Python Imaging Library Overview

PIL 1.1.3 | March 12, 2002 | Fredrik Lundh, Matthew Ellis

Introduction

The **Python Imaging Library** adds image processing capabilities to your Python interpreter.

This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities.

The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

Let's look at a few possible uses of this library:

Image Archives

The Python Imaging Library is ideal for for image archival and batch processing applications. You can use the library to create thumbnails, convert between file formats, print images, etc.

The current version identifies and reads a large number of formats. Write support is intentionally restricted to the most commonly used interchange and presentation formats.

Image Display

The current release includes Tk *PhotoImage* and *BitmapImage* interfaces, as well as a Windows *DIB* interface that can be used with PythonWin. For X and Mac displays, you can use Jack Jansen's *img* library.

For debugging, there's also a *show* method in the Unix version which calls *xv* to display the image.

Image Processing

The library contains some basic image processing functionality, including point operations, filtering with a set of built-in convolution kernels, and colour space conversions.

The library also supports image resizing, rotation and arbitrary affine transforms.

There's a histogram method allowing you to pull some statistics out of an image. This can be used for automatic contrast enhancement, and for global statistical analysis.

Tutorial

Using the Image Class

The most important class in the Python Imaging Library is the **Image** class, defined in the module with the same name. You can create instances of this class in several ways; either by loading images from files, processing other images, or creating images from scratch.

To load an image from a file, use the **open** function in the **Image** module.

```
>>> import Image
>>> im = Image.open("lena.ppm")
```

If successful, this function returns an **Image** object. You can now use instance attributes to examine the file contents.

```
>>> print im.format, im.size, im.mode
PPM (512, 512) RGB
```

The **format** attribute identifies the source of an image. If the image was not read from a file, it is set to None. The **size** attribute is a 2-tuple containing width and height (in pixels). The **mode** attribute defines the number and names of the bands in the image, and also the pixel type and depth. Common modes are "L" (luminance) for greyscale images, "RGB" for true colour images, and "CMYK" for pre-press images.

If the file cannot be opened, an **IOError** exception is raised.

Once you have an instance of the **Image** class, you can use the methods defined by this class to process and manipulate the image. For example, let's display the image we just loaded:

```
    >>> im.show()
```

(The standard version of **show** is not very efficient, since it saves the image to a temporary file and calls the **xv** utility to display the image. If you don't have **xv** installed, it won't even work. When it does work though, it is very handy for debugging and tests.)

The following sections provide an overview of the different functions provided in this library.

Reading and Writing Images

The Python Imaging Library supports a wide variety of image file formats. To read files from disk, use the **open** function in the **Image** module. You don't have to know the file format to open a file. The library automatically determines the format based on the contents of the file.

To save a file, use the **save** method of the **Image** class. When saving files, the name becomes important. Unless you specify the format, the library uses the filename extension to discover which file storage format to use.

Example: Convert files to JPEG

```
import os, sys
import Image

for infile in sys.argv[1:]:
    outfile = os.path.splitext(infile)[0] + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print "cannot convert", infile
```

A second argument can be supplied to the **save** method which explicitly specifies a file format. If you use a non-standard extension, you must always specify the format this way:

Example: Create JPEG Thumbnails

```
import os, sys
import Image

for infile in sys.argv[1:]:
    outfile = os.path.splitext(infile)[0] + ".thumbnail"
    if infile != outfile:
        try:
            im = Image.open(infile)
            im.thumbnail((128, 128))
            im.save(outfile, "JPEG")
        except IOError:
            print "cannot create thumbnail for", infile
```

It is important to note is that the library doesn't decode or load the raster data unless it really has to. When you open a file, the file header is read to determine the file format and extract things like mode, size, and other properties required to decode the file, but the rest of the file is not processed until later.

This means that opening an image file is a fast operation, which is independent of the file size and compression type. Here's a simple script to quickly identify a set of image files:

Example: Identify Image Files

```
import sys
import Image

for infile in sys.argv[1:]:
    try:
        im = Image.open(infile)
        print infile, im.format, "%dx%d" % im.size, im.mode
    except IOError:
        pass
```

Cutting, Pasting and Merging Images

The **Image** class contains methods allowing you to manipulate regions within an image. To extract a sub-rectangle from an image, use the **crop** method.

Example: Copying a subrectangle from an image

```
box = (100, 100, 400, 400)
region = im.crop(box)
```

The region is defined by a 4-tuple, where coordinates are (left, upper, right, lower). The Python Imaging Library uses a coordinate system with (0, 0) in the upper left corner. Also note that coordinates refer to positions between the pixels, so the region in the above example is exactly 300x300 pixels.

The region could now be processed in a certain manner and pasted back.

Example: Processing a subrectangle, and pasting it back

```
region = region.transpose(Image.ROTATE_180)
im.paste(region, box)
```

When pasting regions back, the size of the region must match the given region exactly. In addition, the region cannot extend outside the image. However, the modes of the original image and the region do not need to match. If they don't, the region is automatically converted before being pasted (see the section on *Colour Transforms* below for details).

Here's an additional example:

Example: Rolling an image

```
def roll(image, delta):
    "Roll an image sideways"

    xsize, ysize = image.size

    delta = delta % xsize
    if delta == 0: return image

    part1 = image.crop((0, 0, delta, ysize))
    part2 = image.crop((delta, 0, xsize, ysize))
    image.paste(part2, (0, 0, xsize-delta, ysize))
    image.paste(part1, (xsize-delta, 0, xsize, ysize))

    return image
```

For more advanced tricks, the paste method can also take a transparency mask as an optional argument. In this mask, the value 255 indicates that the pasted image is opaque in that position (that is, the pasted image should be used as is). The value 0 means that the pasted image is completely transparent. Values in-between indicate different levels of transparency.

The Python Imaging Library also allows you to work with the individual bands of an multi-band image, such as an RGB image. The split method creates a set of new images, each containing one band from the original multi-band image. The merge function takes a mode and a tuple of images, and combines them into a new image. The following sample swaps the three bands of an RGB image:

Example: Splitting and merging bands

```
r, g, b = im.split()
im = Image.merge("RGB", (b, g, r))
```

Geometrical Transforms

The **Image** class contains methods to **resize** and **rotate** an image. The former takes a tuple giving the new size, the latter the angle in degrees counter-clockwise.

Example: Simple geometry transforms

```
out = im.resize((128, 128))
out = im.rotate(45) # degrees counter-clockwise
```

To rotate the image in 90 degree steps, you can either use the **rotate** method or the **transpose** method. The latter can also be used to flip an image around its horizontal or vertical axis.

Example: Transposing an image

```
out = im.transpose(Image.FLIP_LEFT_RIGHT)
out = im.transpose(Image.FLIP_TOP_BOTTOM)
out = im.transpose(Image.ROTATE_90)
out = im.transpose(Image.ROTATE_180)
out = im.transpose(Image.ROTATE_270)
```

There's no difference in performance or result between **transpose(ROTATE)** and corresponding **rotate** operations.

A more general form of image transformations can be carried out via the **transform** method. See the reference section for details.

Colour Transforms

The Python Imaging Library allows you to convert images between different pixel representations using the convert function.

Example: Converting between modes

```
    im = Image.open("lena.ppm").convert("L")
```

The library supports transformations between each supported mode and the "L" and "RGB" modes. To convert between other modes, you may have to use an intermediate image (typically an "RGB" image).

Image Enhancement

The Python Imaging Library provides a number of methods and modules that can be used to enhance images.

Filters

The **ImageFilter** module contains a number of pre-defined enhancement filters that can be used with the **filter** method.

Example: Applying filters

```
import ImageFilter
out = im.filter(ImageFilter.DETAIL)
```

Point Operations

The **point** method can be used to translate the pixel values of an image (e.g. image contrast manipulation). In most cases, a function object expecting one argument can be passed to the this method. Each pixel is processed according to that function:

Example: Applying point transforms

```
# multiply each pixel by 1.2
out = im.point(lambda i: i * 1.2)
```

Using the above technique, you can quickly apply any simple expression to an image. You can also combine the **point** and **paste** methods to selectively modify an image:

Example: Processing individual bands

```
# split the image into individual bands
source = im.split()

R, G, B = 0, 1, 2

# select regions where red is less than 100
mask = source[R].point(lambda i: i  100 and 255)

# process the green band
out = source[G].point(lambda i: i * 0.7)

# paste the processed band back, but only where red was  100
source[G].paste(out, None, mask)

# build a new multiband image
im = Image.merge(im.mode, source)
```

Note the syntax used to create the mask:

```
    imout = im.point(lambda i: expression and 255)
```

Python only evaluates the portion of a logical expression as is necessary to determine the outcome, and returns the last value examined as the result of the expression. So if the expression above is false (0), Python does not look at the second operand, and thus returns 0. Otherwise, it returns 255.

Enhancement

For more advanced image enhancement, use the classes in the **ImageEnhance** module. Once created from an image, an enhancement object can be used to quickly try out different settings.

You can adjust contrast, brightness, colour balance and sharpness in this way.

Example: Enhancing images

```
import ImageEnhance

enh = ImageEnhance.Contrast(im)
enh.enhance(1.3).show("30% more contrast")
```

Image Sequences

The Python Imaging Library contains some basic support for image sequences (also called *animation* formats). Supported sequence formats include FLI/FLC, GIF, and a few experimental formats. TIFF files can also contain more than one frame.

When you open a sequence file, PIL automatically loads the first frame in the sequence. You can use the **seek** and **tell** methods to move between different frames:

Example: Reading sequences

```
import Image

im = Image.open("animation.gif")
im.seek(1) # skip to the second frame

try:
    while 1:
        im.seek(im.tell()+1)
        # do something to im
except EOFError:
    pass # end of sequence
```

As seen in this example, you'll get an **EOFError** exception when the sequence ends.

