0.2.

Both images are RGB (that is, they have no transparency channel). However, the `blend` function combines them *as if* the Grid image had an alpha value of 0.2. So the RGB value of the composite image is calculated like this:

```
Composite.r = Image.r * (1 - factor) + Grid.r * factor
Composite.g = Image.g * (1 - factor) + Grid.g * factor
Composite.b = Image.b * (1 - factor) + Grid.b * factor
```

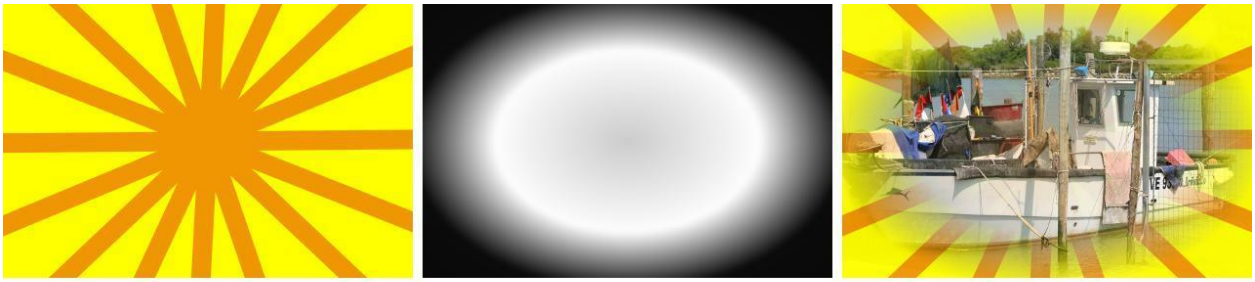
The code is fairly straightforward, we just open the two images and call the blend function. The grid image was created as a PNG file in Inkscape, which generally creates RGBA images, so we first convert the grid image to RGB.

```
from PIL import Image, ImageChops

image = Image.open('boat-small.jpg')
grid_image = Image.open('grid.png').convert('RGB')
composite_image = ImageChops.blend(image, grid_image, 0.2)
```

The order of the images is important. In effect, the function sets the transparency of the second image to 0.2 and applies it on top of the first image.

11.1.3 ImageChops composite function

The ImageChops `composite` function performs a masked alpha blending between two images, like this:



The `composite` function blends the boat image and the border image in varying amounts depending on the grey level of the mask image. Where the mask is white, the main image shows through, where the mask is black the border image shows. For grey values of the mask, the two images are blended according to the grey level.

```
from PIL import Image, ImageFilter, ImageDraw, ImageFont, ImageChops

image = Image.open('boat-small.jpg')
border_image = Image.open('border.png').convert('RGB')
mask_image = Image.open('vignette.png').convert('L')
composite_image = ImageChops.composite(image, border_image, mask_image)
```

The mask image must be a single-channel image. We ensure that by converting the mask to `L` mode (greyscale).

11.2 Blend modes

The ImageChops module provides various ways to blend two images. Each mode combines the two images, but they all work in different ways. Choosing a blend is essentially an artistic decision - pick the one that works best for the effect you are trying to achieve.

Blending works pixel by pixel. Each pixel of the result image is calculated by combining the equivalent pixels from the two source images.

So for example, if we add two images, A and B, to create an image R, then the pixel at position (x, y) is calculated as:

```
R(x, y) = A(x, y) + B(x, y)
```

If both images are greyscale images, then each pixel is a greyscale value from `0` to `255`, so we add them to create a greyscale result.

If both images are RGB, then we add each channel separately to create the result:

```
R.r(x, y) = A.r(x, y) + B.r(x, y)
R.g(x, y) = A.g(x, y) + B.g(x, y)
R.b(x, y) = A.b(x, y) + B.b(x, y)
```

If one image is RGB and one is greyscale, then the result will be an RGB image. The single channel of the greyscale image is combined with the r, g and b channels of the RGB image. In effect, the greyscale image is treated as if it were an RGB image with identical r, g and b channels.

For brevity, we will write this as:

```
R = A + B
```

With the understanding that this means "add every channel of every pair of corresponding pixels", as described above.

11.2.1 Addition

We can add two images, like this:

```
image = Image.open('boat-small.jpg')
gradient_image = Image.open('greygradient.png').convert('RGB')
composite_image = ImageChops.add(image, gradient_image)
```

We combine our boat image with a gradient image that goes from pure white to pure black in the horizontal direction.

The formula is:

```
R = min( A + B, 255 )
```

Here is the effect:

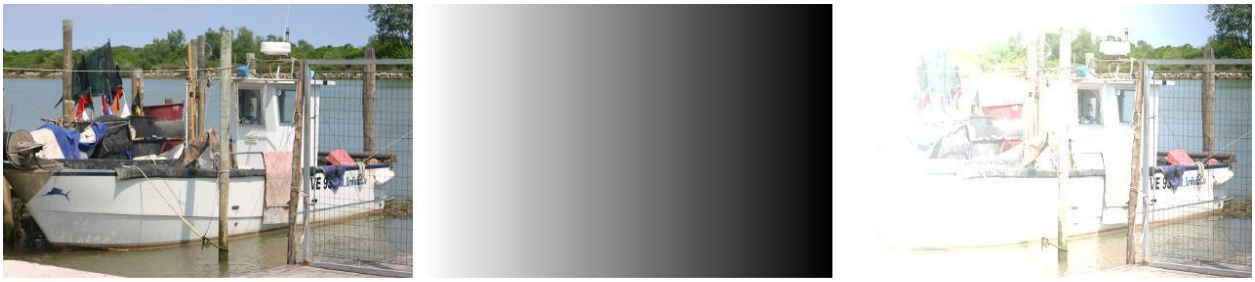


Image Gradient Composite

Since A, B and R are all byte quantities, they must have a value of between 0 and 255. If A was 100 and B was 200, the sum is 300. This is clamped to a value of 255, so R will be set to 255 in that case.

In areas where the gradient image is black (B=0), the boat image is not affected at all. Where the gradient is dark grey, the boat image is made a little lighter. Where the gradient is light grey, the boat image is made a lot lighter and saturates at pure white in places where both images are very bright. This is a good way to selectively lighten an image, but there is a loss of detail in the bright areas (similar to a traditional film camera being over-exposed).

It is possible to correct for this by using the optional `scale` and `offset` parameters in the `add` function:

```
composite_image = ImageChops.add(image, gradient_image, scale=2, offset=0)
```

This modifies the formula to:

```
R = min( (A + B)/scale + offset, 255 )
```

With a scale of 2, the sum of A and B is divided by two, so it never goes above the 255 limit. This is useful if you want to selectively lighten an image without the over-exposed effect in the previous example, like this:

The `add_modulo` function does something slightly different:

```
composite_image = ImageChops.add_modulo(image, gradient_image)
```

Here, the calculation is:

```
R = ( A + B ) module 256
```

This time, if the combination of the two values of greater than 255, the value “wraps around”. This avoids loss of detail, but it can result in some colours being inverted, which isn’t usually what you would want. Here is the effect:

11.2.2 Subtraction

We can subtract one image from another, like this:

```
image = Image.open('boat-small.jpg')
gradient_image = Image.open('greygradient.png').convert('RGB')
composite_image = ImageChops.subtract(image, gradient_image)
```

We combine our boat image with the same gradient image as before.

The formula is:

```
R = max( A - B, 0 )
```

Here is the effect:

Image Gradient Composite

Again A, B and R are all byte quantities with values of between 0 and 255. For pixels where A >= B, the result will always be in range. If B > A, the result will be negative, which is out of range, so the negative value is set to zero.

The effect is the opposite of the `add` function. In areas where the gradient is light, the result image is made darker than the original image. In areas where the gradient is very light and the image is already dark, the result value will be clamped to zero. This is similar to under-exposure in traditional photography.

