

# The Python Imaging Library

The Python Imaging Library (PIL) provides functions for reading, writing and manipulating images in a variety of formats. The use for this library includes image display, format conversions, and a variety of image manipulating operations. Some support for image processing in the scientific sense is also included, and more will be added in future versions.

This document describes release 0.2b3 of this library.

# Contents

The Python Imaging Library .....	1
Contents .....	2
What's here? .....	4
Tutorial .....	5
The Image Class.....	5
Reading and Writing Images .....	6
Cutting, Pasting and Merging Images.....	6
Geometrical Transforms .....	8
Colour Transforms .....	8
Image Enhancement .....	8
Postscript Printing .....	10
More on Reading Images.....	11
Library Reference.....	13
Concepts.....	13
The Image Class .....	15
Example .....	15
Functions.....	15
Methods.....	16
Attributes.....	21
Constants.....	21
The ImageChops Module .....	22
Functions.....	22
The ImageDraw Class .....	24
Example .....	24
Functions.....	24
Methods.....	24
The ImageEnhance Module .....	26
Example .....	26
Interface .....	26
The Color Class.....	26

The Brightness Class.....	26
The Contrast Class .....	26
The Sharpness Class.....	27
The ImageFileIO class.....	28
Functions.....	28
The ImageFilter module .....	29
Example .....	29
Filters .....	29
The ImagePalette Class .....	30
Example .....	30
Functions.....	30
The ImageTk Module.....	31
The BitmapImage Class.....	31
The PhotoImage Class .....	31
The ImageWin Module .....	32
The Dib Class.....	32
Methods.....	32
The PSDraw Class.....	33
Functions.....	33
ImageDraw Methods.....	33
Image File Formats.....	34
Summary.....	34
Format Descriptions.....	35
File Extensions.....	38
Utilities .....	39
Contributions .....	40
Software License .....	41

## What's here?

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

This library intends to provide extensive file format support, an efficient internal representation, and basic image processing capabilities.

The core image library is designed for fast access to data stored in a few, basic pixel formats. It should be well suited as a base for a general image processing tool.

Let's look at a few possible uses for this library.

### Image Archives

The Python Imaging Library is well suited 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, but write support is currently somewhat restricted. This will be addressed in future versions.

### 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. However, only nearest neighbour interpolation is available.

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

## 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 see what the file really contained.

```
>>> 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" (for 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. It is very handy for debugging and tests, though.)

On the following pages, you'll find 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, you 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 in Image class. When saving files, the name becomes important. Unless you specify the format, the library use the filename extension to figure out which file format to use when storing the file.

```
# 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:
            print "cannot convert", infile
```

You can use a second argument to the save method in order to explicitly specify a file format. If you use a non-standard extension, you must always specify the format this way:

```
# 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:
            print "cannot create thumbnail for", infile
```

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

```
# 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:
        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.

```
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 300x300 pixels and nothing else.

You can now process the region in some fashion, and possibly paste 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).

Note that the paste method is the only way to modify an image in place.

Here's an additional example:

```
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 a 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:

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

## Geometrical Transforms

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

```
out = im.resize((128, 128))

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

To rotate the image in full 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.

```
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.

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

The library includes transforms 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 for image enhancement.

### Filters

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

```
import ImageFilter

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



## Point Operations

The point method can be used to translate the pixel values of an image. This can for example be used to manipulate the image contrast. In most cases, you can use pass this function a function object expecting one argument. Each pixel is processed according to that function:

```
# 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:

```
# 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 as much 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.

```
import ImageEnhance

enh = ImageEnhance.Contrast(im)

enh.enhance(1.3).show("30% more contrast")
```

## Postscript Printing

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

```
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, you use the `open` function in the `Image` module to open an image file. In most cases, you simply pass it the filename as argument:

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

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

You can also use a file object instead. The file object must implement the `read`, `seek` and `tell` methods, and be opened in binary mode.

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

To read an image from data that you have in a string, use the `StringIO` class:

```
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 provided with the Python Imaging Library distribution to access it.

```
import TarIO

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

See comments in these modules for details.

