# Python Imaging Library (PIL)


John W. Shipman

2008-01-16 23:30

**Abstract**

Describes an image-processing library for the Python programming language.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

## Table of Contents



## 1. Introduction

The Python Imaging Library (PIL) allows you to create, modify, and convert image files in a wide variety of formats using the Python language.

For additional PIL features not described in this document, refer to the online *PIL handbook*[3].

For more information about Python, refer to the author's companion publication, *Python programming language quick reference*[4], or to the Python Web site[5], for general information about the Python language.

[1] http://www.nmt.edu/tcc/help/pubs/pil/
[2] http://www.nmt.edu/tcc/help/pubs/pil/pil.pdf
[3] http://www.pythonware.com/library/pil/handbook/index.htm
[4] http://www.nmt.edu/tcc/help/pubs/python22/
[5] http://www.python.org/

# 2. Definitions

These terms are used throughout:

## 2.1. Band

An image *band* is a set of values, one per image pixel. Monochrome or grayscale images have one band; color images in the RGB system have three bands, CMYK images have four, and so on. Photoshop users will recognize bands as similar to Photoshop *channels*.

## 2.2. Modes

The *mode* of an image describes the way it represents colors. Each mode is represented by a string:

| Mode | Bands | Description |
|---|---|---|
| `"1"` | 1 | Black and white (monochrome), one bit per pixel. |
| `"L"` | 1 | Gray scale, one 8-bit byte per pixel. |
| `"P"` | 1 | Palette encoding: one byte per pixel, with a palette of class `ImagePalette` translating the pixels to colors. This mode is experimental; refer to the online documentation[6]. |
| `"RGB"` | 3 | True red-green-blue color, three bytes per pixel. |
| `"RGBA"` | 4 | True color with a transparency band, four bytes per pixel, with the `A` channel varying from 0 for transparent to 255 for opaque. |
| `"CMYK"` | 4 | Cyan-magenta-yellow-black color, four bytes per pixel. |
| `"YCbCr"` | 3 | Color video format, three 8-bit pixels. |
| `"I"` | 1 | 32-bit integer pixels. |
| `"F"` | 1 | 32-bit float pixels. |

## 2.3. Sizes

The sizes of objects in the image are described as a 2-tuple `(`*w*`, `*h*`)`, where *h* is the height in pixels and *w* is the width.

## 2.4. Coordinates

The coordinates of a pixel are of its upper left corner. Point (0,0) is the upper left corner of the image. The *x* coordinate increases to the right, and the *y* coordinate increases downward.

When directions are given as compass points such as east or southwest, assume north is up, toward the top of the display.

## 2.5. Angles

Angles are given in degrees. Zero degrees is in the +*x* (east) direction, and the angle increases counter-clockwise, in the usual Cartesian convention. For example, angle 45 points northeast.

[6] http://www.pythonware.com/library/pil/handbook/index.htm

## 2.6. Bboxes (bounding boxes)

A bounding box or *bbox* is a rectangle in the image. It is defined by a 4-tuple, $(x_0, y_0, x_1, y_1)$ where $(x_0, y_0)$ is the top left (northwest) corner of the rectangle, and $(x_1, y_1)$ is the bottom right (southeast) corner.

Generally, the area described by a bounding box will include point $(x_0, y_0)$, but it will *not* include point $(x_1, y_1)$ or the row and column of pixels containing point $(x_1, y_1)$.

For example, drawing an ellipse inside the bounding box `(0,0,5,10)` will produce an ellipse 5 pixels wide and 10 pixels high. The resulting ellipse will include pixel column 4 but not column 5, and will also include pixel row 9 but not row 10.

## 2.7. Colors

You can specify colors in several different ways.

- For single-band images, the color is the pixel value. For example, in a mode `"1"` image, the color is a single integer, 0 for black, 1 for white. For mode `"L"`, it is an integer in the range [0,255], where 0 means black and 255 means white.
- For multi-band images, supply a tuple with one value per band. In an `"RGB"` image, the tuple `(255,0,0)` is pure red.
- You can use CSS-style color name strings of the form `#`*`rrggbb`*, where *`rr`* specifies the red part as two hexadecimal digits, *`gg`* specifies green, and *`bb`* blue. For example, `"#ffff00"` means yellow (full red + full green).
- To specify RGB pixel values in decimal, use a string of the form `"rgb(`*`R`*`,`*`G`*`,`*`B`*`)"`. For example, `"rgb(0,255,0)"` is pure green.
- To specify RGB pixel values as percentages, use a string of the form `"rgb(`*`R`*`%,`*`G`*`%,`*`B`*`%)"`. For example, you can get a light gray with `"rgb(85%,85%,85%)"`.
- To specify colors in the hue-saturation-lightness (HSV) system, use a string of the form `"hsl(`*`H`*`,`*`S`*`%,`*`L`*`%)"`.