Note that most drivers in the current version of the library only allow you to seek to the *next* frame (as in the above example). To rewind the file, you may have to reopen it.

The following iterator class lets you to use the for-statement to loop over the sequence:

Example: A sequence iterator class

```
class ImageSequence:
    def __init__(self, im):
        self.im = im
    def __getitem__(self, ix):
        try:
            if ix:
                self.im.seek(ix)
            return self.im
        except EOFError:
            raise IndexError # end of sequence

for frame in ImageSequence(im):
    # ...do something to frame...
```

Postscript Printing

The Python Imaging Library includes functions to print images, text and graphics on Postscript printers. Here's a simple example:

Example: Drawing Postscript

```
import Image
import PSDraw

im = Image.open("lena.ppm")
title = "lena"
box = (1*72, 2*72, 7*72, 10*72) # in points

ps = PSDraw.PSDraw() # default is sys.stdout
ps.begin_document(title)

# draw the image (75 dpi)
ps.image(box, im, 75)
ps.rectangle(box)

# draw centered title
ps.setfont("HelveticaNarrow-Bold", 36)
w, h, b = ps.textsize(title)
ps.text((4*72-w/2, 1*72-h), title)

ps.end_document()
```

More on Reading Images

As described earlier, the open function of the **Image** module is used to open an image file. In most cases, you simply pass it the filename as an argument:

```
im = Image.open("lena.ppm")
```

If everything goes well, the result is an **Image** object. Otherwise, an **IOError** exception is raised.

You can use a file-like object instead of the filename. The object must implement **read**, **seek** and **tell** methods, and be opened in binary mode.

Example: Reading from an open file

```
fp = open("lena.ppm", "rb")
im = Image.open(fp)
```

To read an image from string data, use the **StringIO** class:

Example: Reading from a string

```
import StringIO

im = Image.open(StringIO.StringIO(buffer))
```

Note that the library rewinds the file (using **seek(0)**) before reading the image header. In addition, seek will also be used when the image data is read (by the load method). If the image file is embedded in a larger file, such as a tar file, you can use the **ContainerIO** or **TarIO** modules to access it.

Example: Reading from a tar archive

```
import TarIO

fp = TarIO.TarIO("Imaging.tar", "Imaging/test/lena.ppm")
im = Image.open(fp)
```

Controlling the Decoder

Some decoders allow you to manipulate the image while reading it from a file. This can often be used to speed up decoding when creating thumbnails (when speed is usually more important than quality) and printing to a monochrome laser printer (when only a greyscale version of the image is needed).

The **draft** method manipulates an opened but not yet loaded image so it as closely as possible matches the given mode and size. This is done by reconfiguring the image decoder.

Example: Reading in draft mode

```
im = Image.open(file)
print "original =", im.mode, im.size

im.draft("L", (100, 100))
print "draft =", im.mode, im.size
```

This prints something like:

```
original = RGB (512, 512)
draft = L (128, 128)
```

Note that the resulting image may not exactly match the requested mode and size. To make sure that the image is not larger than the given size, use the **thumbnail** method instead.

Concepts

The Python Imaging Library handles *raster images*, that is, rectangles of pixel data.

Bands

An image can consist of one or more bands of data. The Python Imaging Library allows you to store several bands in a single image, provided they all have the same dimensions and depth.

To get the number and names of bands in an image, use the **getbands** method.

Mode

The mode of an image defines the type and depth of a pixel in the image. The current release supports the following standard modes:

- 1 (1-bit pixels, black and white, stored as 8-bit pixels)
- L (8-bit pixels, black and white)
- P (8-bit pixels, mapped to any other mode using a colour palette)
- RGB (3x8-bit pixels, true colour)
- RGBA (4x8-bit pixels, true colour with transparency mask)
- CMYK (4x8-bit pixels, colour separation)
- YCbCr (3x8-bit pixels, colour video format)
- I (32-bit integer pixels)
- F (32-bit floating point pixels)

PIL also supports a few special modes, including RGBX (true colour with padding) and RGBa (true colour with premultiplied alpha).

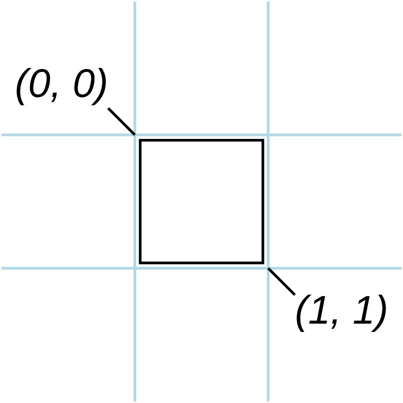
You can read the mode of an image through the **mode** attribute. This is a string containing one of the above values.

Size

You can read the image size through the **size** attribute. This is a 2-tuple, containing the horizontal and vertical size in pixels.

Coordinate System

The Python Imaging Library uses a Cartesian pixel coordinate system, with (0,0) in the upper left corner. Note that the coordinates refer to the implied pixel corners; the centre of a pixel addressed as (0, 0) actually lies at (0.5, 0.5):



Coordinates are usually passed to the library as 2-tuples (x, y). Rectangles are represented as 4-tuples, with the upper left corner given first. For example, a rectangle covering all of an 800x600 pixel image is written as (0, 0, 800, 600).

Palette

The palette mode ("P") uses a colour palette to define the actual colour for each pixel.

Info

You can attach auxiliary information to an image using the **info** attribute. This is a dictionary object.

How such information is handled when loading and saving image files is up to the file format handler (see the chapter on *Image File Formats*).

Filters

For geometry operations that may map multiple input pixels to a single output pixel, the Python Imaging Library provides four different resampling **filters**.

- **NEAREST**. Pick the nearest pixel from the input image. Ignore all other input pixels.
- **BILINEAR**. Use linear interpolation over a 2x2 environment in the input image. Note that in the current version of PIL, this filter uses a fixed input environment when downsampling.
- **BICUBIC**. Use cubic interpolation over a 4x4 environment in the input image. Note that in the current version of PIL, this filter uses a fixed input environment when downsampling.
- **ANTIALIAS**. (New in PIL 1.1.3). Calculate the output pixel value using a high-quality resampling filter (a truncated sinc) on all pixels that may contribute to the output value. In the current version of PIL, this filter can only be used with the **resize** and **thumbnail** methods.

Note that in the current version of PIL, the ANTIALIAS filter is the only filter that behaves properly when downsampling (that is, when converting a large image to a small one). The BILINEAR and BICUBIC filters use a fixed input environment, and are best used for scale-preserving geometric transforms and upsamping.

The Image Module

The **Image** module provides a class with the same name which is used to represent a PIL image. The module also provides a number of factory functions, including functions to load images from files, and to create new images.

Examples

Example: Open, rotate, and display an image

```
import Image
im = Image.open("bride.jpg")
im.rotate(45).show()
```

Example: Create thumbnails

```
import glob

for infile in glob.glob("*.jpg"):
    file, ext = os.splitext(infile)
    im = Image.open(infile)
    im.thumbnail((128, 128), Image.ANTIALIAS)
    im.save(file + ".thumbnail", "JPEG")
```

Functions

new

Image.new(mode, size) => *image*

Image.new(mode, size, color) => *image*

> Creates a new image with the given mode and size. Size is given as a 2-tuple. The colour is given as a single value for single-band images, and a tuple for multi-band images (with one value for each band). If the colour argument is omitted, the image is filled with black. If the colour is None, the image is not initialised.

open

Image.open(infile) => *image*

Image.open(infile, mode) => *image*

> Opens and identifies the given image file. This is a lazy operation; the actual image data is not read from the file until you try to process the data (or call the **load** method). If the mode argument is given, it must be "r".

You can use either a string (representing the filename) or a file object. In the latter case, the file object must implement **read**, **seek**, and **tell** methods, and be opened in binary mode.

blend

Image.blend(image1, image2, alpha) => *image*

Creates a new image by interpolating between the given images, using a constant alpha. Both images must have the same size and mode.

```
out = image1 * (1.0 - alpha) + image2 * alpha
```

If alpha is 0.0, a copy of the first image is returned. If alpha is 1.0, a copy of the second image is returned. There are no restrictions on the alpha value. If necessary, the result is clipped to fit into the allowed output range.

composite

Image.composite(image1, image2, mask) => *image*

Creates a new image by interpolating between the given images, using the mask as alpha. The mask image can have mode "1", "L", or "RGBA". All images must be the same size.

eval

Image.eval(function, image) => *image*

Applies the function (which should take one argument) to each pixel in the given image. If the image has more than one band, the same function is applied to each band. Note that the function is evaluated once for each possible pixel value, so you cannot use random components or other generators.

fromstring

Image.fromstring(mode, size, data) => *image*

Creates an image memory from pixel data in a string, using the standard "raw" decoder.

Image.fromstring(mode, size, data, decoder, parameters) => *image*

Same, but allows you to use any pixel decoder supported by PIL. For more information on available decoders, see the section *Writing Your Own File Decoder*.

Note that this function decodes pixel data, not entire images. If you have an entire image in a string, wrap it in a **StringIO** object, and use **open** to load it.

merge

Image.merge(mode, bands) => *image*

Creates a new image from a number of single band images. The bands are given as a tuple or list of images, one for each band described by the mode. All bands must have the same size.

Methods

An instance of the **Image** class has the following methods. Unless otherwise stated, all methods return a new instance of the **Image** class, holding the resulting image.

convert

im.convert(mode) => *image*

Returns a converted copy of an image. For the "P" mode, this translates pixels through the palette. If mode is omitted, a mode is chosen so that all information in the image and the palette can be represented without a palette.