The `subtract` function has optional `scale` and `offset` parameters, similar to `add`. There is also a `subtract_modulo` function (similar to `add_modulo`). These work in an analogous way to `add` functions.

There is also a `difference` function:

```
composite_image = ImageChops.difference(image, gradient_image)
```

This calculates:

```
R = abs( A - B )
```

This is the absolute value of the difference between the two images. In other words, for pixels where A > B the result is A - B, and for pixels where B > A the result is B - A. If they are equal, the result is zero of course. Here is the effect:

11.2.3 Lighter and darker

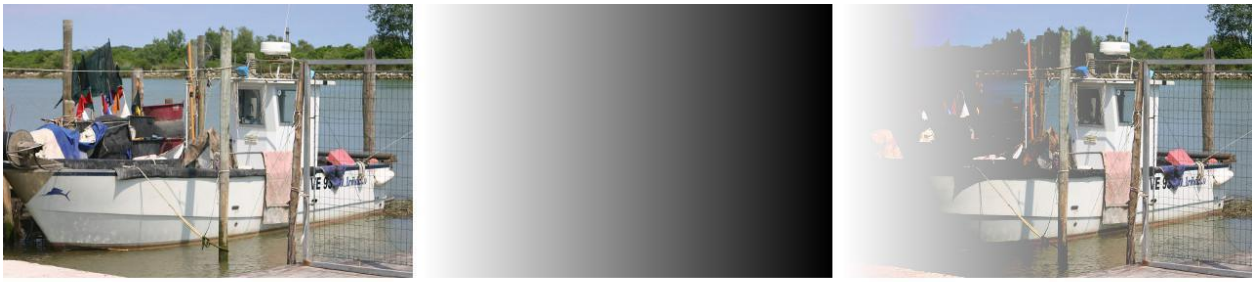
The `lighter` function selects the lightest value from each image:

```
image = Image.open('boat-small.jpg')
gradient_image = Image.open('greygradient.png').convert('RGB')
composite_image = ImageChops.lighter(image, gradient_image)
```

The formula is:

```
R = max( A, B )
```

It selects the highest value for each channel and each pixel. Since the formula is selecting a value from one or the other of the two images, the value is guaranteed to be in the range 0 to 255, so no clamping is required. Here is the effect:



The `darker` function selects the lowest value for each pixel, using the formula:

11.2.4 Multiply and screen

The `multiply` function multiplies the two images together:

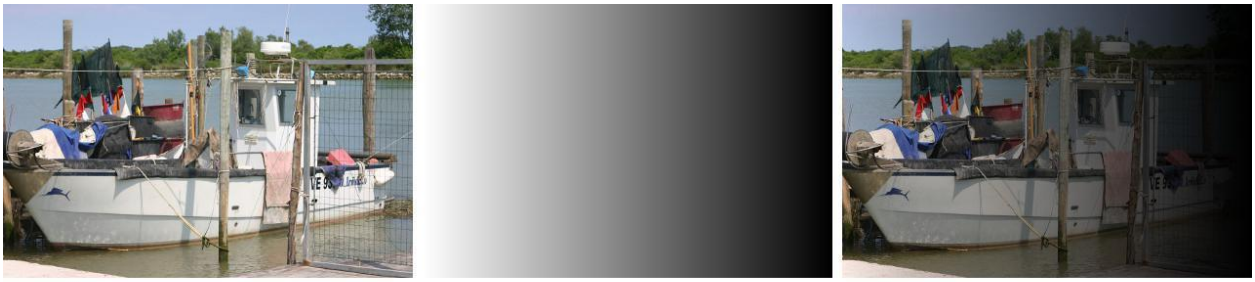
```
R = A * B / 255
```

Notice that, since A and B are both scaled in the range 0 to 255, when we multiply them together the result will be in the range 0 to 255*255. We need to divide the result by 255 to correct the scaling.

What this function does is to darken the image by an amount determined by the blend image B:

- Where B is white, the output image is identical to A.
- Where B is black, the output image is black.
- Where B is 70% white, the output image has its value reduced to 70%, and so on.

Here is the result:



The `screen` function does the opposite of multiply. It:

- Inverts A and B.
- Multiplies the inverted images.
- Inverts the result.

Here is the effect:

11.2.5 Other blend modes

ImageChops has three more blend modes.

The `overlay` function implements the Overlay algorithm (similar to the Overlay mode provided by Photoshop and similar products). It is a combination of the multiply and screen modes:

- If the image A value is < 128, the multiply mode is used, which creates a darker version of A depending on how dark B is.
- If the image A value is >= 128, the overlay mode is used, which creates a lighter version of A depending on how light B is.

The `hard_light` function implements the Hard Light algorithm (similar to the Hard Light mode provided by Photoshop and similar products). This is related to `overlay` because `overlay(A, B)` is equivalent to `hard_light(B, A)`, ie swapping the parameters.

The `soft_light` function implements the Soft Light algorithm (similar to the Soft Light mode provided by Photoshop and similar products). Soft Light is similar to Overlay, but it uses a linear interpolation between multiply and screen:

- If A is zero, the result is `multiply(A, B)`.
- If A is 255, the result is `screen(A, B)`.
- For intermediate values of A, the result is an interpolation between the two.

For example, if A is 64 (25% of full brightness), the result will be:

```
0.75*multiply(A, B) + 0.25*screen(A, B)
```

11.3 Logical combinations

The logical operators use OR, AND, and XOR to combine two images.

Logical operations normally apply to true and false values. Pillow treats black areas of the image as false, and white areas of the image as true.

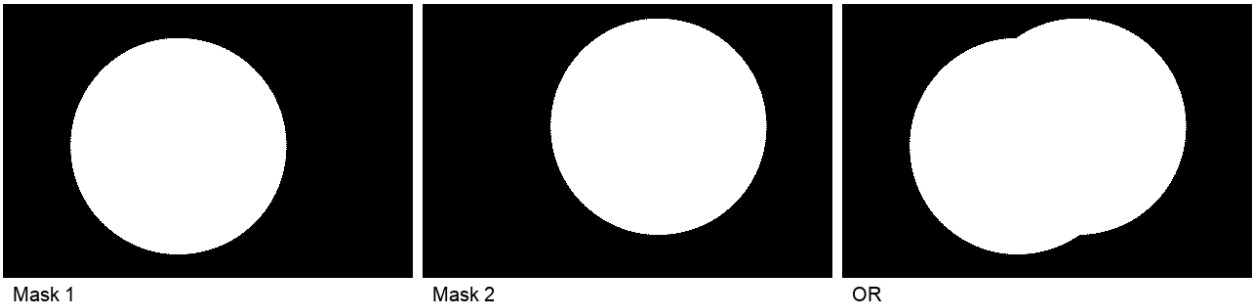
There is no real way to define what would happen if you tried to AND dark red with light yellow. Pillow avoids this issue by restricting the logical operators to 1-bit images. As we saw in the imaging section, in a 1-bit image each pixel is represented by a single bit of data, so the pixel can either be fully black (bit value 0) or fully white (bit value 1).

We can convert any image to 1-bit using the `convert` function with a format value of `'1'`. It is best to start with a black and white image, so you know exactly what you are going to get.