  *`H`* is the hue angle in degrees: 0 is red, 60 is yellow, 120 is green, and so on.

  *`S`* is the saturation: `0%` for unsaturated (gray), `100%` for fully saturated.

  The lightness *`L`* is `0%` for black, `50%` for normal, and `100%` for white.

  For example, `"hsl(180,100%,50%)"` is pure cyan.
- On Unix systems, you can use any of the standard color names from the locally installed set given in file `"/usr/lib/X11/rgb.txt"`, such as `"white"`, `"DodgerBlue"`, or `"coral"`.

## 2.8. Filters

Some operations that reduce the number of pixels, such as creating a thumbnail, can use different filters to compute the new pixel values. These include:

**`NEAREST`**
: Uses the value of the nearest pixel.

**`BILINEAR`**
: Uses linear interpolation over a 2x2 set of adjacent pixels.

**BICUBIC**
Uses cubic interpolation over a 4x4 set of pixels.

**ANTIALIAS**
Neighboring pixels are resampled to find the new pixel value.

# 3. Creating objects of class Image

Instances of the `Image` class contain an image. To use the basic Python imaging library, import it using this syntax:

```
import Image
```

These functions return an `Image` object:

**Image.open(*f*)**
Reads an image from a file. The parameter *f* can be either the name of a file, as a string, or a readable Python file object.

**Image.new(*mode*,*size*,*color*=None)**
Creates a new image of the given mode (p. 2) and size (p. 2). If a color (p. 3) is given, all pixels are set to that color initially; the default color is black.

**Image.blend($i_1$,$i_2$,*a*)**
Creates an image by blending two images $i_1$ and $i_2$. Each pixel in the output is computed from the corresponding pixels $p_1$ from $i_1$ and $p_2$ from $i_2$ using the formula ($p_1$ x (1 - *a*) + $p_2$ x *a*).

**Image.composite($i_1$,$i_2$,*mask*)**
Creates a composite image from two equal-sized images $i_1$ and $i_2$, where *mask* is a mask image with mode `"1"`, `"L"`, or `"RGBA"` of the same size.

Each pixel in the output has a value given by ($p_1$ x (1 - *m*) + $p_2$ x *m*), where *m* is the corresponding pixel from *mask*.

**Image.eval(*f*,*i*)**
Creates a new image by applying a function *f* to each pixel of image *i*.

Function *f* takes one argument and returns one argument. If the image has multiple bands, *f* is applied to each band.

**Image.merge(*mode*,*bandList*)**
Creates a multi-band image from multiple single-band images of equal size. Specify the mode (p. 2) of the new image with the *mode* argument. The *bandList* argument is a sequence of single-band image objects to be combined.

## 3.1. Attributes of the Image object

These attributes of image objects are available:

**.format**
If the image was taken from a file, this attribute is set to the format code, such as `"GIF"`. If the image was created within the PIL, its `.format` will be `None`. See supported formats (p. 11) for a list of the format codes.

**.mode**
The mode (p. 2) of the image, as a string.

**.size**
The image's size (p. 2) in pixels, as 2-tuple (*width*,*height*).

**.palette**
If the mode of the image is "P", this attribute will be the image's palette as an instance of class ImagePalette; otherwise it will be None.

**.info**
A dictionary holding data associated with the image. The exact information depends on the source image format; see the section on supported formats (p. 11) for more information.

## 3.2. Methods on the Image object

Once you have created an image object using one of the functions described above, these methods provide operations on the image:

**.save(*f*,*format*=None)**
Writes the image to a file. The argument *f* can be the name of the file to be written, or a writeable Python file object.

The *format* argument specifies the kind of image file to write. It must be one of the format codes shown below under supported formats (p. 11), such as "JPEG" or "TIFF". This argument is required if *f* is a file object. If *f* is a file name, this argument can be omitted, in which case the format is determined by inspecting the extension of the file name. For example, if the file name is logo.jpg, the file will be written in JPEG format.

**.convert(*mode*)**
Returns a new image with a different mode (p. 2). For example, if image im has mode "RGB", you can get a new image with mode "CMYK" using the expression im.convert("CMYK").

