back next

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

The following script loads an image, rotates it 45 degrees, and displays it using an external viewer (usually **xv** on Unix, and the **paint** program on Windows).

Open, rotate, and display an image (using the default viewer)

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

The following script creates nice 128x128 thumbnails of all JPEG images in the current directory.

Create thumbnails

```
from PIL import Image
import glob, os

size = 128, 128

for infile in glob.glob("*.jpg"):
    file, ext = os.path.splitext(infile)
    im = Image.open(infile)
    im.thumbnail(size, 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 (width, height)-tuple, in pixels. The color is given as a single value for single-band images, and a tuple for multi-band images (with one value for each band). In 1.1.4 and later, you can also use color names (see the **ImageColor** module documentation for details) If the color argument is omitted, the image is filled with zero (this usually corresponds to black). If the color is None, the image is not initialised. This can be useful if you're going to paste or draw things in the image.

```
from PIL import Image
im = Image.new("RGB", (512, 512), "white")
```

open

Image.open(file) ⇒ image

Image.open(file, mode) ⇒ image

Opens and identifies the given image file. This is a lazy operation; the function

reads the file header, but the actual image data is not read from the file until you try to process the data (call the **load** method to force loading). If the mode argument is given, it must be “r”.

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

```
from PIL import Image
im = Image.open("lenna.jpg")
```

```
from PIL import image
from StringIO import StringIO

# read data from string
im = Image.open(StringIO(data))
```

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 the alpha is 0.0, a copy of the first image is returned. If the 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 corresponding pixels from a mask image as alpha. The mask can have mode “1”, “L”, or “RGBA”. All images must be the same size.

eval

Image.eval(image, 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.

frombuffer

Image.frombuffer(mode, size, data) ⇒ image

(New in PIL 1.1.4). Creates an image memory from pixel data in a string or buffer object, using the standard “raw” decoder. For some modes, the image memory will share memory with the original buffer (this means that changes to the original buffer object are reflected in the image). Not all modes can share memory; supported modes include “L”, “RGBX”, “RGBA”, and “CMYK”. For other modes, this function behaves like a corresponding call to the **fromstring** function.

> **Note:** In versions up to and including 1.1.6, the default orientation differs from that of **fromstring**. This may be changed in future versions, so for maximum portability, it’s recommended that you spell out all arguments when using the “raw” decoder:

```
im = Image.frombuffer(mode, size, data, "raw", mode, 0, 1)
```

Image.frombuffer(mode, size, data, decoder, parameters) ⇒ image

Same as the corresponding **fromstring** call.

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 only, not entire images. If you have an entire image file 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

Converts an image to another mode, and returns the new image.

When converting from a palette image, 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.

When converting from a colour image to black and white, the library uses the ITU-R 601-2 luma transform:

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

When converting to a bilevel image (mode "1"), the source image is first converted to black and white. Resulting values larger than 127 are then set to white, and the image is dithered. To use other thresholds, use the **point** method. To disable dithering, use the **dither=** option (see below).

im.convert("P", **options) ⇒ image

Same, but provides better control when converting an "RGB" image to an 8-bit palette image. Available options are:

dither=. Controls dithering. The default is **FLOYDSTEINBERG**, which distributes errors to neighboring pixels. To disable dithering, use **NONE**.

palette=. Controls palette generation. The default is **WEB**, which is the

standard 216-color “web palette”. To use an optimized palette, use **ADAPTIVE**.

colors=. Controls the number of colors used for the palette when **palette** is **ADAPTIVE**. Defaults to the maximum value, 256 colors.

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:

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 copy of 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 get a separate copy, 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 (to be precise, it reconfigures the file reader). 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.

getcolors

im.getcolors() ⇒ a list of (count, color) tuples or None

im.getcolors(maxcolors) ⇒ a list of (count, color) tuples or None

(New in 1.1.5) Returns an unsorted list of (count, color) tuples, where count is the number of times the corresponding color occurs in the image.

If the maxcolors value is exceeded, the method stops counting and returns None. The default maxcolors value is 256. To make sure you get all colors in an image, you can pass in size[0]*size[1] (but make sure you have lots of memory before you do that on huge images).

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, including iteration and basic sequence access. 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 only works for 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.

Note that this method is rather slow; if you need to process larger parts of an image from Python, you can either use pixel access objects (see **load**), or the **getdata** method.

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.

(New in 1.1.6) In 1.1.6 and later, **load** returns a pixel access object that can be used to read and modify pixels. The access object behaves like a 2-dimensional array, so you can do:

```
pix = im.load()
print pix[x, y]
pix[x, y] = value
```

Access via this object is a lot faster than **getpixel** and **putpixel**.

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, and has been removed in PIL 1.2. 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

Returns a copy of the image where each pixel has been mapped through the given lookup 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 the function must have the following format:

```
argument * scale + offset
```

Example:

```
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

Same as above, but specifies a new mode for the output image. This can be used to convert “L” and “P” images to “1” in one step, e.g. to threshold an image.

(New in 1.1.5) This form can also be used to convert “L” images to “I” or “F”, and to convert “I” images with 16-bit data to “L”. In the second case, you must use a 65536-item lookup table.

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”.

(New in PIL 1.1.5) You can use **putalpha** on other modes as well; the image is converted in place, to a mode that matches the current mode but has an alpha layer (this usually means “LA” or “RGBA”). Also, the band argument can be either an image, or a colour value (an integer).

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. For an “L” image, the mode is changed to “P”. 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 a 768-byte 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. If you’re using 1.1.6, pixel access objects (see **load**) provide a faster way to modify the image. If you want to generate an entire image, it can be more efficient to create a Python list and use **putdata** to copy it to the image. For more extensive changes, use **paste** or the **ImageDraw** module instead.

You can speed **putpixel** up a bit by “inlining” the call to the internal **putpixel** implementation method:

```
im.load()
putpixel = im.im.putpixel
for i in range(n):
   ...
   putpixel((x, y), value)
```

In 1.1.6, the above is better written as:

```
pix = im.load()
for i in range(n):
    ...
    pix[x, y] = value
```

quantize

im.quantize(colors, **options) ⇒ image

(Deprecated) Converts an “L” or “RGB” image to a “P” image with the given number of colors, and returns the new image.

For new code, use **convert** with a adaptive palette instead:

```
out = im.convert("P", palette=Image.ADAPTIVE, colors=256)
```

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**.

Note that the bilinear and bicubic filters in the current version of PIL are not well-suited for large downsampling ratios (e.g. when creating thumbnails). You should use **ANTIALIAS** unless speed is much more important than quality.

rotate

im.rotate(angle) ⇒ image

im.rotate(angle, filter=NEAREST, expand=0) ⇒ 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**.

The expand argument, if true, indicates that the output image should be made large enough to hold the rotated image. If omitted or false, the output image has the same size as the input image.

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.

If the save fails, for some reason, the method will raise an exception (usually an **IOError** exception). If this happens, the method may have created the file, and may have written data to it. It’s up to your application to remove incomplete files, if necessary.

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.

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**.

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.

> **Note:** The **tostring** method only fetches the raw pixel data. To save the image to a string in a standard file format, pass a StringIO object (or equivalent) to the **save** method.

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), **MESH** (map a number of source quadrilaterals in one operation), or **PERSPECTIVE**. 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.

im.transform(size, PERSPECTIVE, data) image ⇒ image

im.transform(size, PERSPECTIVE, data, filter) image ⇒ image

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

Data is a 8-tuple (*a, b, c, d, e, f, g, h*) which contains the coefficients for a perspective transform. For each pixel (*x, y*) in the output image, the new value is taken from a position (a *x* + b *y* + c)/(g *x* + h *y* + 1), (d *x* + e *y* + f)/(g *x* + h *y* + 1) in the input image, rounded to nearest pixel.

This function can be used to change the 2D perspective of the original image.

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. This method only works on a newly opened image; if the image has already been loaded, the result is undefined. Also, if you need to load the image after using this method, you must reopen the image file.

Note that this method doesn't catch all possible errors; to catch decoding errors, you may have to load the entire image as well.

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 itself (via a factory function, or by running a method on an existing image), 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." See *Concepts* for a full list.

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. This dictionary is used by file handlers to pass on various non-image information read from the file. See documentation for the various file handlers for details.

Most methods ignore the dictionary when returning new images; since the keys are not standardized, it’s not possible for a method to know if the operation affects the dictionary. If you need the information later on, keep a reference to the **info** dictionary returned from the **open** method.

back next