The current version supports all possible conversions between "L", "RGB" and "CMYK."

When translating a colour image to black and white (mode "L"), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

When translating a greyscale image into a bilevel image (mode "1"), all non-zero values are set to 255 (white). To use other thresholds, use the **point** method.

im.convert(mode, matrix) => *image*

Converts an "RGB" image to "L" or "RGB" using a conversion matrix. The matrix is a 4- or 16-tuple.

The following example converts an RGB image (linearly calibrated according to ITU-R 709, using the D65 luminant) to the CIE XYZ colour space:

Example: Convert RGB to XYZ

```
rgb2xyz = (
    0.412453, 0.357580, 0.180423, 0,
    0.212671, 0.715160, 0.072169, 0,
    0.019334, 0.119193, 0.950227, 0 )
out = im.convert("RGB", rgb2xyz)
```

copy

im.copy() => *image*

Copies the image. Use this method if you wish to paste things into an image, but still retain the original.

crop

im.crop(box) => *image*

Returns a rectangular region from the current image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate.

This is a lazy operation. Changes to the source image may or may not be reflected in the cropped image. To break the connection, call the **load** method on the cropped copy.

draft

im.draft(mode, size)

Configures the image file loader so it returns a version of the image that as closely as possible matches the given mode and size. For example, you can use this method to convert a colour JPEG to greyscale while loading it, or to extract a 128x192 version from a PCD file.

Note that this method modifies the Image object in place. If the image has already been loaded, this method has no effect.

filter

im.filter(filter) => *image*

Returns a copy of an image filtered by the given filter. For a list of available filters, see the **ImageFilter** module.

fromstring

im.fromstring(data)

im.fromstring(data, decoder, parameters)

Same as the **fromstring** function, but loads data into the current image.

getbands

im.getbands() => *tuple of strings*

Returns a tuple containing the name of each band. For example, **getbands** on an RGB image returns ("R", "G", "B").

getbbox

im.getbbox() => *4-tuple or None*

Calculates the bounding box of the non-zero regions in the image. The bounding box is returned as a 4-tuple defining the left, upper, right, and lower pixel coordinate. If the image is completely empty, this method returns None.

getdata

im.getdata() => *sequence*

Returns the contents of an image as a sequence object containing pixel values. The sequence object is flattened, so that values for line one follow directly after the values of line zero, and so on.

Note that the sequence object returned by this method is an internal PIL data type, which only supports certain sequence operations. To convert it to an ordinary sequence (e.g. for printing), use **list(im.getdata())**.

getextrema

im.getextrema() => *2-tuple*

Returns a 2-tuple containing the minimum and maximum values of the image. In the current version of PIL, this is only applicable to single-band images.

getpixel

im.getpixel(xy) => *value or tuple*

Returns the pixel at the given position. If the image is a multi-layer image, this method returns a tuple.

histogram

im.histogram() => *list*

Returns a histogram for the image. The histogram is returned as a list of pixel counts, one for each pixel value in the source image. If the image has more than one band, the histograms for all bands are concatenated (for example, the histogram for an "RGB" image contains 768 values).

A bilevel image (mode "1") is treated as a greyscale ("L") image by this method.

im.histogram(mask) => *list*

Returns a histogram for those parts of the image where the mask image is non-zero. The mask image must have the same size as the image, and be either a bi-level image (mode "1") or a greyscale image ("L").

load

im.load()

Allocates storage for the image and loads it from the file (or from the source, for lazy operations). In normal cases, you don't need to call this method, since the Image class automatically loads an opened image when it is accessed for the first time.

offset

im.offset(xoffset, yoffset) => *image*

(Deprecated) Returns a copy of the image where the data has been offset by the given distances. Data wraps around the edges. If yoffset is omitted, it is assumed to be equal to xoffset.

This method is deprecated. New code should use the **offset** function in the **ImageChops** module.

paste

im.paste(image, box)

Pastes another image into this image. The box argument is either a 2-tuple giving the upper left corner, a 4-tuple defining the left, upper, right, and lower pixel coordinate, or None (same as (0, 0)). If a 4-tuple is given, the size of the pasted image must match the size of the region.

If the modes don't match, the pasted image is converted to the mode of this image (see the convert method for details).

im.paste(colour, box)

Same as above, but fills the region with a single colour. The colour is given as a single numerical value for single-band images, and a tuple for multi-band images.

im.paste(image, box, mask)

Same as above, but updates only the regions indicated by the mask. You can use either "1", "L" or "RGBA" images (in the latter case, the alpha band is used as mask). Where the mask is 255, the given image is copied as is. Where the mask is 0, the current value is preserved. Intermediate values can be used for transparency effects.

Note that if you paste an "RGBA" image, the alpha band is ignored. You can work around this by using the same image as both source image and mask.

im.paste(colour, box, mask)

Same as above, but fills the region indicated by the mask with a single colour.

point

im.point(table) => *image*

im.point(function) image => *image*

Returns a copy of the image where each pixel has been mapped through the given table. The table should contains 256 values per band in the image. If a function is used instead, it should take a single argument. The function is called once for each possible pixel value, and the resulting table is applied to all bands of the image.

If the image has mode "I" (integer) or "F" (floating point), you must use a function, and it must have the following format:

```
argument * scale + offset
```

Example: Map floating point images

```
out = im.point(lambda i: i * 1.2 + 10)
```

You can leave out either the **scale** or the **offset**.

im.point(table, mode) => *image*

im.point(function, mode) => *image*

Map the image through table, and convert it on fly. In the current version of PIL , this can only be used to convert "L" and "P" images to "1" in one step, e.g. to threshold an image.

putalpha

im.putalpha(band)

Copies the given band to the alpha layer of the current image. The image must be an "RGBA" image, and the band must be either "L" or "1".

putdata

im.putdata(data)

im.putdata(data, scale, offset)

Copy pixel values from a sequence object into the image, starting at the upper left corner (0, 0). The scale and offset values are used to adjust the sequence values:

```
pixel = value * scale + offset
```

If the scale is omitted, it defaults to 1.0. If the offset is omitted, it defaults to 0.0.

putpalette

im.putpalette(sequence)

Attach a palette to a "P" or "L" image. The palette sequence should contain 768 integer values, where each group of three values represent the red, green, and blue values for the corresponding pixel index. Instead of an integer sequence, you can use an 8-bit string.

putpixel

im.putpixel(xy, colour)

Modifies the pixel at the given position. The colour is given as a single numerical value for single-band images, and a tuple for multi-band images.

Note that this method is relatively slow. For more extensive changes, use **paste** or the **ImageDraw** module instead.

resize

im.resize(size) => *image*

im.resize(size, filter) => *image*

Returns a resized copy of an image. The size argument gives the requested size in pixels, as a 2-tuple: (**width**, **height**).

The filter argument can be one of **NEAREST** (use nearest neighbour), **BILINEAR** (linear interpolation in a 2x2 environment), **BICUBIC** (cubic spline interpolation in a 4x4 environment), or **ANTIALIAS** (a high-quality downsampling filter). If omitted, or if the image has mode "1" or "P", it is set to **NEAREST**.

rotate

im.rotate(angle) => *image*

im.rotate(angle, filter) => *image*

Returns a copy of an image rotated the given number of degrees counter clockwise around its centre.

The filter argument can be one of **NEAREST** (use nearest neighbour), **BILINEAR** (linear interpolation in a 2x2 environment), or **BICUBIC** (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode "1" or "P", it is set to **NEAREST**.

save

im.save(outfile, options)

im.save(outfile, format, options)

Saves the image under the given filename. If format is omitted, the format is determined from the filename extension, if possible. This method returns None.

Keyword options can be used to provide additional instructions to the writer. If a writer doesn't recognise an option, it is silently ignored. The available options are described later in this handbook.

You can use a file object instead of a filename. In this case, you must always specify the format. The file object must implement the **seek**, **tell**, and **write** methods, and be opened in binary mode.

seek

im.seek(frame)

Seeks to the given frame in a sequence file. If you seek beyond the end of the sequence, the method raises an **EOFError** exception. When a sequence file is opened, the library automatically seeks to frame 0.

Note that in the current version of the library, most sequence formats only allows you to seek to the next frame.

show

im.show()

Displays an image. This method is mainly intended for debugging purposes.

On Unix platforms, this method saves the image to a temporary PPM file, and calls the **xv** utility.

On Windows, it saves the image to a temporary BMP file, and uses the standard BMP display utility to show it (usually **Paint**).

This method returns None.

split

im.split() => *sequence*

Returns a tuple of individual image bands from an image. For example, splitting an "RGB" image creates three new images each containing a copy of one of the original bands (red, green, blue).

tell

im.tell() => *integer*

Returns the current frame number.

thumbnail

im.thumbnail(size)

im.thumbnail(size, filter)

Modifies the image to contain a thumbnail version of itself, no larger than the given size. This method calculates an appropriate thumbnail size to preserve the aspect of the image, calls the **draft** method to configure the file reader (where applicable), and finally resizes the image.

The filter argument can be one of **NEAREST**, **BILINEAR**, **BICUBIC**, or **ANTIALIAS** (best quality). If omitted, it defaults to **NEAREST** (this will be changed to ANTIALIAS in future versions).

Note that the bilinear and bicubic filters in the current version of PIL are not well-suited for thumbnail generation. You should use **ANTIALIAS** unless speed is much more important than quality.

Also note that this function modifies the Image object in place. If you need to use the full resolution image as well, apply this method to a **copy** of the original image. This method returns None.

tobitmap

im.tobitmap() => *string*

Returns the image converted to an X11 bitmap.

tostring

im.tostring() => *string*

Returns a string containing pixel data, using the standard "raw" encoder.

im.tostring(encoder, parameters) => *string*

Returns a string containing pixel data, using the given data encoding.

transform

im.transform(size, method, data) => *image*

im.transform(size, method, data, filter) => *image*

Creates a new image with the given size, and the same mode as the original, and copies data to the new image using the given transform.