**.copy()**
Returns a copy of the image.

**.crop(*bbox*)**
Returns a new image containing the bounding box (p. 3) specified by *bbox* in the original. The rows and columns specified by the third and fourth values of *bbox* will *not* be included. For example, if image im has size 4x4, im.crop((0,0,3,3)) will have size 3x3.

**.filter(*name*)**
Return a copy of the image filtered through a named image enhancement filter. See image filter (p. 9) for a list of the predefined image enhancement filters.

**.getbands()**
Returns a sequence of strings, one per band, representing the mode (p. 2) of the image. For example, if image im has mode "RGB", im.getbands() will return ('R', 'G', 'B').

**.getbbox()**
Returns the smallest bounding box (p. 3) that encloses the nonzero parts of the image.

**.getextrema()**
For single-band images, returns a tuple (*min*,*max*) where *min* is the smallest pixel value and *max* the largest pixel value in the image.

For multi-band images, returns a tuple containing the (*min*,*max*) tuple for each band.

**.getpixel(*xy*)**

Returns the pixel value or values at the given coordinates (p. 2). For single-band images, returns a single number; a sequence of pixel values is returned for multi-band images. For example, if the top left pixel of an image `im` of mode `"L"` has value 128, then `im.getpixel((0,0))` returns 128.

**.histogram(*mask*=None)**

For single-band images, returns a list of values $[c_0, c_1, ...]$ where $c_i$ is the number of pixels with value $i$.

For multi-band images, returns a single sequence that is the concatenation of the sequences for all bands.

To get a histogram of selected pixels from the image, build a same-sized mask image of mode `"1"` or `"L"` and pass it to this method as an argument. The resulting histogram data will include only those pixels from the image that correspond to nonzero pixels in the *mask* argument.

**.offset(*dx*,*dy*=None)**

Returns a new image the same size as the original, but with all pixels rotated *dx* in the *+x* direction, and *dy* in the *+y* direction. Pixels will wrap around. If *dy* is omitted, it defaults to the same value as *dx*.

For example, if `im` is an image of size 4x4, then in the new image `im.offset(2,1)`, the pixel that was at `(1,1)` will now be at `(3,2)`. Wrapping around, the pixel that was at `(3,3)` will now be at `(0,1)`.

**.paste(*i2*,*where*,*mask*=None)**

Pixels are replaced with the pixels from image *i2*.

- If *where* is a pair of coordinates (p. 2) (*x*,*y*), the new pixels are pasted so that the (0,0) pixel of *i2* pixel replaces the (*x*,*y*) pixel, with the other pixels of *i2* pasted in corresponding positions. Pixels from *i2* that fall outside the original are discarded.
- If *where* is a bounding box (p. 3), *i2* must have the same size as that bbox, and replaces the pixels at that bbox's position.
- If *where* is `None`, image *i2* must be the same size as the original, and replaces the entire original.

If the mode of *i2* is different, it is converted before pasting.

If a *mask* argument is supplied, it is used to control which pixels get replaced. The mask argument must be an image of the same size as the original.

- If the mask has mode `"1"`, original pixels are left alone where there are 0 pixels in the mask, but replaced where the mask has pixels of value 1.
- If the mask has mode `"L"`, original pixels are left alone where there are 0 pixels in the mask, and replaced where the mask has 255 pixels. Intermediate values interpolate between the pixel values of the original and replacement images.
- If the mask has `"RGBA"` mode, its `"A"` band is used in the same way as an `"L"`-mode mask.

**.paste(*color*,*box*=None,*mask*=None)**

Sets multiple pixels to the same color. If the *box* argument is omitted, the entire image is set to that color. If present, it must be a bounding box (p. 3), and the rectangle defined by that bbox is set to that color.

If a *mask* is supplied, it selects the pixels to be replaced as in the previous form of the `.paste()` method, above.

**.point(*function*)**

Returns a new image with each pixel modified. To apply a function *f(x)* to each pixel, call this method with *f* as the first argument. The function should operate on a single pixel value (0 to 255) and return the new pixel value.

**.point(*table*)**
To translate pixels using a table lookup, pass that table as a sequence to the first argument. The table should be a sequence of 256*n* values, where *n* is the number of bands in the image. Each pixel in band *b* of the image is replaced by the value from *table*[*p*+256**b*], where *p* is the old pixel's value in that band.