## Reading Images from Streams

To be able to use a non-seekable device (such as a socket), the library includes a special file class, `ImageFileIO`, that implements a buffering scheme supporting `seek` and `tell` operations.

```
import urllib
import ImageFileIO

fp = urllib.urlopen("http://www.python.org/logo.gif")

fp = ImageFileIO.ImageFileIO(fp)
im = Image.open(fp)
```

### Controlling the Decoder

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

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

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

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

Note that the resulting image may not exactly match the requested mode and size.

# Library Reference

## Concepts

The Python Imaging Library handles raster image data, 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 of bands in an image, check the length of the 'mode' attribute.

### Mode

The mode of an image defines the type and depth of a pixel in the image, and the number of bands. The current release supports the following 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)
- RGB (3x8-bit pixels, true colour)
- RGBA (4x8-bit pixels, true colour with transparency mask)
- CMYK (4x8-bit pixels, colour separation)

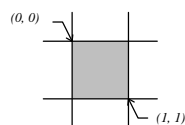
You can read the mode of an image through the 'mode' attribute. This is a string containing one of the above values. Note that each character in this string corresponds to a band in the image.

### 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 pixel corners:



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. The palette is represented by an ImagePalette, which is used to map to any of the other modes.

**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.

# The Image Class

To use the Image class and related functions, import the Image module.

## Example

```
import Image

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

import glob
for infile in glob.glob("*.jpg"):
    try:
        outfile = os.splitext(file)[0] + ".thumbnail"
        Image.open(infile).resize(128, 128).save(outfile, "JPEG")
    except:
        print "Cannot create thumbnail for %s" % infile
```

## Functions

### new

**new( mode, size )** or **new( mode, size, colour )** creates a new image with the given mode and size. Size is given as a 2-tuple. If colour is not given, the image is initialised to solid black.

### open

**open( infile )** or **open(infile, "r" )** opens and identifies the given image file. The actual image data is not read from the file until you try to process the data (or call the load method).

You can use either a string (giving 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.

To read images from streams, use the **ImageFileIO** classes.

### blend

**blend( image 1, image 2, alpha )** 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

**composite( image1, image2, mask )** creates a new image by interpolating between the given images, using the mask as alpha. The mask can be either "1", "L", or "RGBA." All images must have the same size.

**eval**

**eval**(*function*, *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.

**merge**

**merge**(*mode*, *bands*) 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 have the following methods. Unless otherwise stated, all methods return a new instance of the Image class, holding the resulting image.

**convert**

**convert**(*mode*) 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 release 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 an greyscale image into a bilevel image (mode “1”), values below 128 are set to 0 and values above are set to 255. To use other thresholds, use the point method.

**copy**

**copy**() copies the image. Use this function if you wish to paste things into an image, but still retain the original.

**crop**

**crop**(*box*) returns a rectangular region from the current image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate.

**draft**

**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**

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



**getbbox**

**getbbox()** 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.

**getdata**

**getdata()** returns the contents of a the image as a sequence object containing pixel values.

**getpixel**

**getdata( *x*, *y* )** returns the pixel at the given position. If the image is a multi-layer image, this method returns a tuple.

**histogram**

**histogram()** 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 an greyscale (“L”) image by this method.

**load**

**load()** allocates storage for the image and loads it from the file. In normal cases, you don’t need to call this method, since the Image class automatically loads an opened image when it is accessed the first time.

**offset**

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

**paste**

**paste( *image*, *box* )** or **paste( *image*, *box*, *mask* )** pastes an image into self. 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 self is assumed. In any case, the size of the pasted image must match the size of the region. If the mode does not match the mode of self, conversions are automatically applied (see the convert method for details).

If a mask is given, it is interpreted as transparency mask for the image being pasted. 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. Values in-between can be used for transparency effects.

Note that if you paste an “RGBA” image, the alpha band is ignored unless you use the same image as mask as well.

**point**

**point( *table* )** or **point( *function* )** maps each pixel in the image through the given table. The table should contains 256 values per band in the image. If a function is used instead, it is evaluated once for each possible pixel value, and the resulting table is applied to all bands of the image.