In the current version of PIL, the *method* argument can be **EXTENT** (cut out a rectangular subregion), **AFFINE** (affine transform), **QUAD** (map a quadrilateral to a rectangle), or **MESH** (map a number of source quadrilaterals in one operation). The various methods are described below.

The filter argument defines how to filter pixels from the source image. In the current version, it can be **NEAREST** (use nearest neighbour), **BILINEAR** (linear interpolation in a 2x2 environment), or **BICUBIC** (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode "1" or "P", it is set to **NEAREST**.

im.transform(size, EXTENT, data) => *image*

im.transform(size, EXTENT, data, filter) => *image*

Extracts a subregion from the image.

Data is a 4-tuple (*x0, y0, x1, y1*) which specifies two points in the input image's coordinate system. The resulting image will contain data sampled from between these two points, such that (*x0, y0*) in the input image will end up at (0,0) in the output image, and (*x1, y1*) at *size*.

This method can be used to crop, stretch, shrink, or mirror an arbitrary rectangle in the current image. It is slightly slower than **crop**, but about as fast as a corresponding **resize** operation.

im.transform(size, AFFINE, data) => *image*

im.transform(size, AFFINE, data, filter) => *image*

Applies an affine transform to the image, and places the result in a new image with the given size.

Data is a 6-tuple (*a, b, c, d, e, f*) which contain the first two rows from an affine transform matrix. For each pixel (*x, y*) in the output image, the new value is taken from a position (a *x* + b *y* + c, d *x* + e *y* + f) in the input image, rounded to nearest pixel.

This function can be used to scale, translate, rotate, and shear the original image.

im.transform(size, QUAD, data) => *image*

im.transform(size, QUAD, data, filter) => *image*

Maps a quadrilateral (a region defined by four corners) from the image to a rectangle with the given size.

Data is an 8-tuple (*x0, y0, x1, y1, x2, y2, y3, y3*) which contain the upper left, lower left, lower right, and upper right corner of the source quadrilateral.

im.transform(size, MESH, data) image => *image*

im.transform(size, MESH, data, filter) image => *image*

Similar to **QUAD**, but data is a list of target rectangles and corresponding source quadrilaterals.

transpose

im.transpose(method) => *image*

Returns a flipped or rotated copy of an image.

Method can be one of the following: **FLIP_LEFT_RIGHT**, **FLIP_TOP_BOTTOM**, **ROTATE_90**, **ROTATE_180**, or **ROTATE_270**.

verify

im.verify()

Attempts to determine if the file is broken, without actually decoding the image data. If this method finds any problems, it raises suitable exceptions. If you need to load the image after using this method, you must reopen the image file.

Attributes

Instances of the **Image** class have the following attributes:

format

im.format *(string or None)*

The file format of the source file. For images created by the library, this attribute is set to None.

mode

im.mode *(string)*

Image mode. This is a string specifying the pixel format used by the image. Typical values are "1", "L", "RGB", or "CMYK."

size

im.size *((width, height))*

Image size, in pixels. The size is given as a 2-tuple (width, height).

palette

im.palette *(palette or None)*

Colour palette table, if any. If mode is "P", this should be an instance of the **ImagePalette** class. Otherwise, it should be set to None.

info

im.info *(dictionary)*

A dictionary holding data associated with the image.

The ImageChops Module

The **ImageChops** module contains a number of arithmetical image operations, called *channel operations* ("chops"). These can be used for various purposes, including special effects, image compositions, algorithmic painting, and more.

At this time, channel operations are only implemented for 8-bit images (e.g. "L" and "RGB").

Functions

Most channel operations take one or two image arguments and returns a new image. Unless otherwise noted, the result of a channel operation is always clipped to the range 0 to MAX (which is 255 for all modes supported by the operations in this module).

constant

ImageChops.constant(image, value) => *image*

Return a layer with the same size as the given image, but filled with the given pixel value.

duplicate

ImageChops.duplicate(image) => *image*

Return a copy of the given image.

invert

ImageChops.invert(image) => *image*

Inverts an image.

```
out = MAX - image
```

lighter

ImageChops.lighter(image1, image2) => *image*

Compares the two images, pixel by pixel, and returns a new image containing the lighter values.

```
out = max(image1, image2)
```

darker

ImageChops.darker(image1, image2) => *image*

Compares the two images, pixel by pixel, and returns a new image containing the darker values.

```
out = min(image1, image2)
```

difference

ImageChops.difference(image1, image2) => *image*

Returns the absolute value of the difference between the two images.

```
out = abs(image1 - image2)
```

multiply

ImageChops.multiply(image1, image2) => *image*

Superimposes two images on top of each other. If you multiply an image with a solid black image, the result is black. If you multiply with a solid white image, the image is unaffected.

```
out = image1 * image2 / MAX
```

screen

ImageChops.screen(image1, image2) => *image*

Superimposes two inverted images on top of each other.

```
out = MAX - ((MAX - image1) * (MAX - image2) / MAX)
```

add

ImageChops.add(image1, image2, scale, offset) => *image*

Adds two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

```
out = (image1 + image2) / scale + offset
```

subtract

ImageChops.subtract(image1, image2, scale, offset) => *image*

Subtracts two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

```
out = (image1 - image2) / scale + offset
```

blend

ImageChops.blend(image1, image2, alpha) => *image*

Same as the **blend** function in the **Image** module.

composite

ImageChops.composite(image1, image2, mask) => *image*

Same as the **composite** function in the **Image** module.

offset

ImageChops.offset(xoffset, yoffset) => *image*

ImageChops.offset(offset) => *image*

Returns a copy of the image where data has been offset by the given distances. Data wraps around the edges. If yoffset is omitted, it is assumed to be equal to xoffset.

The ImageCrackCode Module (PIL Plus)

The **ImageCrackCode** module allows you to detect and measure features in an image. This module is only available in the PIL Plus package.

Functions

CrackCode

CrackCode(image, position) => *CrackCode instance*

> Identifies a feature in the given image. If the position is omitted, the constructor searches from the top left corner.

Methods and attributes

area

cc.area

> (attribute). The feature area, in pixels.

bbox

cc.bbox

> (attribute). The bounding box, given as a 4-tuple (left, upper, right, lower).

caliper

cc.caliper

> (attribute). The caliper size, given as a 2-tuple (height, width).

centroid

cc.centroid

> (attribute). The center of gravity.

edge

cc.edge

(attribute). True if the feature touches the edges of the image, zero otherwise.

links

cc.links

(attribute). The number of links in the crack code chain.

offset

cc.offset

(attribute). The offset from the upper left corner of the image, to the feature's bounding box,

start

cc.start

(attribute). The first coordinate in the crack code chain.

top

cc.top

(attribute). The topmost coordinate in the crack code chain.

hit

cc.hit(xy) => *flag*

Check if the given point is inside this feature.

topath

cc.topath(xy) => *path*

Return crack code outline as an ImagePath object.

getmask

cc.getmask() => *image*

Get filled feature mask, as an image object.

getoutline

cc.getoutline() => *image*

Get feature outline, as an image object.

The ImageDraw Module

The **ImageDraw** module provide basic graphics support for **Image** objects. It can for example be used to create new images, annotate or retouch existing images, and to generate graphics on the fly for web use.

Example

Example: Draw a Grey Cross Over an Image

```
import Image, ImageDraw

im = Image.open("lena.pgm")

draw = ImageDraw.Draw(im)
draw.line((0, 0) + im.size, fill=128)
draw.line((0, im.size[1], im.size[0], 0), fill=128)
del draw

# write to stdout
im.save(sys.stdout, "PNG")
```

Functions

Draw

Draw(image) => *Draw instance*

Creates an object that can be used to draw in the given image.

Note that the image will be modified in place.

Methods

arc

draw.arc(xy, start, end, options)

Draws an arc (a portion of a circle outline) between the start and end angles, inside the given bounding box.

The **outline** option gives the colour to use for the arc.

bitmap

draw.bitmap(xy, bitmap, options)

Draws a bitmap at the given position, using the current fill colour.

chord

draw.chord(xy, start, end, options)

Same as **arc**, but connects the end points with a straight line.

The **outline** option gives the colour to use for the chord outline. The **fill** option gives the colour to use for the chord interior.

ellipse

draw.ellipse(xy, options)

Draws an ellipse inside the given bounding box.

The **outline** option gives the colour to use for the ellipse outline. The **fill** option gives the colour to use for the ellipse interior.

line

draw.line(xy, options)

Draws a line between the coordinates in the **xy** list.

The coordinate list can be any sequence object containing either 2-tuples [(**x**, **y**), ...] or numeric values [**x**, **y**, ...]. It should contain at least two coordinates.

The **fill** option gives the colour to use for the line.

pieslice

draw.pieslice(xy, start, end, options)

Same as **arc**, but also draws straight lines between the end points and the center of the bounding box.

The **outline** gives the colour to use for the pieslice outline. The **fill** option gives the colour to use for the pieslice interior.

point

draw.point(xy, options)

Draws points (individual pixels) at the given coordinates.

The coordinate list can be any sequence object containing either 2-tuples [(**x**, **y**), …] or numeric values [**x**, **y**, …].

The **fill** option gives the colour to use for the points.

polygon

draw.polygon(xy, options)

Draws a polygon.

The polygon outline consists of straight lines between the given coordinates, plus a straight line between the last and the first coordinate.

The coordinate list can be any sequence object containing either 2-tuples [(**x**, **y**), …] or numeric values [**x**, **y**, …]. It should contain at least three coordinates.