**.putalpha(*band*)**
To add an alpha (transparency) band to an "RGBA"-mode image, call this method and pass it an image of the same size having mode "L" or "1". The pixels of the *band* image replace the alpha band of the original image in place.

**.putpixel(*xy*, *color*)**
Replaces one pixel of the image at coordinates (p. 2) *xy*. For single-band images, specify the *color* as a single pixel value; for multi-band images, provide a sequence containing the pixel values of each band.

For example, im.putpixel((0,0), (0,255,0)) would set the top left pixel of an RGB image im to green.

Replacing many pixels with this method may be slow. Refer to the ImageDraw module for faster techniques for pixel modification.

**.resize(*size*,*filter*=None)**
Returns a new image of the given *size* by linear stretching or shrinking of the original image. You may provide a filter (p. 3) to specify how the interpolation is done; the default is NEAREST.

**.rotate(*theta*)**
Returns a new image rotated around its center by *theta* degrees. Any pixels that are not covered by rotation of the original image are set to black.

**.show()**
On Unix systems, this method runs the *xv* image viewer to display the image. On Windows boxes, the image is saved in BMP format and can be viewed using Paint. This can be useful for debugging.

**.split()**
Returns a tuple containing each band of the original image as an image of mode "L". For example, applying this method to an "RGB" image produces a tuple of three images, one each for the red, green, and blue bands.

**.thumbnail(*size*,*filter*=None)**
Replaces the original image, in place, with a new image of the given size (p. 2). The optional *filter* argument works in the same way as in the .resize() method.

The aspect ratio (height : width) is preserved by this operation. The resulting image will be as large as possible while still fitting within the given size. For example, if image im has size (400,150), its size after im.thumbnail((40,40)) will be (40,15).

**.transform($x_s$, $y_s$, Image.EXTENT, ($x_0$,$y_0$,$x_1$,$y_1$))**
Returns a transformed copy of the image. In the transformed image, the point originally at ($x_0$,$y_0$) will appear at (0,0), and point ($x_1$,$y_1$) will appear at ($x_s$, $y_s$).

**.transform($x_s$, $y_s$, Image.AFFINE, (*a*,*b*,*c*,*d*,*e*,*f*))**
Affine transformation. The values *a* through *f* are the first two rows of an affine transform matrix. Each pixel at (*x*,*y*) in the resulting image comes from position (*ax*+*by*+*c*,*dx*+*ey*+*f*) in the input image, rounded to the nearest pixel.

**.transpose(*method*)**
Return a flipped or rotated copy of the original image. The *method* can be any of:
- Image.FLIP_RIGHT_LEFT to reflect around the vertical centerline.

- `Image.FLIP_TOP_BOTTOM` to reflect around the horizontal centerline.
- `Image.ROTATE_90` to rotate the image 90 degrees clockwise.
- `Image.ROTATE_180` to rotate the image 180 degrees.
- `Image.ROTATE_270` to rotate the image clockwise by 270 degrees.

# 4. The `ImageDraw` module

When you need to draw on an image, import the `ImageDraw` module like this:

```
    import ImageDraw
```

Then instantiate a `Draw` object:

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

where *`i`* is the `Image` object you want to draw on.

Methods on a `Draw` object include:

**`.arc(`*`bbox`*`,`*`start`*`,`*`end`*`,`*`fill`*`=None)`**
Draws an arc, that part of the ellipse fitting inside the bounding box (p. 3) *`bbox`* and lying between angles *`start`* and *`end`*. *Note*: Angles increase clockwise, unlike angles elsewhere in the PIL. For example, `draw.arc((0,0,200,200), 0, 135)` would draw a circular arc centered at (100,100) and extending from the east to the southwest.

If supplied, the *`fill`* argument specifies the color of the arc. The default color is white.

**`.chord(`*`bbox`*`,`*`start`*`,`*`end`*`,`*`fill`*`=None,`*`outline`*`=None)`**
Same as the `.arc()` method, but it also draws a straight line connecting the endpoints of the arc.

For this method, the *`fill`* argument determines the color inside, that is, between the chord and the arc. The default is that this area is not filled.

To change the color of the perimeter border around the chord's area, set the *`outline`* argument to the color you want. The default color is white.

**`.ellipse(`*`bbox`*`,`*`fill`*`=None,`*`outline`*`=None)`**
Draws the ellipse that fits inside the bounding box (p. 3) *`bbox`*. To draw a circle, use a square bounding box.