Here is the code to OR two images together. Each image (called *mask1* and *mask2*) is a white circle on a black background, but the circles are in different positions in the two images. Here is the code:

```
mask1 = Image.open('circle1.png').convert('1')
mask2 = Image.open('circle2.png').convert('1')
composite_image = ImageChops.logical_or(mask1, mask2)
```

We open the two images and convert them to 1-bit before combining them into a composite image. Here is the result:

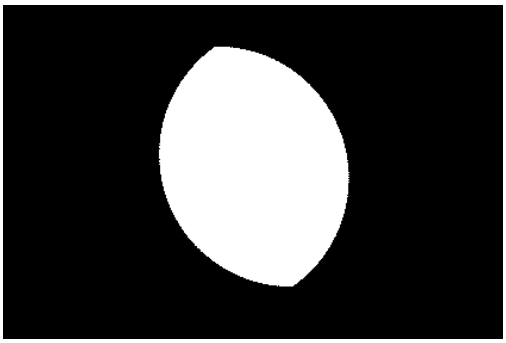


This shows the two masks and the result of applying the `logical_or` function. The result is:

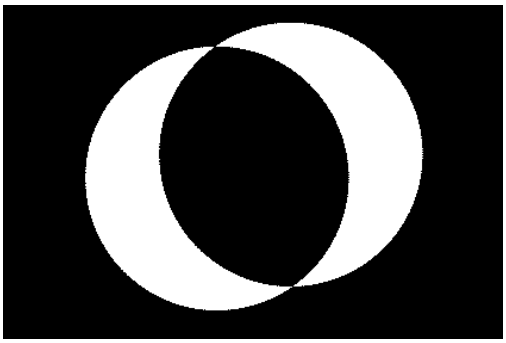
- White anywhere that either or both masks are white.
- Black anywhere both masks are black.

As the variable names suggest, logical operations are particularly useful for working with image masks. If you have two or more mask shapes defined as black and white images, you can combine them into more complex shapes.

The `logical_and` function creates an image that is white anywhere both images are white, black everywhere else:



The `logical_xor` function creates an image that is white anywhere that the two images are different (one is black and the other is white). The image is black anywhere the images are the same (either both black or both white):



Don't forget that you can also invert an image, using the ImageOps `invert` function. For a 1-bit image, this swaps black and white, essentially performing a logical NOT function.

12. Drawing on images

Pillow can draw basic shapes and text on an image. It can be used, for example, to annotate an image by drawing an outline around a key part of the image, or by adding a title or timestamp to the image.

The drawing features are quite basic - you can only fill or outline shapes using flat colours (not patterns or gradients), and the system doesn't support transforms such as scaling or rotating shapes. But they are good enough for automated annotations and similar.

It is also possible to be a little more creative, by combining shapes with some of the techniques we used in the *Image compositing* chapter.

12.1 Coordinate system

The coordinate system for drawing is the same as the coordinate system for pixels:

- The top left of the image has coordinates (0, 0).
- The x-coordinate increases as you move to the right. Each pixel is one unit in the coordinate system.
- The y-coordinate increases as you move down the image. Each pixel is one unit in the coordinate system.

This means that, for example, the image pixel that 10 pixels to the right and 20 pixels down from the top left of the image will have coordinates (10, 20).

This system is fixed. Unlike most vector drawing libraries, you cannot transform the coordinate system in any way.

12.2 Drawing shapes

To draw a shape, we must first create a `Draw` object, as shown in the code samples below. We can then draw a variety of shapes

12.2.1 Drawing rectangles

Here is the code to draw three rectangles on top of the normal boat image:

```
from PIL import Image, ImageDraw

image = Image.open('boat-small.jpg')

draw = ImageDraw.Draw(image)
draw.rectangle((20, 100, 80, 250), 'red')
draw.rectangle((150, 100, 250, 200), outline=(0, 128, 0), width=6)
draw.rectangle((300, 50, 350, 200), fill='white', outline='black', width=10)

image.save('imagedraw-rectangles.jpg')
```

Here is the image it produces:

First, we import the Image and ImageDraw modules and open the boat image in the usual way.

To prepare for drawing, we first create a `Draw` object:

```
draw = ImageDraw.Draw(image)
```

A `Draw` object can only draw on one image, and we need to pass that image in when we construct the object.

Here is how we draw a rectangle:

```
draw.rectangle((20, 100, 80, 250), 'red')
```

The first parameter is a sequence giving the coordinates of the corners of the rectangle, which are (20, 100) and (80, 250).

Wherever a `Draw` method takes a list of x, y values, you can use a sequence of values `(x1, y1, x2, y2)` or a sequence of pairs `((x1, y1), (x2, y2))`.

Many graphics libraries specify a rectangle in terms of a corner and the width and height. Pillow uses two corners. It amounts to the same thing, but remember that it doesn't work quite like other libraries you might have used.

The second parameter is the fill colour of the rectangle. As usual, the CSS colour name 'red' is automatically converted into an RGB value.

The code above also creates two more rectangles:

```
draw.rectangle((150, 100, 250, 200), outline=(0, 128, 0), width=6)
draw.rectangle((300, 50, 350, 200), fill='white', outline='black', width=10)
```

This time we have used named parameters, for clarity. We draw an unfilled rectangle with an outline of `(0, 128, 0)` which is a dark green colour, with a line width of 6 pixels. We also draw a rectangle that is filled with white and outlined in black, with a 10-pixel line width.

12.2.2 Drawing other shapes

In addition to drawing rectangles, the ImageDraw module can draw:

- Lines (and multi-lines).
- General polygons.
- Regular polygons.
- Ellipses (including circles).
- Arcs, chords and pie segments.

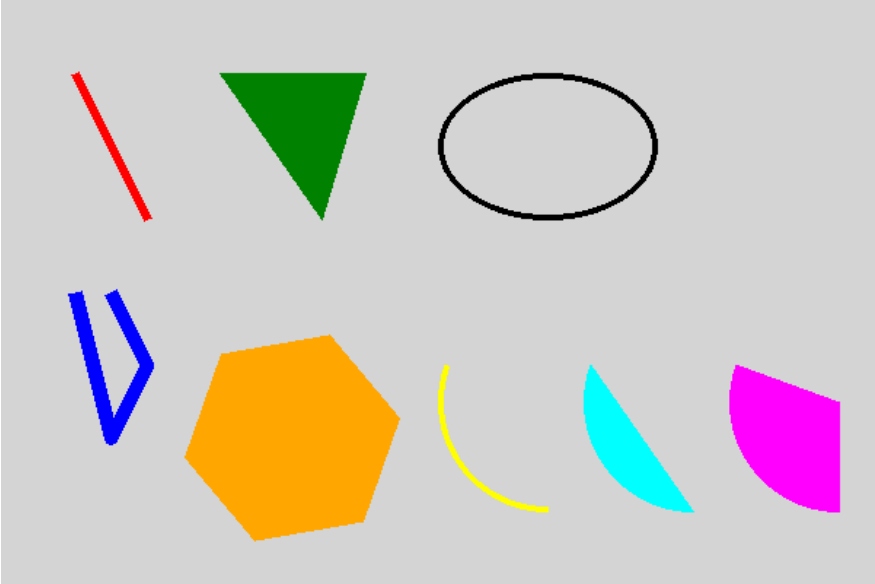
These all follow the same basic pattern - the drawing method supplies the parameters of the shape, and an optional `fill`, `outline` and `width` for drawing the shape. The process is similar to drawing rectangles, so we won't go through every function in a huge amount of detail. Instead, we will look at a code example and then pick out any specific points for each shape.

Here is the sample code:

```
image = Image.new('RGB', (600, 400), 'lightgray')

draw = ImageDraw.Draw(image)
draw.line((50, 50, 100, 150), fill='red', width=6)
draw.line((50, 200, 75, 300, 100, 250, 75, 200), fill='blue', width=10,
                joint='curve')
draw.polygon((150, 50, 220, 150, 250, 50), fill='green')
draw.regular_polygon((200, 300, 75), n_sides=6, rotation=10, fill='orange')
draw.ellipse((300, 50, 450, 150), outline='black', width=4)
```

```
draw.arc((300, 200, 450, 350), start=90, end=200, fill='yellow', width=4)
draw.chord((400, 200, 550, 350), start=90, end=200, fill='cyan')
draw.pieslice((500, 200, 650, 350), start=90, end=200, fill='magenta')

image.save('imagedraw-shapes.png')
```

Here is the image it produces:



Let's look at these shapes one by one.

```
draw.line((50, 50, 100, 150), fill='red', width=6)
draw.line((50, 200, 75, 300, 100, 250, 75, 200), fill='blue', width=10, joint='curve\
')
```

The first call to `line` here draws a red line from point `(50, 50)` to point `(100, 150)`.