The **outline** option gives the colour to use for the polygon outline. The **fill** option gives the colour to use for the polygon interior.

rectangle

draw.rectangle(box, options)

Draws a rectangle.

The box can be any sequence object containing either 2-tuples [(**x**, **y**), (**x**, **y**)] or numeric values [**x**, **y**, **x**, **y**]. It should contain two coordinates.

Note that the second coordinate pair defines a point just outside the rectangle, also when the rectangle is not filled.

The **outline** option gives the colour to use for the rectangle outline. The **fill** option gives the colour to use for the rectangle interior.

text

draw.text(position, string, options)

Draws the string at the given position. The position gives the upper right corner of the text.

The **font** option is used to specify which font to use. It should be an instance of the **ImageFont** class, typically loaded from file using the **load** method in the **ImageFont** module.

The **fill** option gives the colour to use for the text.

textsize

draw.textsize(string, options) => *(width, height)*

Return the size of the given string, in pixels.

The **font** option is used to specify which font to use. It should be an instance of the **ImageFont** class, typically loaded from file using the **load** method in the **ImageFont** module.

Options

outline *(integer or tuple)*

fill *(integer or tuple)*

font *(ImageFont instance)*

Compatibility

The **Draw** class contains a constructor and a number of methods which are provided for backwards compatibility only. For this to work properly, you should *either* use options on the drawing primitives, or these methods. Do not mix the old and new calling conventions.

ImageDraw

ImageDraw(image) => *Draw instance*

(Deprecated). Same as **Draw**. Don't use this name in new code.

setink

draw.setink(ink)

(Deprecated). Sets the colour to use for subsequent draw and fill operations.

setfill

draw.setfill(mode)

(Deprecated). Sets the fill mode.

If the mode is 0, subsequently drawn shapes (like polygons and rectangles) are outlined. If the mode is 1, they are filled.

setfont

draw.setfont(font)

(Deprecated). Sets the default font to use for the **text** method.

The **font** argument should be an instance of the **ImageFont** class, typically loaded from file using the **load** method in the **ImageFont** module.

The ImageEnhance Module

The **ImageEnhance** module contains a number of classes that can be used for image enhancement.

Example

Example: Vary the Sharpness of an Image

```
import ImageEnhance

enhancer = ImageEnhance.Sharpness(image)

for i in range(8):
    factor = i / 4.0
    enhancer.enhance(factor).show("Sharpness %f" % factor)
```

See the *enhancer.py* demo program in the *Scripts* directory.

Interface

All enhancement classes implement a common interface, containing a single method:

enhance

enhancer.enhance(factor) => *image*

> Returns an enhanced image. The factor is a floating point value controlling the enhancement. Factor 1.0 always returns a copy of the original image, lower factors mean less colour (brightness, contrast, etc), and higher values more. There are no restrictions on this value.

The Color Class

The colour enhancement class is used to adjust the colour balance of an image, in a manner similar to the controls on a colour TV set. This class implements the enhancement interface as described above.

Color

ImageEnhance.Color(image) => *Color enhancer instance*

> Creates an enhancement object for adjusting colour in an image. A factor of 0.0 gives a black and white image, a factor of 1.0 gives the original image.

The Brightness Class

The brightness enhancement class is used to control the brightness of an image.

Brightness

ImageEnhance.Brightness(image) => *Brightness enhancer instance*

> Creates an enhancement object for adjusting brightness in an image. A factor of 0.0 gives a black image, factor 1.0 gives the original image.

The Contrast Class

The contrast enhancement class is used to control the contrast of an image, similar to the contrast control on a TV set.

Contrast

ImageEnhance.Contrast(image) => *Contrast enhancer instance*

> Creates an enhancement object for adjusting contrast in an image. A factor of 0.0 gives an solid grey image, factor 1.0 gives the original image.

The Sharpness Class

The sharpness enhancement class is used to control the sharpness of an image.

Sharpness

ImageEnhance.Sharpness(image) => *Sharpness enhancer instance*

> Creates an enhancement object for adjusting the sharpness of an image. The factor 0.0 gives a blurred image, 1.0 gives the original image, and a factor of 2.0 gives a sharpened image.

The ImageFile Module

The **ImageFile** module provides support functions for the image open and save functions.

In addition, it provides a **Parser** class which can be used to decode an image piece by piece (e.g. while receiving it over a network connection). This class implements the same consumer interface as the standard **sgmllib** and **xmllib** modules.

Example

Example: Parse An Image

```
import ImageFile

fp = open("lena.pgm", "rb")

p = ImageFile.Parser()

while 1:
    s = fp.read(1024)
    if not s:
        break
    p.feed(s)

im = p.close()

im.save("copy.jpg")
```

Functions

Parser

ImageFile.Parser() => *Parser instance*

Creates a parser object. Parsers cannot be reused.

Methods

feed

parser.feed(data)

Feed a string of data to the parser. This method may raise an IOError exception.

close

parser.close() => *image or None*

Tells the parser to finish decoding. If the parser managed to decode an image, it returns an **Image** object. Otherwise, this method raises an IOError exception.

Note: If the file cannot be identified the parser will raise an IOError exception in the close method. If the file can be identified, but not decoded (for example, if the data is damaged, or if it uses an unsupported compression method), the parser will raise an IOError exception as soon as possible, either in feed or close.

The ImageFileIO Module

The **ImageFileIO** module can be used to read an image from a socket, or any other stream device.

This module is deprecated. New code should use the **Parser** class in the ImageFile module instead.

Functions

ImageFileIO

ImageFileIO.ImageFileIO(stream)

> Adds buffering to a stream file object, in order to provide **seek** and **tell** methods required by the **Image.open** method. The stream object must implement **read** and **close** methods.

The ImageFilter Module

The **ImageFilter** module contains definitions for the pre-defined set of filters, to be used in conjuction with the **filter** method of the **Image** class.

Example

Example: Filter an Image

```
import ImageFilter

imout = im.filter(ImageFilter.BLUR)
```

Filters

The current version of the library provides the following set of predefined image enhancement filters:

BLUR, CONTOUR, DETAIL, EDGE_ENHANCE, EDGE_ENHANCE_MORE, EMBOSS, FIND_EDGES, SMOOTH, SMOOTH_MORE, and SHARPEN.

The ImageFont Module

The **ImageFont** module defines a class with the same name. Instances of this class store bitmap fonts, and are used with the **text** method of the **ImageDraw** class.

PIL uses it's own font file format to store bitmap fonts. You can use the **pilfont** utility to convert BDF and PCF font descriptors (X window font formats) to this format.

TrueType support is available as part of the PIL Plus package.

Functions

load

ImageFont.load(file) => *Font instance*

> Loads a font from the given file, and returns the corresponding font object. If this function fails, it raises an **IOError** exception.

load

ImageFont.load_path(file) => *Font instance*

> Same as **load**, but searches for the file along **sys.path** if it's not found in the current directory.

Methods

The following methods are used by the **ImageDraw** layer.

getsize

font.getsize(text) => *(width, height)*

> Returns the width and height of the given text, as a 2-tuple.

getmask

font.getmask(text) => *Image object*

> Returns a bitmap for the text. The bitmap should be an internal PIL storage memory instance (as defined by the **_imaging** interface module).
>
> If the font uses antialiasing, the bitmap should have mode "L" and use a maximum value of 255. Otherwise, it should have mode "1".

The ImageGrab Module

(New in 1.1.3) The **ImageGrab** module can be used to copy the contents of the screen to a PIL image memory.

The current version works on Windows only.

Functions

grab

ImageGrab.grab() => *image*

ImageGrab.grab(bbox) => *image*

> Take a snapshot of the screen, and return an "RGB" image. The bounding box argument can be used to copy only a part of the screen.

The ImageOps Module

(New in 1.1.3) The **ImageOps** module contains a number of 'ready-made' image processing operations. This module is somewhat experimental, and most operators only work on L and RGB images.

Functions

autocontrast

ImageOps.autocontrast(image, cutoff=0) => *image*

> Maximize (normalize) image contrast. This function calculates a histogram of the input image, removes *cutoff* percent of the lightest and darkest pixels from the histogram, and remaps the image so that the darkest pixel becomes black (0), and the lightest becomes white (255).

colorize

ImageOps.colorize(image, black, white) => *image*

> Colorize grayscale image. The *black* and *white* arguments should be RGB tuples; this function calculates a colour wedge mapping all black pixels in the source image to the first colour, and all white pixels to the second colour.

crop

ImageOps.crop(image, border=0) => *image*

> Remove *border* pixels from all four edges. This function works on all image modes.

deform

ImageOps.deform(image, deformer) => *image*

> Deform the image using the given *deformer* object.

equalize

ImageOps.equalize(image) => *image*

Equalize the image histogram. This function applies a non-linear mapping to the input image, in order to create a uniform distribution of grayscale values in the output image.

expand

ImageOps.expand(image, border=0, fill=0) => *image*

Add *border* pixels of border to the image, at all four edges.

fit

ImageOps.fit(image, size, method, bleed, centering) => *image*

Returns a sized and cropped version of the image, cropped to the requested aspect ratio and size. The *size* argument is the requested output size in pixels, given as a (width, height) tuple.

The *method* argument is what resampling method to use. The default is Image.NEAREST (nearest neighbour).

The *bleed* argument allows you to remove a border around the outside the image (from all four edges). The value is a decimal percentage (use 0.01 for one percent). The default value is 0 (no border).

The *centering* argument is used to control the cropping position. (0.5, 0.5) is center cropping (i.e. if cropping the width, take 50% off of the left side (and therefore 50% off the right side), and same with top/bottom).