Note that the ellipse will include the left and top sides of the bounding box, but they will *exclude* the right and bottom sides of the box. For example, a bounding box `(0,0,10,10)` will give you a circle of diameter 10, not diameter 11.

If you omit the *`fill`* argument, only the perimeter of the ellipse is drawn. If you pass a color to this argument, the interior of the ellipse will be filled with that color.

Use *`outline`*`=`*`c`* to draw the perimeter border using color *`c`*. The default color is white.

**`.line(`*`L`*`, `*`fill`*`=None)`**
Draws one or more line segments. The argument *`L`* specifies the endpoints, and can have either of these forms:

- A sequence of 2-element sequences, each of which specifies one endpoint. For example, `draw.line([(10,20),(100,20)]` would draw a straight line from (10,20) to (100,20). You can specify any number of points to get a "polyline;" for example, `draw.line(((60,60), (90,60), (90,90), (60,90), (60,60)))` would draw a square 30 pixels on a side.

- A sequence containing an even number of values. Each succeeding pair is taken as an *(x,y)* coordinate. For example, `draw.line((10,20,100,20))` would give you the same result as the first example in the paragraph above.

**.pieslice(*bbox*,*start*,*end*,*fill*=None,*outline*=None)**
Similar to the `.arc()` method, but draws two radii connecting the endpoints of the arc to the center. The *fill* and *outline* arguments work in the same way as in the `.chord()` method: *fill* determines the interior color, and *outline* sets the color of the border.

**.point(*xy*,*fill*=None)**
Sets the pixel at coordinates (p. 2) *xy* to the color specified by the *fill* argument. The default color is white.

**.polygon(*L*, *fill*=None, *outline*=None)**
Works like the `.line()` method, but after drawing all the specified line segments, it draws one more that connects the last point back to the first. The interior displays the *fill*, transparent by default. The border is drawn in the *outline* color, defaulting to white.

For example, `draw.polygon([(60,60), (90,60), (90,90), (60,90)], fill="red", outline="green")` would draw a square box with a green outline, filled with red.

**.text(*xy*,*message*,*fill*=None,*font*=None)**
Draws the text of the string *message* on the image with its upper left corner at coordinates (p. 2) *xy*.

To write the text in color, pass that color to the *fill* argument; the default text color is white.

There is a default font, rather small (about 11 pixels tall) and in a serif style. You can also specify a font using the *font* argument; see the `ImageFont` module for more information on fonts.

**.textsize(*message*,*font*=None)**
For a given text string *message*, returns a tuple `(w,h)` where *w* is the width in pixels that text will occupy on the display, and *h* its the height in pixels.

If the *font* argument is omitted, you will get the size using the default font. Supply a font to this method's *font* argument to get the size of the text in that font.

# 5. Image enhancement: the `ImageFilter` module

Certain operations allow you to filter the image data using one of a set of predefined filters. Use this form of import:

```
    import ImageFilter
```

Once you have imported the module, you can use any of these filters:

- `ImageFilter.BLUR`
- `ImageFilter.CONTOUR`
- `ImageFilter.DETAIL`
- `ImageFilter.EDGE_ENHANCE`
- `ImageFilter.EDGE_ENHANCE_MORE`
- `ImageFilter.EMBOSS`
- `ImageFilter.FIND_EDGES`
- `ImageFilter.SMOOTH`
- `ImageFilter.SMOOTH_MORE`
- `ImageFilter.SHARPEN`

# 6. The ImageFont module

To specify fonts for the `.text()` method from the `ImageDraw` module, import this module:

```
    import ImageFont
```

You can then create a font object from any TrueType font by using this function:

**`ImageFont.truetype(`*`file`*`,`*`size`*`)`**
Returns a font object representing the TrueType font whose file name is given by the *`file`* argument, with a font height of *`size`* pixels.

Methods on font objects include:

**`.getsize(`*`text`*`)`**
For a given string *`text`*, returns a tuple `(`*`w`*`,`*`h`*`)` where *`w`* is the width in pixels that text will occupy on the display, and *`h`* its the height in pixels.

On Unix systems locally, TrueType fonts can be found in this directory:

```
/usr/share/fonts/
```

At this writing, there were two families of public-domain fonts:

- Deja Vu fonts: `/usr/share/fonts/dejavu-lgc/`.
- Liberation fonts: `/usr/share/fonts/liberation/`.