The slightly odd thing about `line` is that you set its colour using the `fill` parameter rather than the `outline` parameter. `line` and `arc` are the only methods that do this. You still use the `width` parameter to set the width of the line.

The next call draws a blue multi-line from point `(50, 200)` to `(75, 300)` to `(100, 250)` to `(75, 200)`. This draws three connected lines (you could also think of it as an open polygon). We set the `joint` parameter to `'curve'` so that the lines join with rounded corners. Without this parameter, the lines are drawn as completely separate lines, without proper joints, which looks a bit ugly.

```
draw.polygon((150, 50, 220, 150, 250, 50), fill='green')
draw.regular_polygon((200, 300, 75), n_sides=6, rotation=10, fill='orange')
```

Calling `polygon` draws a polygon, in this case, a triangle with corners `(150, 50)`, `(220, 150)`, and `(250, 50)`. It works a bit like the multi-line example before, but it creates a closed polygon. We use `fill` to set the fill colour.

`regular_polygon` can be used to draw polygons that have all sides and angles equal. You could do this using `polygon`, of course, but `regular_polygon` will calculate the positions of the corners automatically.

The polygon size and position are defined by `(200, 300, 75)`, which means that the polygon fits into a circle with centre `(200, 300)` and radius 75. We set `n_sides` to 6, which creates a hexagon. We have also set `rotation` to 10. This means that the shape is rotated by 10 degrees, about its centre, in the counter-clockwise direction. Rotation is optional and defaults to zero.

```
draw.ellipse((300, 50, 450, 150), outline='black', width=4)
draw.arc((300, 200, 450, 350), start=90, end=200, fill='yellow', width=4)
draw.chord((400, 200, 550, 350), start=90, end=200, fill='cyan')
draw.pieslice((500, 200, 650, 350), start=90, end=200, fill='magenta')
```

`ellipse` draws an ellipse. It is defined by a bounding box, in the same way as `rectangle`. The ellipse exactly fits the bounding rectangle. To draw a circle, use a square bounding box.

The way `ellipse` is defined is not the same as the way the bounding circle is defined in `regular_polygon`.

The `arc` function works in a similar way to `ellipse`, but it only draws part of the circumference, from the `start` angle to the `end` angle. Angles are measured from the x-axis (3 o'clock), in degrees, in the clockwise direction. This means that the `start` angle of 90 corresponds to 6 o'clock, and the `end` angle of 200 corresponds to (approximately) 10 o'clock.

`arc` only draws the circumference, it doesn't fill the area so (similar to `line`) you must use the `fill` parameter to set the line colour.

`chord` is similar to `arc`, except that it joins the two ends of the arc with a straight line (a chord of the circle). This creates a *segment* of the circle.

`pieslice` is also similar to `arc`, but this time it joins the two ends of the arc back to the centre of the circle with straight lines. This creates a *sector* of the circle.

12.2.3 Points

The `point` method can be used to set one or more pixels to a given colour. For example:

```
draw.point((100, 200), fill='black')
draw.point((50, 70, 150, 140), fill='red')
```

The first example will set the image pixel `(100, 200)` to black. The second example will set both the pixels `(50, 70)` and `(150, 140)` to red.

A typical application of this might be to draw a small shape by setting groups of pixels to a particular colour. If you want to make a shape such as a circle or square, it is usually best to use the `rectangle` or `ellipse` methods, but for a non-standard shape, the `point` method can be useful.

Another application might be to fill part of an image with calculated colours, for example, a gradient or even a fractal image. Be aware that calling `point` many times to set a large number of pixels can be quite slow, so you might be better using a more efficient pixel access method described elsewhere in this book.

12.3 Handling text

Pillow provides functionality for handling text via the ImaegFont module (for selecting a font) and the ImageDraw module.

In this section, we will mainly look at creating and positioning text in Latin based scripts, as used by many European languages. Pillow is capable of handling other writing systems, including those with right-to-left and vertical text directions, but we will not cover those here.

12.3.1 Drawing simple text

There are two stages to drawing simple text:

- Selecting a font.
- Drawing a text string using that font.

A *font* is a particular typeface. It defines the shapes of letters, digits, punctuation, and other symbols in a particular style.

Here is how we might load a font called Arial:

```
font = ImageFont.truetype('arial.ttf', 100)
```

There are many different formats for storing fonts. TrueType is a popular format and most systems will have a set of TrueType fonts available (which will almost certainly include the Arial font, a standard basic font).

We can either provide the full path to a TrueType font, or we can just supply the filename and Pillow will search the standard system fonts folder(s).

This call also sets the size of the text. Font size is measured in *points*, which are a traditional printing unit that is equal to 1/72nd of an inch (about 0.35 mm). But Pillow assumes an image resolution of 1 point per pixel, so 100 also gives the size of the font in pixels. This font will be approximately 100 pixels high. This is approximate because the exact height of the font is left to the font designer. There will be some variation between different fonts, even with the same nominal size.

Having chosen a font, we can draw the font using:

```
draw.text((100, 50), 'ABCdefg', font=font, fill='red')
```

This draws the text `'ABCdefg'` at position `(100, 50)`, using the previously defined font and setting the font colour to red. Here is the result:

The blue circle is positioned at `(100, 50)`. As you can see, the text is positioned so that the top-left of the text is at the specified position. We can change this using the *anchor* as we will see later.

Here is the full code:

```
from PIL import Image, ImageDraw, ImageFont

image = Image.new('RGB', (600, 200), 'lightgrey')
font = ImageFont.truetype('arial.ttf', 100)

draw = ImageDraw.Draw(image)
draw.ellipse((95, 45, 105, 55), fill='blue')
draw.text((100, 50), 'ABCdefg', font=font, fill='red')

image.save('imagedraw-text.png')
```

12.3.2 Font and text metrics

We use metrics to determine the size of the text, so we can position and align it properly. We can use:

- Font metrics, that depend only on the font and font size.
- Text metrics, that depend on the actual text content, as well as the font and font size.

Here is an example of the font metrics:

The *baseline* is the line that the text naturally sits on. We also have:

- The *ascent* is the maximum height of a normal character above the baseline. For most fonts, the height of the capital letters defines the ascent, although the ascent also usually includes a small extra margin.
- The *descent* is the maximum amount certain characters descend below the baseline. This is normally defined by the tails of characters like g or y.
- The *middle* is halfway between the ascent and the descent.

There is no guarantee that every character will stay within the bounds of the ascent and descent. Ordinary letters and numbers will *usually* stay within the bounds, but certain characters, such as square brackets, might be slightly larger in certain fonts. You might even find a font where the normal characters stray outside the range, but this is usually only "novelty" fonts.

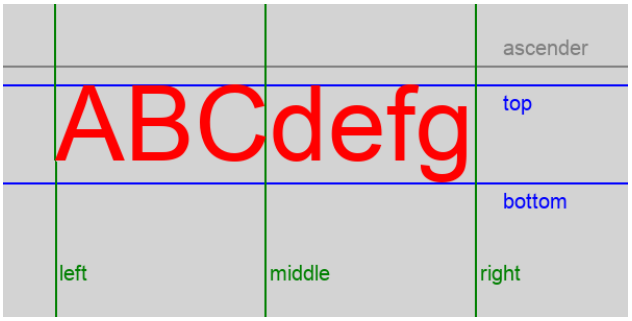
You can find these metrics line this:

```
ascent, descent = font.getmetrics()  # font is an ImageFont object
```

The middle is halfway between the ascent and descent.