(0.0, 0.0) will crop from the top left corner (i.e. if cropping the width, take all of the crop off of the right side, and if cropping the height, take all of it off the bottom).

(1.0, 0.0) will crop from the bottom left corner, etc. (i.e. if cropping the width, take all of the crop off the left side, and if cropping the height take none from the top (and therefore all off the bottom)).

The **fit** function was contributed by Kevin Cazabon.

flip

ImageOps.flip(image) => *image*

Flip the image vertically (top to bottom).

grayscale

ImageOps.grayscale(image) => *image*

Convert the image to grayscale.

invert

ImageOps.invert(image) => *image*

Invert (negate) the image.

mirror

ImageOps.mirror(image) => *image*

Flip image horizontally (left to right).

posterize

ImageOps.posterize(image, bits) => *image*

Reduce the number of bits for each colour channel.

solarize

ImageOps.solarize(image, threshold=128) => *image*

Invert all pixel values above the given threshold.

The ImagePath Module

The **ImagePath** module is used to store and manipulate 2-dimensional vector data. Path objects can be passed to the methods in the **ImageDraw** module.

Functions

Path

ImagePath.Path(coordinates) => *Path instance*

> Creates a path object. The coordinate list can be any sequence object containing either 2-tuples [(**x**, **y**), ...] or numeric values [**x**, **y**, ...].

The ImageSequence Module

The **ImageSequence** module contains a wrapper class that provides iteration over the frames of an image sequence.

Functions

Iterator

ImageSequence.Iterator(image) => *Iterator instance*

> Creates an **Iterator** instance that lets you loop over all frames in a sequence.

Methods

The **Iterator** class implements the following method:

The [] Operator

> You can call this operator with integer values from 0 and upwards. It raises an **IndexError** exception when there are no more frames.

The ImageStat Module

The **ImageStat** module calculates global statistics for an image, or a region of an image.

Functions

Stat

ImageStat.Stat(image) => *Stat instance*

ImageStat.Stat(image, mask) => *Stat instance*

> Calculates statistics for the give image. If a mask is included, only the regions covered by that mask are included in the statistics.

ImageStat.Stat(list) => *Stat instance*

> Same as above, but calculates statistics for a previously calculated histogram.

Attributes

The following attributes contain a sequence with one element for each layer in the image. All attributes are lazily evaluated; if you don't need a value, it won't be calculated.

extrema

stat.extrema

> (Attribute). Get min/max values for each band in the image.

count

stat.count

> (Attribute). Get total number of pixels.

sum

stat.sum

> (Attribute). Get sum of all pixels.

sum2

stat.sum2

(Attribute). Squared sum of all pixels.

mean

stat.mean

(Attribute). Average pixel level.

median

stat.median

(Attribute). Median pixel level.

rms

stat.rms

(Attribute). RMS (root-mean-square).

var

stat.var

(Attribute). Variance.

stddev

stat.stddev

(Attribute). Standard deviation.

The ImageTk Module

The **ImageTk** module contains support to create and modify Tkinter **BitmapImage** and **PhotoImage** objects.

For examples, see the demo programs in the *Scripts* directory.

The BitmapImage Class

BitmapImage

ImageTk.BitmapImage(image, options) => *BitmapImage instance*

> Create a Tkinter-compatible bitmap image, which can be used everywhere Tkinter expects an image object.
>
> The given image must have mode "1". Pixels having value 0 are treated as transparent. Options, if any, are passed to Tkinter. The most commonly used option is **foreground**, which is used to specify the colour for the non-transparent parts. See the Tkinter documentation for information on how to specify colours.

The PhotoImage Class

PhotoImage

ImageTk.PhotoImage(image) => *PhotoImage instance*

> Creates a Tkinter-compatible photo image, which can be used everywhere Tkinter expects an image object. If the image is an RGBA image, pixels having alpha 0 are treated as transparent.

ImageTk.PhotoImage(mode, size) => *PhotoImage instance*

> Same as above, but creates an empty (transparent) photo image object. Use **paste** to copy image data to this object.

paste

photo.paste(image, box)

> Pastes an image into the photo image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate. If the box is omitted, or None, all of the image is assumed. In all cases, the size of the pasted image must match the size of the region. If the image mode does not match the photo image mode, conversions are automatically applied.

The ImageWin Module

The **ImageWin** module contains support to create and display images under Windows 95/98, NT, 2000 and later.

The Dib Class

Dib

ImageWin.Dib(mode, size) => *Dib instance*

> This constructor creates a Windows bitmap with the given mode and size. Mode can be one of "1", "L", or "RGB".
>
> If the display requires a palette, this constructor creates a suitable palette and associates it with the image. For an "L" image, 128 greylevels are allocated. For an "RGB" image, a 6x6x6 colour cube is used, together with 20 greylevels.
>
> To make sure that palettes work properly under Windows, you must call the **palette** method upon certain events from Windows. See the method descriptions below.

Methods

expose

dib.expose(hdc)

> Expose (draw) the image using the given device context handle. The handle is an integer representing a Windows **HDC** handle.
>
> In PythonWin, you can use the **GetHandleAttrib** method of the **CDC** class to get a suitable handle.

palette

dib.palette(hdc)

> Installs the palette associated with the image in the given device context. The handle is an integer representing a Windows **HDC** handle.
>
> This method should be called upon **QUERYNEWPALETTE** and **PALETTECHANGED** events from Windows. If this method returns a non-zero value, one or more display palette entries were changed, and the image should be redrawn.

paste

dib.paste(image, box)

Pastes an image into the bitmap image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate. If None is given instead of a tuple, all of the image is assumed. In all cases, the size of the pasted image must match the size of the region. If the image mode does not match the bitmap mode, conversions are automatically applied.

The PSDraw Module

The **PSDraw** module provides basic print support for Postscript printers. You can print text, graphics and images through this module.

Functions

PSDraw

PSDraw.PSDraw(file) => *PSDraw instance*

> Sets up printing to the given file. If file is omitted, *sys.stdout* is assumed.

PSDraw Methods

begin

ps.begin_document()

> Sets up printing of a document.

end

ps.end_document()

> Ends printing.

line

ps.line(from, to)

> Draws a line between the two points. Coordinates are given in Postscript point coordinates (72 points per inch, (0, 0) is the lower left corner of the page).

rectangle

ps.rectangle(box)

> Draws a rectangle.

text

ps.text(position, text)

ps.text(position, text, alignment)

> Draws text at the given position. You must use setfont before calling this method.

setfont

ps.setfont(font, size)

> Selects which font to use. The font argument is a Postscript font name, the size argument is given in points.

setink

ps.setink(ink)

> Selects the pixel value to use with subsequent operations.

setfill

ps.setfill(onoff)

> Selects if subsequent rectangle operations should draw filled rectangles or just outlines.

The pildriver Utility

The **pildriver** tool gives access to most PIL functions from your operating system's command-line interface.

```
$ pildriver program
```

When called as a script, the command-line arguments are passed to a **PILDriver** instance (see below). If there are no command-line arguments, the module runs an interactive interpreter, each line of which is split into space-separated tokens and passed to the execute method.

The **pildriver** tool was contributed by Eric S. Raymond.

Examples

The following example loads **test.png**, crops out a portion of its upper-left-hand corner and displays the cropped portion:

```
$ pildriver show crop 0 0 200 300 open test.png
```

The following example loads **test.tiff**, rotates it 30 degrees, and saves the result as **rotated.png** (in PNG format):

```
$ pildriver save rotated.png rotate 30 open test.tiff
```

The PILDriver Class

The **pildriver** module provides a single class called **PILDriver**.

An instance of the **PILDriver** class is essentially a software stack machine (Polish-notation interpreter) for sequencing PIL image transformations. The state of the instance is the interpreter stack.

The only method one will normally invoke after initialization is the **execute** method. This takes an argument list of tokens, pushes them onto the instance's stack, and then tries to clear the stack by successive evaluation of PILdriver operators. Any part of the stack not cleaned off persists and is part of the evaluation context for the next call of the execute method.

PILDriver doesn't catch any exceptions, on the theory that these are actually diagnostic information that should be interpreted by the calling code.

Methods

In the method descriptions below, each line lists a command token, followed by <>-enclosed arguments which describe how the method interprets the entries on the stack. Each argument specification begins with a type specification: either **int**, **float**, **string**, or **image**.