Here's a complete program that creates a 200x50 gray image, writes text on it in red, and saves it to file `runaway.jpg`. File `DejaVuLGCSansCondensed-Bold.ttf` is sans-serif, condensed, bold font in the Deja Vu family.

```
#!/usr/local/bin/python
import Image
import ImageDraw
import ImageFont

fontPath = "/usr/share/fonts/dejavu-lgc/DejaVuLGCSansCondensed-Bold.ttf"
sans16  =  ImageFont.truetype ( fontPath, 16 )

im  =  Image.new ( "RGB", (200,50), "#ddd" )
draw  =  ImageDraw.Draw ( im )
draw.text ( (10,10), "Run awayyyy!", font=sans16, fill="red" )
im.save ( "runaway.jpg" )
```

On Windows systems, look in `C:\WINDOWS\Fonts`.

# 7. The ImageTk module

This module is for use with the graphical user interface functions of the Tkinter package. For more information about the Tkinter widget set, see the local Tkinter page[7].

To import the `ImageTk` module:

[7] http://www.nmt.edu/tcc/help/lang/python/tkinter.html

```
    import ImageTk
```

This module contains two class constructors:

**ImageTk.BitmapImage(*i*,***options*)**
Given an image *i*, constructs a BitMapImage object that can be used wherever Tkinter expects an image object. The image must have mode "1". Any keyword *options* are passed on to Tkinter. For example, the option foreground=*c* can be used to display the pixels of value 1 in color *c*, while pixels of value 0 are transparent.

**ImageTk.PhotoImage(*i*)**
Given an image *i*, constructs a PhotoImage object that can be used wherever Tkinter expects an image object.

## Warning

There is a bug in the current version of the Python Imaging Library that can cause your images not to display properly. When you create an object of class PhotoImage, the reference count for that object does not get properly incremented, so unless you keep a reference to that object somewhere else, the PhotoImage object may be garbage-collected, leaving your graphic blank on the application.

For example, if you have a canvas or label widget that refers to such an image object, keep a list named .imageList in that object, and append all PhotoImage objects to it as they are created. If your widget may cycle through a large number of images, you will also want to delete the objects from this list when they are no longer in use.

**ImageTk.PhotoImage(*mode*, *size*)**
Creates an empty PhotoImage object with the given mode (p. 2) and size (p. 2).

To change image data in a PhotoImage object, use this method:

**.paste(*i2*, *bbox*=None)**
Image data from an Image object *i2* is pasted into the PhotoImage object. To paste the new data into a given location, specify the *bbox* argument as a bounding box (p. 3); this argument can be omitted if *i2* is the same size as the PhotoImage object.

# 8. Supported file formats

These are the kinds of image files supported by the PIL. The first column gives the PIL file type code, as used in the Image.format attribute and the Image.save() method (although not all types can be saved). The second column shows the file extensions associated with that type. The next two columns show which modes can be read (if any) and written (if any), or the word "None" if that file type cannot be read or written.

| Type | Extensions | Read modes | Write modes | Remarks |
|---|---|---|---|---|
| "BMP" | .bmp .dib | 1 L P RGB | 1 L P RGB | |
| "DCX" | .dcx | 1 L P RGB | None | Fax format. Only the first image is read. |
| "EPS" | .eps .ps | None | L RGB | Encapsulated PostScript |
| "GIF" | .gif | P | P | |

| Type | Extensions | Read modes | Write modes | Remarks |
|---|---|---|---|---|
| "IM" | .im | All | All | Used by LabEye and other applications. |
| "JPEG" | .jpg .jpe .jpeg | L RGB CMYK | L RGB CMYK | |
| "PCD" | .pcd | RGB | None | Photo CD format; will read the 768x512 resolution. |
| "PCX" | .pcx | 1 L P RGB | 1 L P RGB | |
| "PDF" | .pdf | None | 1 RGB | Adobe Page Description Format. |
| "PNG" | .png | 1 L P RGB RGBA | 1 L P RGB RGBA | Portable Network Graphics format. |
| "PPM" | .pbm .pgm .ppm | 1 L RGB | 1 L RGB | |
| "PSD" | .psd | P | None | Photoshop format. |
| "TIFF" | .tif .tiff | 1 L RGB CMYK | 1 L RGB CMYK | |
| "XBM" | .xbm | 1 | 1 | X bitmap files. |
| "XPM" | .xpm | P | P | X pixmap files with up to 256 colors. |

This table includes only some of the more common image formats. Refer to the online documentation[8] for a full list.

[8] http://www.pythonware.com/library/pil/handbook/index.htm