Notice that these values depend only on the font. Even if the string you are displaying has no capitals or descender, for example, the string "aeiou", the ascent and descent will still be the same.

Text metrics depend on the actual string. Here is an example of text metrics:



The text metrics (top, bottom, left, right) indicate the exact rectangle that is marked on the page. The middle is halfway between left and right.

There are two functions for calculating the text metrics:

```
xoffset, yoffset = font.getoffset(text)
width, height = font.getsize(text)
```

Here, once again, `font` is the ImageFont object. `text` is the text string you are measuring.

Assuming you are using the left ascender as the anchor (see later), and the text is position at `(x, y)` then:

- `left` is at `x + xoffset`.
- `right` is at `x + width`.
- `top` is at `y + yoffset`.
- `bottom` is at `ascender + height`

Notice that the height of the text is measured from the ascender, rather than the top of the text. It is a quirk of Pillow, but not a major issue so long as you remember it.

12.3.3 Anchoring

When you draw text, you can choose the *anchor* you want to use. In other words, if you place your text at `(x, y)`, which part of the text is over that exact point?

You have a choice:

- In the horizontal direction you can choose to position the text so that the left (l), middle (m), or right (r) point of the string is aligned with x.
- In the vertical direction, you can choose to position the text so that the ascender (a), top (t), middle (m), baseline(s), or bottom (r) point of the string is aligned with y.

Pillow uses a two-character string to specify the anchor. For example:

- `'la'` aligns the left ascender with `(x, y)`. This is the default.
- `'ls'` aligns the left baseline with `(x, y)`. This is the most natural option.
- `'mt'` aligns the horizontal-middle and top with `(x, y)`.
- `'rm'` aligns the right and vertical-middle with `(x, y)`.
- And so on - there are 15 combinations altogether.

The anchor is set using the `anchor` parameter of the `text` method, like this:

```
draw.text((x, y), 'Align ls', anchor='ls', font=font, fill='red')
```

Here are some examples. In each case the blue dot marks the position `(x, y)` of the string:

12.3.4 Drawing multiline text

ImageDraw contains special functions for drawing multiline text. It automatically handles simple line spacing and alignment, but it doesn't have any advanced features such as automatic line splitting or text justification.

Here is how we can draw multiline text:

```
draw.multiline_text((x, y), 'Multiline\nText', anchor='ls', font=font, fill='red')
```

Points to note here are:

- We use the `multiline_text` method rather than the `text` method.
- The string uses newlines (`'\n'`) to indicate line ends.
- The anchor `'ls'` refers to the baseline of the first line of the text (see the blue dot in the example below).

Here is how we find the exact bounding box of the text:

```
box = draw.multiline_textbbox((x, y), 'Multiline\nText', anchor='ls', font=font)
```

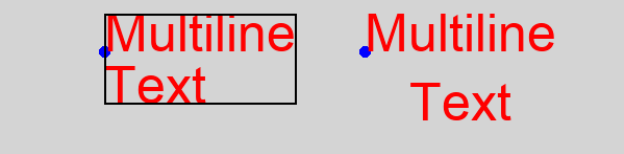
Alternatively, you could use `multiline_textsize`, which gives a similar result but with a bounding box that includes the top line ascender and bottom line descender.

`multiline_text` has various options, here are a couple of useful ones:

```
draw.multiline_text((x, y), 'Multiline\nText', anchor='ls', align='center', spacing=\
20, font=font, fill='red')
```

- `align` is used to align the individual lines within the text block. Possible values are `'left'` (the default), `'center'`, and `'right'`.
- `spacing` controls the spacing between the lines. It is measured in pixels.

Here are the two examples above:



And here is the full code:

```
from PIL import Image, ImageDraw, ImageFont

image = Image.new('RGB', (600, 150), 'lightgrey')

font = ImageFont.truetype('arial.ttf', 50)

draw = ImageDraw.Draw(image)

x, y = 100, 50
draw.ellipse((x-5, y-5, x+5, y+5), fill='blue')
draw.multiline_text((x, y), 'Multiline\nText', anchor='ls', font=font, fill='red')

box = draw.multiline_textbbox((x, y), 'Multiline\nText', anchor='ls', font=font)
draw.rectangle(box, outline='black', width=2)

x, y = 350, 50
draw.ellipse((x-5, y-5, x+5, y+5), fill='blue')
draw.multiline_text((x, y), 'Multiline\nText', anchor='ls', align='center', spacing=\
20, font=font, fill='red')

image.save('imagedraw-multilinetext.png')
```

12.4 Paths

Paths allow us to define shapes that we can draw later. They can be useful if you want to use the same shape multiple times, or even if you want to draw the same shape that has been transformed in various ways.

Pillow has a very simple implementation of paths. A path consists of a list of `(x, y)` points, joined by straight lines. Paths cannot include curves, although you can simulate a curve by drawing lots of short lines.

Once you have a path you can:

- Transform the points in the path, using a 2D matrix.
- Compact the path (removing any points that are very close together to improve drawing efficiency).
- Pass the points through a function. This can be used, for example, to truncate the shape.

The final path can be drawn by passing it into the `line` or `polygon` method.

12.4.1 Drawing a path

To draw our path, we first create a list of points, using this code:

```
count = 201

def curve(x):
    y = (x-100)**2/100
    return x, y

points = [curve(t) for t in range(0, count, 10)]
```

The curve function plots the curve:

```
y = (x-100)**2/100
```

This is a curve in x-squared, which is U-shaped (a parabola in mathematical terms). The `points` list contains `(x, y)` values for x from 0 to 200 in steps of 10.

Here is the code to plot the points:

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

draw = ImageDraw.Draw(image)

path = ImagePath.Path(points)
path.compact()

draw.line(path, fill='blue', width=4)

image.save('imagedraw-path.png')
```

In this code, we first create an `image`, and a `draw` object attached to that image.

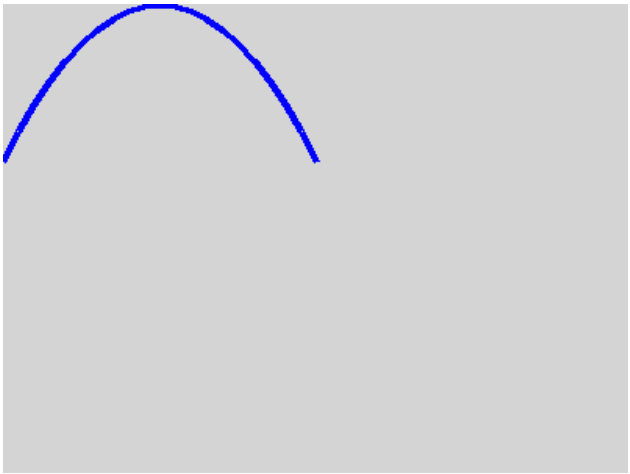
This code creates a Path object based on the points list, and compacts (usually a good idea in case any points are very close together and might not draw smoothly):

```
path = ImagePath.Path(points)
path.compact()
```

Finally, we draw the points using `line` in the usual way (except that we pass a Path object rather than a list of points):

```
draw.line(path, fill='blue', width=4)
```

Here is the result. It is a standard x-squared shape, but upside down because in Pillow the y coordinate increases as you move down the image:



In this case, we didn't really need to use a Path object. We could just have passed the `points` list into the `line` method. We will see the advantages of Path in the next couple of sections.

12.4.2 Transforming paths

We can apply a matrix transformation to a Path. A matrix allows us to:

- Translate the shape to a new position.
- Scale the shape.
- Rotate the shape.
- Skew the shape.
- Mirror the shape in the x or y directions.
- Any combination of the above in a single operation.

The transformation matrix has 6 values:

```
[a, b, c
 d, e, f]
```

Here are some examples:

```
[1, 0, 0    # Unit transform (doesn't change the path)
 0, 1, 0]