All operations consume their arguments off the stack (use **dup** to keep copies around). Use **verbose 1** to see the stack state displayed before each operation.

add <image:pic1> <image:pic2> <int:offset> <float:scale>
Pop the two top images, produce the scaled sum with offset.

blend <image:pic1> <image:pic2> <float:alpha>
Replace two images and an alpha with the blended image.

brightness <image:pic1>
Enhance brightness in the top image.

clear
Clear the stack.

color <image:pic1>
Enhance colour in the top image.

composite <image:pic1> <image:pic2> <image:mask>
Replace two images and a mask with their composite.

contrast <image:pic1>
Enhance contrast in the top image.

convert <string:mode> <image:pic1>
Convert the top image to the given mode.

copy <image:pic1>
Make and push a true copy of the top image.

crop <int:left> <int:upper> <int:right> <int:lower> <image:pic1>
Crop and push a rectangular region from the current image.

darker <image:pic1> <image:pic2>
Pop the two top images, push an image of the darker pixels of both.

difference <image:pic1> <image:pic2>
Pop the two top images, push the difference image

draft <string:mode> <int:xsize> <int:ysize>
Configure the loader for a given mode and size.

dup
Duplicate the top-of-stack item.

filter <string:filtername> <image:pic1>
Process the top image with the given filter.

format <image:pic1>
Push the format of the top image onto the stack.

getbbox
Push left, upper, right, and lower pixel coordinates of the top image.

extrema
Push minimum and maximum pixel values of the top image.

invert <image:pic1>
Invert the top image.

lighter <image:pic1> <image:pic2>
Pop the two top images, push an image of the lighter pixels of both.

merge <string:mode> <image:pic1> [<image:pic2> [<image:pic3> [<image:pic4>]]]
Merge top-of stack images in a way described by the mode.

mode <image:pic1>
Push the mode of the top image onto the stack.

multiply <image:pic1> <image:pic2>
Pop the two top images, push the multiplication image.

new <int:xsize> <int:ysize> <int:color>:
Create and push a greyscale image of given size and colour.

offset <int:xoffset> <int:yoffset> <image:pic1>

Offset the pixels in the top image.

open <string:filename>

Open the indicated image, read it, push the image on the stack.

paste <image:figure> <int:xoffset> <int:yoffset> <image:ground>

Paste figure image into ground with upper left at given offsets.

pop

Discard the top element on the stack.

resize <int:xsize> <int:ysize> <image:pic1>

Resize the top image.

rotate <int:angle> <image:pic1>

Rotate image through a given angle

save <string:filename> <image:pic1>

Save image with default options.

save2 <string:filename> <string:options> <image:pic1>

Save image with specified options.

screen <image:pic1> <image:pic2>

Pop the two top images, superimpose their inverted versions.

sharpness <image:pic1>

Enhance sharpness in the top image.

show <image:pic1>

Display and pop the top image.

size <image:pic1>

Push the image size on the stack as (y, x).

subtract <image:pic1> <image:pic2> <int:offset> <float:scale>

Pop the two top images, produce the scaled difference with offset.

swap

Swap the top-of-stack item with the next one down.

thumbnail <int:xsize> <int:ysize> <image:pic1>

Modify the top image in the stack to contain a thumbnail of itself.

transpose <string:operator> <image:pic1>

Transpose the top image.

verbose <int:num>

Set verbosity flag from top of stack.

The pilconvert Utility

The **pilconvert** tool converts an image from one format to another. The output format is determined by the target extension, unless explicitly specified with the **-c** option.

```
$ pilconvert lena.tif lena.png
$ pilconvert -c JPEG lena.tif lena.tmp
```

The pilfile Utility

The **pilfile** tool identifies image files, showing the file format, size, and mode for every image it can identify.

```
$ pilfile *.tif
lena.tif: TIFF 128x128 RGB
```

Use the **-i** option to display the info attribute. Use the **-t** option to display the tile descriptor (which contains information used to load the image).

The pilfont Utility

The **pilfont** tool converts BDF or PCF font files to a format that can be used with PIL's **ImageFont** module.

```
$ pilfont *.pdf
```

The pilprint Utility

The **pilprint** tool prints an image to any PostScript level 1 printer. The image is centred on the page, with the filename (minus path and extension) written above it. Output is written to standard output.

```
  $ pilprint lena.tif | lpr -h
```

You can use the **-p** option to print directly via **lpr** and **-c** to print to a colour printer (otherwise, a colour image is translated to greyscale before being sent to the printer).

A. Software License

The Python Imaging Library is:

> Copyright © 1997-2002 by Secret Labs AB
> Copyright © 1995-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B. Getting Support

Patches, fixes, updates, and new utilities are welcome. If you stumble upon files that the library does not handle as expected, post a note to the Image SIG mailing list (see below). If you fix such a problem and supply a patch, you may send me the image file anyway so I don't mess things up again in later revisions.

Ideas on formats and features that should be added, sample files, and other contributions are also welcome.

For all sorts of updates, including information on commercial support and extensions to PIL, check the PIL product page, at:

> http://www.pythonware.com/products/pil

You may also find information related to PIL at http://www.pythonware.com or via the Python home page http://www.python.org

For support and general questions, send e-mail to the Python Image SIG mailing list:

> image-sig@python.org

You can join the Image SIG by sending a mail to *image-sig-request@python.org*. Put *subscribe* in the message body to automatically subscribe to the list, or *help* to get additional information.

Alternatively, you can use the Python mailing list, *python-list@python.org*, or the newsgroup *comp.lang.python*.

C. Image File Formats

The Python Imaging Library supports a wide variety of raster file formats. Nearly 30 different file formats can be identified and read by the library. Write support is less extensive, but most common interchange and presentation formats are supported.

The **open** function identifies files from their contents, not their names, but the **save** method looks at the name to determine which format to use, unless the format is given explicitly.

Format Descriptions

BMP

PIL reads and writes Windows and OS/2 BMP files containing "1", "L", "P", or "RGB" data. 16-colour images are read as "P" images. Run-length encoding is not supported.

The **open** method sets the following **info** properties:

compression

Set to "bmp_rle" if the file is run-length encoded.

CUR (read only)

CUR is used to store cursors on Windows. The CUR decoder reads the largest available cursor. Animated cursors are not supported.

DCX (read only)

DCX is a container file format for PCX files, defined by Intel. The DCX format is commonly used in fax applications. The DCX decoder files containing "1", "L", "P", or "RGB" data. Only the first image is read.

EPS (write-only)

The library identifies EPS files containing image data. It can also write EPS images.

FLI, FLC (read only)

The library reads Autodesk FLI and FLC animations.

The **open** method sets the following **info** properties:

duration

The delay (in milliseconds) between each frame.

FPX (read only)

The library reads Kodak *FlashPix* files. In the current version, only the highest resolution image is read from the file, and the viewing transform is not taken into account.

Note: To enable full FlashPix support, you need to build and install the IJG JPEG library before building the Python Imaging Library. See the distribution README for details.

GBR (read only)

The GBR decoder reads GIMP brush files.

The **open** method sets the following **info** properties:

description

The brush name.

GD (read only)

The library reads GD uncompressed files. Note that this file format cannot be automatically identified, so you must use the **open** function in the **GdImageFile** module to read such a file.

The **open** method sets the following **info** properties:

transparency

Transparency colour index. This key is omitted if the image is not transparent.

GIF

The library reads GIF87a and GIF89a versions of the GIF file format. The library writes run-length encoded GIF87a files. Note that GIF files are always read as palette mode ("P") images.

The **open** method sets the following **info** properties:

version

Version (either "GIF87a" or "GIF89a").

transparency

Transparency colour index. This key is omitted if the image is not transparent.

ICO (read only)

ICO is used to store icons on Windows. The largest available icon is read.

IM

IM is a format used by LabEye and other applications based on the IFUNC image processing library. The library reads and writes most uncompressed interchange versions of this format.

IM is the only format that can store *all* internal PIL formats.

IMT (read only)

The library reads Image Tools images containing "L" data.

JPEG

The library reads JPEG, JFIF, and Adobe JPEG files containing "L", "RGB", or "CMYK" data. It writes standard and progressive JFIF files.

Using the **draft** method, you can speed things up by converting "RGB" images to "L", and resize images to 1/2, 1/4 or 1/8 of their original size while loading them. The draft method also configures the JPEG decoder to trade some quality for speed.

The **open** method sets the following **info** properties:

jfif

JFIF application marker found. If the file is not a JFIF file, this key is not present.

adobe

Adobe application marker found. If the file is not an Adobe JPEG file, this key is not present.

progression

Indicates that this is a progressive JPEG file.

The **save** method supports the following options:

quality

The image quality, on a scale from 1 (worst) to 100 (best). The default is 75.

optimize

If present, indicates that the encoder should make an extra pass over the image in order to select optimal encoder settings.

progression

If present, indicates that this image should be stored as a progressive JPEG file.

Note: To enable JPEG support, you need to build and install the IJG JPEG library before building the Python Imaging Library. See the distribution README for details.

MIC (read only)

The library identifies and reads Microsoft Image Composer (MIC) files. When opened, the first sprite in the file is loaded. You can use **seek** and **tell** to read other sprites from the file.

MCIDAS (read only)

The library identifies and reads 8-bit McIdas area files.

MPEG (identify only)

The library identifies MPEG files.

MSP

The library identifies and reads MSP files from Windows 1 and 2. The library writes uncompressed (Windows 1) versions of this format.

PCD (read only)

The library reads PhotoCD files containing "RGB" data. By default, the 768x512 resolution is read. You can use the **draft** method to read the lower resolution versions instead, thus effectively resizing the image to 384x256 or 192x128. Higher resolutions cannot be read by the Python Imaging Library.

PCX

The library reads and writes PCX files containing "1", "L", "P", or "RGB" data.

PDF (write only)

The library can write PDF (Acrobat) images. Such images are written as binary PDF 1.1 files, using either JPEG or HEX encoding depending on the image mode (and whether JPEG support is available or not).

PNG

The library identifies, reads, and writes PNG files containing "1", "L", "P", "RGB", or "RGBA" data. Interlaced files are currently not supported.

The **open** method sets the following **info** properties:

gamma

> Gamma, given as a floating point number.

transparency

> Transparency colour index. This key is omitted if the image is not a transparent palette image.

The **save** method supports the following options:

optimize

> If present, instructs the PNG writer to make the output file as small as possible. This includes extra processing in order to find optimal encoder settings.

Note: To enable PNG support, you need to build and install the ZLIB compression library before building the Python Imaging Library. See the distribution README for details.

PPM

The library reads and writes PBM, PGM and PPM files containing "1", "L" or "RGB" data.

PSD (read only)

The library identifies and reads PSD files written by Adobe Photoshop 2.5 and 3.0.

SGI (read only)

The library reads uncompressed "L" and "RGB" files. This driver is highly experimental.

SUN (read only)

The library reads uncompressed "1", "P", "L" and "RGB" files.

TGA (read only)

The library reads 24- and 32-bit uncompressed and run-length encoded TGA files.