**putdata****resize**

**resize**( *size* ) returns a resized copy of an image. The size argument gives the requested size in pixels.

**rotate**

**rotate**( *angle* ) returns a copy of an image rotated the given number of degrees counter clockwise around its center.

**save**

**save**( *outfile* ) or **save**( *outfile, format* ) 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.

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.

**show**

**show**() displays an image. On Unix platforms, this method saves the image to disk as a PPM file, and calls the **xv** utility. This method returns None.

**split**

**split**() returns a tuple of individual image bands from an image. For example, if you split an “RGB” image, you get three new images, containing copies of the red, green, and blue bands from the original image.

**thumbnail**

**thumbnail**( *size* ) 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 calls **resize**.

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.

**transform**






**transform**( *xsize, ysize, EXTENT, ( x0, y0, x1, y1 )* ) returns a transformed copy of an image. The four values gives two points in the input image’s coordinate system. The resulting image will contain data sampled from between these two points, so that (x0, y0) in the input image will end up at (0,0) in the output image, and (x1, y1) at (xsize, ysize). 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.

**transform**( *xsize, ysize, AFFINE, ( a, b, c, d, e, f )* ) returns a transformed copy of an image. The six values are the first two rows from an affine transform matrix, which can be used to control scaling, translation, rotation, and shear.

The affine transform matrix defines a coordinate translation from coordinates in the output image to coordinates in the original image. For each pixel  $(x, y)$  in the output image, the new value is taken from a position  $(ax + by + c, dx + ey + f)$  in the input image, rounded to nearest pixel.

**transpose**

**transpose**( *method* ) returns a flipped or rotated copy of an image.

<i>Method</i>	<i>Result</i>
FLIP_RIGHT_LEFT	
FLIP_TOP_BOTTOM	
ROTATE_90	
ROTATE_180	
ROTATE_270	

**verify**

**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

### **format**

The file format that this image was read from. For images created by the library, this attribute is set to None.

### **mode**

Image mode. This is a string specifying the pixel format used by the image, with typical values like “1”, “L”, “RGB”, or “CMYK”.

### **size**

Image size, in pixels. The size is given as a 2-tuple, with the horizontal size given first.

### **palette**

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**

A dictionary holding data associated with the image.

## Constants

The following constants are defined in the Image module.

### **Geometric Transform Models**

```
AFFINE  
EXTENT
```

### **Transpose Methods**

```
FLIP_LEFT_RIGHT  
FLIP_TOP_BOTTOM  
ROTATE_90  
ROTATE_180  
ROTATE_270
```

## The ImageChops Module

This 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.

### Functions

All functions take one or two image arguments and returns a new image. The result of a channel operation is always clipped to the range 0 to MAX (which is 255 for all modes supported by the current version of the Python Imaging Library).

#### **invert**

**invert**( *image* ) inverts an image.

$$out = MAX - image$$

#### **lighter**

**lighter**( *image1*, *image2* ) compares the two images, pixels by pixel, and returns a new image containing the lighter value for each pixel.

$$out = \max( image1, image2 )$$

#### **darker**

**darker**( *image1*, *image2* ) compares the two images, pixels by pixel, and returns a new image containing the darker value for each pixel.

$$out = \min( image1, image2 )$$

#### **difference**

**difference**( *image1*, *image2* ) returns the absolute value of the difference between the two images.

$$out = \text{abs}( image1 - image2 )$$

**multiply**

**multiply**( *image1*, *image2* ) 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**

**screen**( *image1*, *image2* ) superimposes two inverted images on top of each other.

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

**add**

**add**( *image1*, *image2*, *scale*, *offset* ) adds two images, dividing the result by scale and adding the offset.

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

**subtract**

**subtract**( *image1*, *image2*, *scale*, *offset* ) subtracts two images, dividing the result by scale and adding the offset.

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

## The ImageDraw Class

This class provide some basic graphics support for “1”, “L” and “P” images.

To use the ImageDraw class, import the ImageDraw module.