[1, 0, tx    # Translate path by tx in x direction
 0, 1, ty]   #                   ty in y direction

s = math.sin(s)
c = math.cos(a)
[c, -s, 0    # rotate path by angle a
 s, c, 0]
```

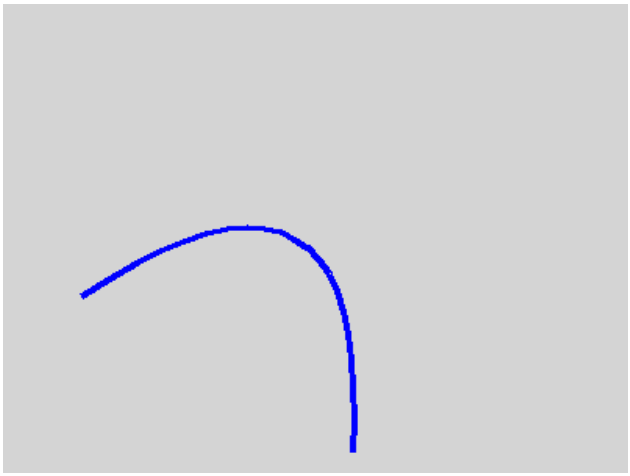
If you want to learn more about matrix transformation, see any good maths book/website.

Here is some code that rotates the path by pi/6 radians (30 degrees), and translates it by (100, 100):

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

draw = ImageDraw.Draw(image)

c = math.cos(math.pi/6)
s = math.sin(math.pi/6)
path = ImagePath.Path(points)
path.transform([c, -s, 100, s, c, 100])
path.compact()
```

```
draw.line(path, fill='blue', width=4)

image.save('imagedraw-transformpath.png')
```

We create the path, transform it, then compact it as before. Here is the result:



12.4.3 Mapping points

ImagePath also has a `map` method. Similar to the built-in `map` function in Python, it accepts a function and applies it to every point in the path.

The mapping function must accept two values, `x` and `y`, and return a tuple `(x, y)` with the new position of that point.

Here is a simple example of a suitable function:

```
def sketch(x, y):
    return x + random.randrange(6), y + random.randrange(6)
```

This function adds a small random amount to the `x` and `y` values of the point, which displaces it by a random amount. When this function is applied to all the points in the path, each point is displaced by a different random amount. This has the effect of making the curve waver slightly giving it a hand-drawn appearance. Here is the code:

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

draw = ImageDraw.Draw(image)

path = ImagePath.Path(points)
path.transform([c, -s, 100, s, c, 100])
path.compact()
path.map(sketch)

draw.line(path, fill='blue', width=4)

image.save('imagedraw-sketchpath.png')
```

And here is the result:

13. Accessing pixel data

Sometimes you might want to read or set image pixels individually. This can be used for:

- Creating special fills or gradients.
- Creating calculated images such as fractals.
- Performing specialised pixel calculations.
- Applying special effects at the pixel level.

This chapter will look at how we can access pixel data, with a couple of simple examples:

- Processing an image to create a simple effect.
- Creating a new image filled with a mathematically generated pattern.

13.1 Processing an image

Here is a simple example where we process every pixel in the image. We will shift the green and blue channels by a few pixels in the x-direction, to simulate the effect of a badly registered image. This is purely done as an example, it has no practical use except as a slightly unusual artistic effect. Here is what the final image looks like:

Here is the code:

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

pixels = image.load()
for x in range(image.size[0]-20):    # image.size[0] is the image width
    for y in range(image.size[1]):   # image.size[1] is the image height
        r, _, _ = pixels[x, y]
        _, g, _ = pixels[x+10, y]
        _, _, b = pixels[x+20, y]
        pixels[x, y] = (r, g, b)

image.close()
image.save('deregister.jpg')
```

The main stages are:

- Open an existing image `'boat.jpg'`.
- Call `load` to load the image data into a `PixelAccess` object that we store as `pixels`.
- Loop over the pixels, changing their RGB value.
- Close the image (see below).
- Save the image.

In Pillow, you don't normally need to explicitly call `Image.close` to close an image, but it is good practice to close the image after using pixel access to modify pixels. This ensures that the image data is fully updated.

The key part of this code is the `load` method. This loads a `PixelAccess` object for the image, which then gets stored in `pixels`. This is a two-dimensional array of pixel colours. We can read a pixel like this:

```
r, g, b = pixels[x, y]
```

Notice that we use the syntax `[x, y]` to index the pixel (rather than `[x][y]` that you might use to access a 2D list). Since the image is an RGB image, the result is a tuple of r, g, and b values. We unpack this into 3 separate variables. Each colour value will be an integer in the range 0 to 255.

We can modify a pixel like this:

```
pixels[x, y] = (r, g, b)
```

This sets the pixel at `(x, y)` to the tuple value `(r, g, b)`.

In the actual code, we loop over every `x` and `y` value in the image (except that we only use `x` values up to `width - 20`). Here is the important part of the code:

```
r, _, _ = pixels[x, y]
_, g, _ = pixels[x+10, y]
_, _, b = pixels[x+20, y]
pixels[x, y] = (r, g, b)
```

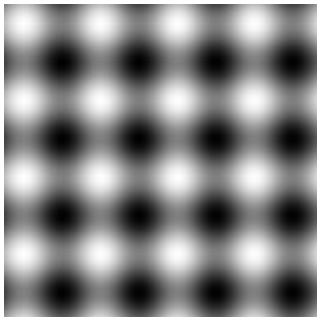
We unpack the current pixel, storing the red value in `r`. Unpacking to _ discards the green and blue values. In a similar way we get the green value from the pixel located at `(x+10, y)`, and the blue value from the pixel located at `(x+20, y)`.

We set the pixel at `(x, y)` with the red, green and blue values from different parts of the image, giving the misregistered effect.

This effect could have been done differently. We could have split the image into 3 separated bands using `Image.split`, then cropped the 10 pixels off the left edge of the green plane (and 20 off the blue), then joined the bands together again using `Image.merge`. That might have been more efficient, but this example is only really intended to illustrate pixel access.

13.2 Creating an image

In this next example, we will create a new image rather than modifying an existing image. We will fill the image with a simple mathematical pattern of dots, based on the sine function, like this:



Here is the code:

```
image = Image.new('L', (256, 256), 'black')

pixels = image.load()
for x in range(256):
    for y in range(256):
        pixels[x, y] = 128+int(63*math.sin(x/10) + 63*math.sin(y/10))

image.close()
image.save('sine-pattern.jpg')
```

This time we create a new image - a greyscale image, 256 pixels square, initialise to black.

In the main loop, we calculate a pattern based on the function:

```
math.sin(x) + math.sin(y)
```

This pattern produces diffuse black and white dots. However, the values this function produces are in the range of -2.0 to +2.0. We multiply the values by 63 and add 128 to produce values that are closer to the range 0 to 255. We also divide `x` and `y` by 10 to make the dots bigger. The actual pattern here is not too important, you can easily replace it with your own function.

We could have used an RGB image for this, of course. In that case, we could have needed to calculate an `(r, g, b)` tuple to assign to `pixels[x, y]` in the loop, just like the previous example.

13.3 Performance

You should be aware that accessing pixels individually is relatively slow. If you are implementing an algorithm that involves reading or writing a large number of pixels, expect it to be considerably slower than calling one of Pillow's built-in functions. That is because most of Pillow's low-level functions are written in C, which is quite a lot faster than Python.

Having said that, performance isn't terrible. On a modern PC, writing every pixel of a medium-size image (6 million pixels) will take less than a second. Performance can become more of an issue when:

- Processing very large images.
- Batch processing a large number of images, for example applying an effect to all the images in a particular folder.
- Processing images interactively in a GUI application, where users typically expect an instant response.

In general:

- Use Pillow built-in functions if at all possible.
- Consider converting the image to a NumPy array to perform complex processes, especially if you are processing every pixel in the same way. See the chapter *Integrating Pillow with other libraries*.