TIFF

The library reads and writes TIFF files containing "1", "L", "RGB", or "CMYK" data. It reads both striped and tiled images, pixel and plane interleaved multi-band images, and either uncompressed, or Packbits, LZW, or JPEG compressed images. In the current version, PIL always writes uncompressed TIFF files.

The **open** method sets the following **info** properties:

compression

Compression mode.

In addition, the **tag** attribute contains a dictionary of decoded TIFF fields. Values are stored as either strings or tuples. Note that only short, long and ASCII tags are correctly unpacked by this release.

XBM

The library reads and writes X bitmap files (mode "1").

XPM (read only)

The library reads X pixmap files (mode "P") with 256 colours or less.

The **open** method sets the following **info** properties:

transparency

Transparency colour index. This key is omitted if the image is not transparent.

File Extensions

The Python Imaging Library associates file name extensions to each file format. The **open** function identifies files from their contents, not their names, but the **save** method looks at the name to determine which format to use, unless the format is given explicitly.

BMP: ".bmp", ".dib"

CUR: ".cur"

DCX: ".dcx"

EPS: ".eps", ".ps"

FLI: ".fli", ".flc"

FPX: ".fpx"

GBR: ".gbr"

GD: ".gd"

GIF: ".gif"

ICO: ".ico"

IM: ".im"

JPEG: ".jpg", ".jpe", ".jpeg"

MIC: ".mic"

MSP: ".msp"

PCD: ".pcd"

PCX: ".pcx"

PDF: ".pdf"

PNG: ".png"

PPM: ".pbm", ".pgm", ".ppm"

PSD: ".psd"

SGI: ".bw", ".rgb", ".cmyk"

SUN: ".ras"

TGA: ".tga"

TIFF: ".tif", ".tiff"

XBM: ".xbm"

XPM: ".xpm"

Keep in mind that not all of these formats can actually be saved by the library.

D. Writing Your Own File Decoder

The Python Imaging Library uses a *plug-in* model which allows you to add your own decoders to the library, without any changes to the library itself. Such plug-ins have names like *XxxImagePlugin.py*, where *Xxx* is a unique format name (usually an abbreviation).

A decoder plug-in should contain a decoder class, based on the *ImageFile* base class defined in the module with the same name. This class should provide an *_open* method, which reads the file header and sets up at least the *mode* and *size* attributes. To be able to load the file, the method must also create a list of *tile* descriptors. The class must be explicitly registered, via a call to the *Image* module.

For performance reasons, it is important that the *_open* method quickly rejects files that do not have the appropriate contents.

Example

The following plug-in supports a simple format, which has a 128-byte header consisting of the words "SPAM" followed by the width, height, and pixel size in bits. The header fields are separated by spaces. The image data follows directly after the header, and can be either bi-level, greyscale, or 24-bit true colour.

Example: File: SpamImagePlugin.py

```
import Image, ImageFile
import string

class SpamImageFile(ImageFile.ImageFile):

    format = "SPAM"
    format_description = "Spam raster image"

    def _open(self):

        # check header
        header = self.fp.read(128)
        if header[:4] != "SPAM":
            raise SyntaxError, "not a SPAM file"

        header = string.split(header)

        # size in pixels (width, height)
        self.size = int(header[1]), int(header[2])

        # mode setting
        bits = int(header[3])
        if bits == 1:
            self.mode = "1"
        elif bits == 8:
            self.mode = "L"
        elif bits == 24:
            self.mode = "RGB"
        else:
            raise SyntaxError, "unknown number of bits"

        # data descriptor
        self.tile = [
            ("raw", (0, 0) + self.size, 128, (self.mode, 0, 1))
        ]

Image.register_open("SPAM", SpamImageFile)

Image.register_extension("SPAM", ".spam")
Image.register_extension("SPAM", ".spa") # dos version
```

The format handler must always set the *size* and *mode* attributes. If these are not set, the file cannot be opened. To simplify the decoder, the calling code considers exceptions like *SyntaxError*, *KeyError*, and *IndexError*, as a failure to identify the file.

Note that the decoder must be explicitly registered using the *register_open* function in the *Image* module. Although not required, it is also a good idea to register any extensions used by this format.

The Tile Attribute

To be able to read the file as well as just identifying it, the *tile* attribute must also be set. This attribute consists of a list of tile descriptors, where each descriptor specifies how data should be loaded to a given region in the image. In most cases, only a single descriptor is used, covering the full image.

The tile descriptor is a 4-tuple with the following contents:

```
(decoder, region, offset, parameters)
```

The fields are used as follows:

decoder. Specifies which decoder to use. The "raw" decoder used here supports uncompressed data, in a variety of pixel formats. For more information on this decoder, see the description below.

region. A 4-tuple specifying where to store data in the image.

offset. Byte offset from the beginning of the file to image data.

parameters. Parameters to the decoder. The contents of this field depends on the decoder specified by the first field in the tile descriptor tuple. If the decoder doesn't need any parameters, use None for this field.

Note that the *tile* attribute contains a *list* of tile descriptors, not just a single descriptor.

The Raw Decoder

The raw decoder is used to read uncompressed data from an image file. It can be used with most uncompressed file formats, such as PPM, BMP, uncompressed TIFF, and many others. To use the raw decoder with the *fromstring* function, use the following syntax:

```
image = fromstring(
    mode, size, data, "raw",
    raw mode, stride, orientation
    )
```

When used in a tile descriptor, the parameter field should look like:

```
(raw mode, stride, orientation)
```

The fields are used as follows:

raw mode. The pixel layout used in the file, and is used to properly convert data to PIL's internal layout. For a summary of the available formats, see the table below.

stride. The distance in bytes between two consecutive lines in the image. If 0, the image is assumed to be packed (no padding between lines). If omitted, the stride defaults to 0.

orientation. Whether the first line in the image is the top line on the screen (1), or the bottom line (-1). If omitted, the orientation defaults to 1.

The *raw mode* field is used to determine how the data should be unpacked to match PIL's internal pixel layout. PIL supports a large set of raw modes; for a complete list, see the table in the *Unpack.c* module. The following table describes some commonly used *raw modes*:

"1". 1-bit bilevel, stored with the leftmost pixel in the most significant bit. 0 means black, 1 means white.

"1;I". 1-bit inverted bilevel, stored with the leftmost pixel in the most significant bit. 0 means white, 1 means black.

"1;R". 1-bit reversed bilevel, stored with the leftmost pixel in the least significant bit. 0 means black, 1 means white.

"L". 8-bit greyscale. 0 means black, 255 means white.

"L;I". 8-bit inverted greyscale. 0 means white, 255 means black.

"P". 8-bit palette-mapped image.

"RGB". 24-bit true colour, stored as (red, green, blue).

"BGR". 24-bit true colour, stored as (blue, green, red).

"RGBX". 24-bit true colour, stored as (blue, green, red, pad).

"RGB;L". 24-bit true colour, line interleaved (first all red pixels, the all green pixels, finally all blue pixels).

Note that for the most common cases, the raw mode is simply the same as the mode.

The Python Imaging Library supports many other decoders, including JPEG, PNG, and PackBits. For details, see the *decode.c* source file, and the standard plug-in implementations provided with the library.

Decoding Floating Point Data

PIL provides some special mechanisms to allow you to load a wide variety of formats into a mode "F" (floating point) image memory.

You can use the "raw" decoder to read images where data is packed in any standard machine data type, using one of the following raw modes:

"F". 32-bit native floating point.

"F;8". 8-bit unsigned integer.

"F;8S". 8-bit signed integer.

"F;16". 16-bit little endian unsigned integer.

"F;16S". 16-bit little endian signed integer.

"F;16B". 16-bit big endian unsigned integer.

"F;16BS". 16-bit big endian signed integer.

"F;16N". 16-bit native unsigned integer.

"F;16NS". 16-bit native signed integer.

"F;32". 32-bit little endian unsigned integer.

"F;32S". 32-bit little endian signed integer.

"F;32B". 32-bit big endian unsigned integer.

"F;32BS". 32-bit big endian signed integer.

"F;32N". 32-bit native unsigned integer.

"F;32NS". 32-bit native signed integer.

"F;32F". 32-bit little endian floating point.

"F;32BF". 32-bit big endian floating point.

"F;32NF". 32-bit native floating point.

"F;64F". 64-bit little endian floating point.

"F;64BF". 64-bit big endian floating point.

"F;64NF". 64-bit native floating point.

The Bit Decoder

If the raw decoder cannot handle your format, PIL also provides a special "bit" decoder which can be used to read various packed formats into a floating point image memory.

To use the bit decoder with the *fromstring* function, use the following syntax:

```
image = fromstring(
    mode, size, data, "bit",
    bits, pad, fill, sign, orientation
    )
```

When used in a tile descriptor, the parameter field should look like:

```
(bits, pad, fill, sign, orientation)
```

The fields are used as follows:

bits. Number of bits per pixel (2-32). No default.

pad. Padding between lines, in bits. This is either 0 if there is no padding, or 8 if lines are padded to full bytes. If omitted, the pad value defaults to 8.

fill. Controls how data are added to, and stored from, the decoder bit buffer.

fill=0. Add bytes to the msb end of the decoder buffer; store pixels from the msb end.

fill=1. Add bytes to the lsb end of the decoder buffer; store pixels from the msb end.

fill=2. Add bytes to the msb end of the decoder buffer; store pixels from the lsb end.

fill=3. Add bytes to the lsb end of the decoder buffer; store pixels from the lsb end.

If omitted, the fill order defaults to 0.

sign. If non-zero, bit fields are sign extended. If zero or omitted, bit fields are unsigned.

orientation. Whether the first line in the image is the top line on the screen (1), or the bottom line (-1). If omitted, the orientation defaults to 1.