### Example

```
import Image, ImageDraw

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

# draw a grey cross over the image
draw = ImageDraw.ImageDraw(im)
draw.setink(128)
draw.line((0, 0), im.size)
draw.line((0, im.size[1], (im.size[0], 0))
del draw

im.show()
```

### Functions

#### ImageDraw (constructor)

**ImageDraw**( *image* ) creates an object that can be used to draw in the given image. The image must be a “1”, “L” or “P” image. Ink is set to 255, and fill is off.



## Methods

### **line**

**line**(*from*, *to* ) draws a line between the two points.

**line**( *list* ) connects the points in the list. The list can be any sequence object containing either 2-tuples [(*x*, *y*), ...] or numeric values [*x*, *y*, ...].

### **point**

**point**( *position* ) draws a point at the given position.

### **polygon**

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

### **rectangle**

**rectangle**( *box* ) draws a rectangle. Note that the second coordinate pair defines a point just outside the rectangle, also when the rectangle is not filled.

### **setink**

**setink**( *ink* ) selects the pixel value to use with subsequent operations.

### **setfill**

**setfill**( *onoff* ) selects if subsequently polygons and rectangles should be filled objects or just outlined.

## The ImageEnhance Module

This module contains a number of classes that can be used for image enhancement. The constructors all take the current image as argument, and returns a new object that can be used to quickly adjust the enhancement factor.

### 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)
```

Also see the **enhancer.py** demo program in the **Scripts** directory.

### Interface

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

#### enhance

**enhance**(*factor*) 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 means less colour (brightness, contrast, etc), and higher values more. There is no restrictions on this value.

### The Color Class

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

#### Color (constructor)

**Color**(*image*) 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 (constructor)

**Brightness**(*image*) 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 control on a TV set.

### Contrast (constructor)

**Contrast**( *image* ) 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 (constructor)

**Sharpness**( *image* ) creates an enhancement object for adjusting sharpness in 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 ImageFileIO class

This class is used to provide correct file class semantics for data read from a socket, or other stream device. To use this class, import the ImageFileIO module.

This class should only be used with the Python Imaging Library.

### Functions

#### **ImageFileIO (constructor)**

**ImageFileIO**(*file* ) adds buffering to a stream file object, in order to provide seek and tell methods for the Python Imaging Library. The stream object must implement read and close operations.

## The ImageFilter module

This module contains definitions for the pre-defined set of filters, for use with the filter method in the Image class.

### Example

```
import ImageFilter

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

### Filters

The following set of image enhancement filters are provided with release 0.1.

<i>Filter</i>	<i>Description</i>
<b>BLUR</b>	
<b>CONTOUR</b>	
<b>DETAIL</b>	
<b>EDGE_ENHANCE</b>	
<b>EDGE_ENHANCE_MORE</b>	
<b>EMBOSS</b>	
<b>FIND_EDGES</b>	
<b>SMOOTH</b>	
<b>SMOOTH_MORE</b>	
<b>SHARPEN</b>	

## The ImagePalette Class

To use the ImagePalette class and related functions, import the ImagePalette module.

### Example

```
import ImagePalette

im.palette = ImagePalette.ImagePalette("RGB")
```

### Functions

#### ImagePalette (constructor)

**ImagePalette**(*mode* = "RGB" ). This constructor create a new palette, mapping from "P" to the given mode. The palette is initialised to a linear ramp.

# The ImageTk Module

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

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

## The BitmapImage Class

### BitmapImage (constructor)

**BitmapImage**( *image*, *options* ). Create a bitmap image object, which can be used everywhere Tkinter expects an image object.

The given image must have mode “1”. Pixels having value 0 is 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 (constructor)

**PhotoImage**( *image* ) creates a Tkinter-compatible photo image, which can be used everywhere Tkinter expects an image object.

**PhotoImage**( *mode*, *size* ) creates an empty (transparent) photo image object. Use **paste** to copy image data to this object.

### paste

**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 None is given instead of a tuple, all of the image is assumed. In any case, 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

This module contains support to create and display images under Windows 95 and NT.

## The Dib Class

### Dib (constructor)

**Dib**( *mode*, *size* ). 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

**expose**( *hdc* ). Expose 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

**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

**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 any case, 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 Class

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

## Functions

### PSDraw (constructor)

**PSDraw**( *image* ) sets up printing to the given file. If file is omitted, sys.stdout is assumed.

## ImageDraw Methods

### **begin\_document**

**begin\_document**() sets up printing of a document.

### **end\_document**

Ends printing.