14. Integrating Pillow with other libraries

It is sometimes useful to be able to exchange image information between Pillow and other libraries so that you can use or process the data differently.

14.1 NumPy integration

This is probably the most important integration, because:

- NumPy can store and process large amounts of data very efficiently.
- Many other Python packages are compatible with NumPy. Once you have your data in NumPy format, you can do lots of things with it.

NumPy stores data in the form of a multi-dimensional array of numerical values. The values can have different types and sizes. For example, if a NumPy array stores integer data, you can choose to store 8-bit, 16-bit, 32-but or 64-bit data, which can be signed or unsigned. This compares to a Python integer, which is an object and can store integers of any size.

To integrate with Pillow, we normally use unsigned 8-bit integers.

The code below:

* Reads an image file using Pillow.

- Converts the image to a NumPy array.
- Modifies the NumPy array.
- Converts the array back into a Pillow image.
- Writes it to file.

```
from PIL import Image
import numpy as np

image = Image.open('boat-small.jpg')

image_array = np.array(image)
print('shape', image_array.shape)

image_array = 255 - image_array
image_array[100:200, 150:350] = np.array([255, 128, 0])

out_image = Image.fromarray(image_array)
out_image.save('numpy-image.jpg')
```

14.1.1 Converting a Pillow image to Numpy

Here is the code that reads an image into Pillow and converts it to a NumPy array.

```
from PIL import Image
import numpy as np

image = Image.open('boat-small.jpg')

image_array = np.array(image)
print('shape', image_array.shape)
```

The first thing to notice is that we have to import NumPy. We import `numpy` *as* `np`. This means that we can use `np` rather than `numpy` in our code whenever we need to use a NumPy function. It is entirely optional, but most people who use NumPy tend to do this. Even the official NumPy documentation does it.

If you haven't already, you must also *install* NumPy. This is slightly less straightforward than most Python packages because NumPy contains C code that needs to be compiled. It is best to use Anaconda for this. Anaconda is both a package manager (like pip) and a *distribution*. This means that it contains versions of the packages that have already been build for various platforms (such as Windows and various flavours of Linux). It will install NumPy and most other things you might need, and it will just work. Visit anaconda.com for more information, it will save you a lot of heartache.

After the imports, we open our boat image in the usual way, creating a Pillow Image object stored in `image`.

We create a NumPy array using the `np.array` function, applying it to the Pillow image. It creates a NumPy array that we store as `image_array`.

The `array` function is a standard NumPy function. You might be wondering, how does NumPy know how to convert an Image object into a NumPy array? Well, Image objects implement the *array interface* that NumPy also supports. NumPy doesn't specifically know about Image objects, but it knows how to get data from an array object.

14.1.2 Image data in a NumPy array

Our NumPy array now contains the image data. To verify this, we print the shape of the array, `image_array.shape`. Here is the result:

```
(280, 420, 3)
```

This tells us that the array is 3-dimensional (because there are 3 elements in the `shape` tuple), and it has a size 280 by 420 by 3.

Our boat image is 420 by 280 pixels, so why is the NumPy shape different? Well, NumPy stores data in row, column order. This is similar to how spreadsheets work. If we displayed our NumPy array as a spreadsheet:

- It would have 280 rows.
- Each row has 420 columns.
- Each cell would contain 3 values, the RGB components of the pixel.

So the image has a height of 280 and a width of 420, which matches the original image, it is just that the width and height are swapped because we are using row, column order.

In fact, we need to swap `x` and `y` everywhere when we are using NumPy for images. For example:

```
image_array(10, 20)
```

Would access the pixel at `(x, y)` position (20, 10).

14.1.3 Modifying the NumPy image

This isn't a book about NumPy, so we go into too much detail here, but here is some code to modify the image (just to prove that we have really converted the data to NumPy and back again):

```
image_array = 255 - image_array
```

If you aren't familiar with NumPy, when we perform mathematical operating on the array, NumPy will actually perform that operation on every single element. So this code performs the operation `x = 255 -x` on every value in the array. So every colour component of every column of every row. This has the effect of inverting the image.

We also perform a second operation:

```
image_array[100:200, 150:350] = np.array([255, 128, 0])
```

This accesses a slice of the data. It touches every pixel that has:

- A row value between 100 and 199.
- And a column value between 150 and 349.

That is a rectangular area of the image. For each pixel in that area, it replaces the value with `[255, 128, 0]`, which is an orange colour.

Notice again we are working with rows and columns. In terms of image coordinates:

- The rectangle top left is at (150, 100).
- It is 200 pixels wide and 100 pixels high.

14.1.4 Converting a NumPy array to a Pillow image

```
out_image = Image.fromarray(image_array)
out_image.save('numpy-image.jpg')
```

The Image `fromarray` method converts the NumPy array to an Image object. Again, Pillow doesn't specifically know about NumPy arrays, it will work on any object that supports the array interface.

We save the image. Here is what it looks like:

III Reference

15. Pillow colour representation

Pillow stores colours as Python tuples. Each colour channel is stored as a value between 0 and 255, where zero represents none of that colour, and 255 represents the maximum amount of the colour. The length of the tuple is equal to the number of channels in the colour space. For example:

- An RGB colour has the form `(128, 0, 255)`, which would represent a colour of 50% red. 0% green, 100% blue.
- An RGBA colour has the form `(64, 255, 0, 128)`, which would represent a colour of 25% red. 100% green, 0% blue, with an alpha (transparency) value of 50%.
- An L colour has the form `(128)`, which would represent a colour of 50% grey.

Colours can also be represented as strings. The string values are converted to tuples. Whenever a Pillow accepts a colour value, you can normally use either a tuple or a string. The supported formats are listed below. See the *Computer colour* chapter for more information on the different colour spaces.

15.1 Hexadecimal colour specifiers

The form `#rrggbb` treats the string as three hexadecimal numbers. So for the value `#804020`:

- The red value is hex 80 (decimal 128).
- The green value is hex 40 (decimal 64).
- The blue value is hex 20 (decimal 32).

The shorter form `#rgb` just uses a single digit for each colour value. This is first expanded by repeating each digit. So `#812` is expanded to `#881122`, and then converted to decimal `(136, 17, 37)` as before.

The form `#rrggbbaa` represents an RGBA quantity. So for the value `#00FF0040`:

- The red value is hex 00 (decimal 0).
- The green value is hex FF (decimal 255).
- The blue value is hex 0 (decimal 0).
- The alpha value is hex 40 (decimal 64).

The form `#rgba` can also be used.

15.2 RGB functions

The form `rgb(128, 10, 50)` represents the RGB values directly. It simply translates into the tuple `(128, 10, 50)`.

The similar form `rgb(10%, 50%, 0%)` expresses RGB values as percentages of the maximum 255, so:

- The red value is 10% (decimal 26).
- The green value is 50% (decimal 128).
- The blue value is hex 0% (decimal 0).

15.3 HSL functions

The form `hsl(hue, saturation% lightness%)` represents an HSL colour. For example, `hsl(180, 100% 50%)` is interpreted like this:

- The hue value is interpreted as an angle (in degrees) on the colour circle. An angle of 180 degrees is halfway round the colour circle, giving a cyan colour.
- The saturation of 100% gives a fully saturated cyan colour.
- The lightness of 50% is halfway between black and white, which gives a pure cyan.

The result is a fully saturated pure cyan that has an RGB value of `(0, 255, 255)`, ie a mix of full green and blue.

Although the colour is specified using HSL values, the resulting colour is an RGB tuple. The colour is automatically converted to RGB.

15.4 HSV functions

The form `hsv(hue, saturation% value%)` represents an HSV colour. It works in exactly the same way as HSL but using the HSV colour model. Hue and saturation are the same as in the HSL case, but V (value) component is calculated differently from the L component in HSL.