### **line**

**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**

**rectangle**( *box* ) draws a rectangle.

### **text**

**text**( *position*, *text* ) or **text**( *position*, *text*, *alignment* ) draws text at the given position. You must use setfont before calling this method.

### **setfont**

**setfont**( *font*, *size* ) selects which font to use. The font argument is a Postscript font name, the size argument is given in points.

### **setink**

**setink**( *ink* ) selects the pixel value to use with subsequent operations.

### **setfill**

**setfill**( *onoff* ) selects if subsequent rectangle operations should draw filled rectangles or just outlines.

## Image File Formats

The Python Imaging Library supports a wide variety of raster file formats. Some 20 different file formats can be identified by the library, and most of these can be loaded as well. Write support is currently somewhat restricted.

### Summary

In the following table, “open” means that a format can be identified by the library, “draft” that the draft method can be used to control loading, “load” that it can be loaded into memory, and “save” that it can be written from memory. “Experimental” means that the driver has only been tested with a few samples.

<i>Format</i>	<i>Operations</i>
<b>BMP, DIB</b>	open, load, save (“I”, “L”, “P”, “RGB”).
<b>CUR</b>	open, load (“P”).
<b>DCX</b>	open, load (“I”, “L”, “P”, “RGB”).
<b>EPS</b>	open, save (“L”, “RGB”, “CMYK”).
<b>FLI, FLC</b>	open (“P”).
<b>GBR</b>	open, load (“L”).
<b>GD</b>	open, load (“P”).
<b>GIF</b>	open, load (“L”, “P”).
<b>ICO</b>	open, load (“P”).
<b>IM</b>	open, load, save.
<b>IMT</b>	open, load (“L” only).
<b>JFIF, JPEG</b>	open, draft, load, save (“L”, “RGB”, “CMYK”). Requires the IJG JPEG library.
<b>MPEG</b>	open (“RGB”).
<b>MSP</b>	open (“I”).
<b>PBM, PGM, PPM</b>	open, load, save (“I”, “L”, “RGB”).
<b>PCD</b>	open, draft, load (“RGB”, max 768x512).
<b>PCX</b>	open, load (“I”, “L”, “P”, “RGB”).
<b>PDF</b>	save (“I”, “L”, “P”, “RGB”, “CMYK”).
<b>PNG</b>	open, load (“I”, “L”, “P”, “RGB”, “RGBA”; non-interlaced only). Requires the ZLIB library.
<b>PSD</b>	open (“I”, “L”, “P”, “RGB”).
<b>SGI</b>	open, load (“L”, “RGB”; uncompressed only).
<b>SUN</b>	open, load (“I”, “L”, “P”, “RGB”; uncompressed only).
<b>TGA</b>	open, load (“RGB”; 24-bit and 32-bit uncompressed only; experimental).
<b>TIFF</b>	open, load, save (“I”, “L”, “P”, “RGB”, “CMYK”; pixel or plane interleave; striped and tiled; uncompressed, LZW, PackBits or JPEG).
<b>XBM</b>	open, load, save (“I”).
<b>WMF</b>	open, load (“P”; limited rendering support, experimental).

## Format Descriptions

### BMP

The library reads 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.

<i>Info key</i>	<i>Value</i>
<b>compression</b>	Set to “ <b>bmp_rle</b> ” if file is run-length encoded.

### CUR

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

### DCX

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

### EPS

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

### FLI, FLC

The library identifies Autodesk FLI and FLC animations.

<i>Info key</i>	<i>Value</i>
<b>aspect</b>	Aspect ratio, given as a 2-tuple (width, height).
<b>frames</b>	The number of animation frames in the file.
<b>frame_delay</b>	The delay (in fractional seconds) between each frame.

### GBR

The library reads GIMP brush files.

<i>Info key</i>	<i>Value</i>
<b>description</b>	The brush name.

### GD

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.

<i>Info key</i>	<i>Value</i>
<b>transparent</b>	Transparency index. This key is omitted if the image is not transparent.

### GIF

The library reads GIF87a and GIF89a versions of the GIF file format. GIF files are always read as palette mode (“P”) images.

<i>Info key</i>	<i>Value</i>
<b>version</b>	Version (either “GIF87a” or “GIF89a”).

<b>transparent</b>	Transparency index. This key is omitted if the image is not transparent.
--------------------	--

**ICO**

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 most uncompressed interchange versions of this format, but only saves “L” images.

**IMT**

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 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.

<i>Info key</i>	<i>Value</i>
<b>jfif</b>	JFIF application marker found. If the file is not a JFIF file, this key is not present.
<b>adobe</b>	Adobe application marker found. If the file is not an Adobe JPEG file, this key is not present.

*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.*

**MPEG**

The library identifies MPEG files.

**MSP**

The library identifies MSP files from Windows 1 and 2.

**PCD**

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.

**PCX**

The library reads PCX files containing “1”, “L”, “P”, or “RGB” data.

**PDF**

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 mode (and whether JPEG support is available or not).

**PNG**

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

**PPM**

The library reads and writes PBM, PGM and PPM files containing “1”, “L” or “RGB” data. Some commonly used extensions are also supported.

**PSD**

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

**SGI**

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

**SUN**

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

**TGA**

The library reads 24- and 32-bit uncompressed TGA files as “RGB”. This driver is highly experimental.

**TIFF**

The library reads TIFF files containing “1”, “L”, “RGB”, or “CMYK” data. It supports both striped and tiled images, pixel and plane interleaved multi-band images, and either uncompressed, or Packbits, LZW, or JPEG compressed images.

<i>Info key</i>	<i>Value</i>
<b>compression</b>	Compression mode.

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”). Also see the **tobitmap** method, which returns a string containing an XBM representation of an image.

## 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** format looks at the name to determine which format to use, unless the format is given explicitly.

<i>Format</i>	<i>Extensions</i>
<b>BMP</b>	“.bmp”, “.dib”
<b>CUR</b>	“.cur”
<b>DCX</b>	“.dcx”
<b>EPS</b>	“.eps”, “.ps”
<b>GBR</b>	“.gbr”
<b>GD</b>	“.gd”
<b>GIF</b>	“.gif”
<b>ICO</b>	“.ico”
<b>IM</b>	“.im”
<b>IMT</b>	<i>None</i> (must explicitly specify format)
<b>JPEG</b>	“.jpg”, “.jpe”, “.jpeg”
<b>MSP</b>	“.msp”
<b>PCD</b>	“.pcd”
<b>PCX</b>	“.pcx”
<b>PDF</b>	“.pdf”
<b>PNG</b>	“.png”
<b>PPM</b>	“.pbm”, “.pgm”, “.ppm”
<b>PSD</b>	“.psd”
<b>SGI</b>	“.bw”, “.rgb”, “.cmyk”
<b>SUN</b>	“.ras”
<b>TGA</b>	“.tga”
<b>TIFF</b>	“.tif”, “.tiff”
<b>XBM</b>	“.xbm”

## Utilities

This version of the Python Imaging Library contains a small set of utility programs. They are all to be considered as experimental. All improvements are welcome.

### **pilfile**

This utility 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 member. Use the **-t** option to display the tile descriptor (information used to control decoding).

### **pilconvert**

Convert 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.ppm
lena.ppm...

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

### **pilprint**

Print 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(1)` and **-c** to print to a colour printer (otherwise, a colour image is translated to greyscale before being sent to the printer).

## Contributions

Patches, fixes, updates, and new utilities are welcome. If you stumble upon files that the library does not handle as expected, drop me a line and if possible, include the image file. If you fix such a problem and supply a patch, you may send me the file anyway so I don't mess things up again in later revisions.

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

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@cw.nl*** or the newsgroup ***comp.lang.python***.

To contact me directly, use the following e-mail address:

***fredrik\_lundh@ivab.se***



## Software License

The Python Imaging Library is copyright © 1995-96 Fredrik Lundh. All rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose 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 the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL 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.