The form `hsb(hue, saturation% brightness%)` represents an HSB colour. HSV and HSB are different names for the same colour space, so `hsb` and `hsv` create identical results.

15.5 Named colours

Finally, you can supply named colours, such as `orange` in the example code. This will be converted into an RGB tuple representing orange. It accepts the named colours defined in CSS for use by web browsers. Colour names are case insensitive.

15.6 Example

This example uses the function `ImageColor.getrgb()` to convert a various colour strings to RGB tuples.

You don't normally need to use the `getrgb()` function, you can usually just pass a string as a colour value, and Pillow will convert it automatically.

```
from PIL import ImageColor

color = ImageColor.getrgb('#804020')
print(color)                        # (128, 64, 32)

color = ImageColor.getrgb('#800')
print(color)                        # (136, 0, 0)

color = ImageColor.getrgb('#00FF0040')
print(color)                        # (0, 255, 0, 64))

color = ImageColor.getrgb('rgb(128, 10, 50)')
print(color)                        # (128, 10, 50)

color = ImageColor.getrgb('rgb(10%, 50%, 0%)')
print(color)                        # (26, 128, 0)

color = ImageColor.getrgb('hsl(180, 100%, 50%)')
print(color)                        # (0, 255, 255)

color = ImageColor.getrgb('hsv(180, 100%, 100%)')
print(color)                        # (0, 255, 255)

color = ImageColor.getrgb('orange')
print(color)                        # (255, 165, 0)
```

15.7 Image modes

The image *mode* is a string value that specifies both the colour space and the number of bits per pixel. Here is a complete list of all supported modes.

These are 8-bit modes, where each colour component uses 8 bits per pixel

- `'RGB'` - 24-bit RGB.

- `'RGBA'` - 32-bit RGB plus alpha (8 bits per colour and 8-bit alpha).
- `'L'` - 8-bit greyscale.
- `'CMYK'` - 32-bit cyan, magenta, yellow, black.
- `'YCbCr'` - 24-bit luminance (Y) and 2 chrominance channels (Cr and Cb), used for video.
- `'LAB'` - 24-bit L_*a*_b colour space.
- `'HSV'` - 24-bit Hue, Saturation, Value colour space.
- `P` - 8-bit palette mode. Each pixel is an 8-bit value that maps into a table of RGB values.
- `1` - 1-bit data. Each pixel is either fully black or fully white. This data is stored in `L` format (8 bits per pixel) but only values `0` and `1` are permitted. When 1-bit data is stored in a file, t is usually stored as 8 pixels per byte, but the bit ordering depends on the file format used.

These are 32-bit modes supported:

- `I`- each pixel is stored as a 32-bit signed integer.
- `F`- each pixel is stored as a 32-bit signed float.

The remaining modes have limited support (they do not necessarily work with all Pillow functions). These modes have additional colour components:

- `'LA'` - 16-bit greyscale with alpha. This is similar to `L` but with an extra component representing the alpha component.
- `'PA'` - 16-bit palette with alpha. This is similar to `P` but with an extra component representing the alpha component.
- `'RGBX'` - 32-bit RGB with padding. This is similar to `RGB` but with an extra padding component. Each pixel occupies 4 bytes but the extra byte is unused.

There are various ways to process these modes. One way is to separate the image into bands using the `Image.getbands` method we looked at in the *Image class*chapter. The other way is to convert the image to one of the standard modes. For example, we can convert an `'LA'` image to an `'RGBA'` image like this:

```
la_image = Image.new('LA', (200, 200), (64, 128))
print('LA bands', la_image.getbands())      # ('L', 'A')
print('Pixel', la_image.getpixel((0, 0)))   # (64, 128)

converted_image = la_image.convert(mode="RGBA")
print('Converted bands', converted_image.getbands())  # ('R', 'G', 'B', 'A')
print('Pixel', converted_image.getpixel((0, 0)))      # (64, 64, 64, 128)
```

As a test, we first create the `'LA'` filled with colour (64, 128). To verify this, we print the bands (which are “L” and “A”) and the first pixel value (which is (64, 128)).

We then convert the image to `'RGBA'` mode. As expected the image now has bands "R", "G", "B", and "A". Also, the pixel value is (64, 64, 64, 128), which gives a grey level of 64 and an alpha of 128. Perfect! We can now process this image normally.

There are also a couple of modes with *premultiplied alpha*:

- `'RGBa'` - 32-bit RGB with premultiplied alpha.
- `'La'` - 16-bit greyscale with premultiplied alpha.

The *pre-multiplied alpha* modes are similar to normal alpha modes, except that the colour values have been multiplied by the alpha value. So for example, consider this colour:

```
RBBA = (255, 128, 0, 64)
```

The alpha value 128 corresponds to an opacity of 64/255 (about 0.251). To represent this as an RGBa colour, we multiply the RGB values by 0.251, giving:

```
RBBa = (64, 32, 0, 64)
```

The motivation here is that the pre-multiplied RGB values can be added to the background without any further calculation. This helps to speed up the processing of transparent images in some circumstances. With modern processors, the advantage is less significant than it used to be, so pre-multiplied alpha is rarely used these days.

We can handle these modes, once again, by converting them to `'RGBA'` mode. For example:

```
# RGBa image
rgba_image = Image.new('RGBa', (200, 200), (64, 32, 0, 64))
print('RGBa bands', rgba_image.getbands())   # ('R', 'G', 'B', 'a')
print('Pixel', rgba_image.getpixel((0, 0)))  # (64, 32, 0, 64)

converted_image = rgba_image.convert(mode="RGBA")
print('Converted bands', converted_image.getbands())  # ('R', 'G', 'B', 'A')
print('Pixel', converted_image.getpixel((0, 0)))      # (255, 127, 0, 64)
```

This time we start with a test image in mode `'RGBa'` filled with (64, 32, 0, 64).

We convert this to `'RGBA'` mode. Pillow very helpfully corrects the pre-multiplied RGB values so that the image is now filled with (255, 127, 0, 64). This is quite similar to the original `'RGBA'` value of (255, 128, 0, 64), it differs slightly due to rounding errors.

Finally, there are some modes that provide limited support non-standard pixel formats:

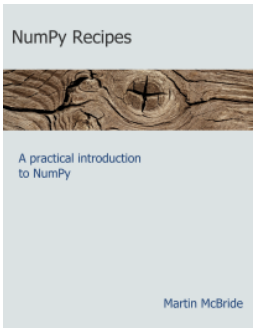
- `'I;16'` 16-bit unsigned integer pixels.

- `'I;16L'` 16-bit little-endian unsigned integer pixels.
- `'I;16B'` 16-bit big-endian unsigned integer pixels.
- `'I;16N'` 16-bit native endian unsigned integer pixels.
- `'BGR;15'` 15-bit reversed true colour.
- `'BGR;16'` 16-bit reversed true colour.
- `'BGR;24'` 24-bit reversed true colour.
- `'BGR;32'` 32-bit reversed true colour.

More books from this author

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

Numpy Recipes

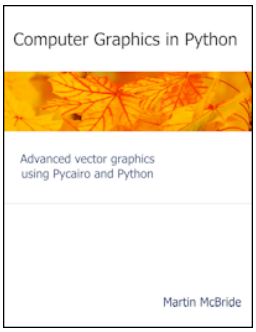


NumPy Recipes takes practical approach to the basics of NumPy

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.

Computer Graphics in Python with Pycairo



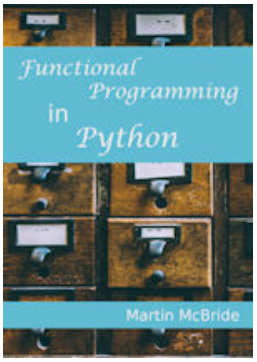
The Pycairo library is a Python graphics library. This book covers the library in detail, with lots of practical code examples.

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.

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.