Pillow (PIL fork) Documentation

Release 2.7.0

Author

September 21, 2015

Contents

Pillow is the 'friendly' PIL fork by Alex Clark and Contributors. PIL is the Python Imaging Library by Fredrik Lundh and Contributors. To install Pillow, please follow the installation instructions. To download source and/or contribute to development of Pillow please see: https://github.com/python-pillow/Pillow.

CHAPTER 1

Installation

Warning: Pillow >= 2.1.0 no longer supports "import _imaging". Please use "from PIL.Image import core as _imaging" instead.

Warning: Pillow >= 1.0 no longer supports "import Image". Please use "from PIL import Image" instead.

Warning: PIL and Pillow currently cannot co-exist in the same environment. If you want to use Pillow, please remove PIL first.

Note: Pillow >= 2.0.0 supports Python versions 2.6, 2.7, 3.2, 3.3, 3.4

Note: Pillow < 2.0.0 supports Python versions 2.4, 2.5, 2.6, 2.7.

1.1 Simple installation

Note: The following instructions will install Pillow with support for most formats. See *External libraries* for the features you would gain by installing the external libraries first. This page probably also include specific instructions for your platform.

You can install Pillow with **pip**:

```
$ pip install Pillow
```

Or **easy_install** (for installing Python Eggs, as **pip** does not support them):

```
$ easy_install Pillow
```

Or download the compressed archive from PyPI, extract it, and inside it run:

```
$ python setup.py install
```

1.2 External libraries

Note: You *do not* need to install all of the external libraries to use Pillow's basic features.

Many of Pillow's features require external libraries:

- **libjpeg** provides JPEG functionality.
  - Pillow has been tested with libjpeg versions **6b**, **8**, and **9** and libjpeg-turbo version **8**.
- **zlib** provides access to compressed PNGs
- **libtiff** provides compressed TIFF functionality
  - Pillow has been tested with libtiff versions **3.x** and **4.0**
- **libfreetype** provides type related services
- **littlecms** provides color management
  - Pillow version 2.2.1 and below uses liblcms1, Pillow 2.3.0 and above uses liblcms2. Tested with **1.19** and **2.2**.
- **libwebp** provides the WebP format.
  - Pillow has been tested with version **0.1.3**, which does not read transparent WebP files. Versions **0.3.0** and **0.4.0** support transparency.
- **tcl/tk** provides support for tkinter bitmap and photo images.
- **openjpeg** provides JPEG 2000 functionality.
  - Pillow has been tested with openjpeg **2.0.0** and **2.1.0**.

Once you have installed the prerequisites,run:

```
$ pip install Pillow
```

If the prerequisites are installed in the standard library locations for your machine (e.g. `/usr` or `/usr/local`), no additional configuration should be required. If they are installed in a non-standard location, you may need to configure setuptools to use those locations by editing `setup.py` or `setup.cfg`, or by adding environment variables on the command line:

```
$ CFLAGS="-I/usr/pkg/include" pip install pillow
```

1.3 Build Options

- Environment Variable: `MAX_CONCURRENCY=n`. By default, Pillow will use multiprocessing to build the extension on all available CPUs, but not more than 4. Setting `MAX_CONCURRENCY` to 1 will disable parallel building.
- Build flags: `--disable-zlib`, `--disable-jpeg`, `--disable-tiff`, `--disable-freetype`, `--disable-tcl`, `--disable-tk`, `--disable-lcms`, `--disable-webp`, `--disable-webpmux`, `--disable-jpeg2000`. Disable building the corresponding feature even if the development libraries are present on the building machine.
- Build flags: `--enable-zlib`, `--enable-jpeg`, `--enable-tiff`, `--enable-freetype`, `--enable-tcl`, `--enable-tk`, `--enable-lcms`, `--enable-webp`, `--enable-webpmux`, `--enable-jpeg2000`. Require that the corresponding feature is built. The build will raise an exception if the libraries are not found. Webpmux (WebP metadata) relies on WebP support. Tcl and Tk also must be used together.

Sample Usage:

```
$ MAX_CONCURRENCY=1 python setup.py build-ext --enable-[feature] install
```

1.4 Linux installation

Note: Fedora, Debian/Ubuntu, and ArchLinux include Pillow (instead of PIL) with their distributions. Consider using those instead of installing manually.

We do not provide binaries for Linux. If you didn't build Python from source, make sure you have Python's development libraries installed. In Debian or Ubuntu:

```
$ sudo apt-get install python-dev python-setuptools
```

Or for Python 3:

```
$ sudo apt-get install python3-dev python3-setuptools
```

In Fedora, the command is:

```
$ sudo yum install python-devel
```

Prerequisites are installed on **Ubuntu 12.04 LTS** or **Raspian Wheezy 7.0** with:

```
$ sudo apt-get install libtiff4-dev libjpeg8-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.5-dev tk8.5-dev python-tk
```

Prerequisites are installed on **Ubuntu 14.04 LTS** with:

```
$ sudo apt-get install libtiff5-dev libjpeg8-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev python-tk
```

Prerequisites are installed on **Fedora 20** with:

```
$ sudo yum install libtiff-devel libjpeg-devel libzip-devel freetype-devel \
    lcms2-devel libwebp-devel tcl-devel tk-devel
```

1.5 Mac OS X installation

We provide binaries for OS X in the form of Python Wheels. Alternatively you can compile Pillow with with XCode.

The easiest way to install external libraries is via Homebrew. After you install Homebrew, run:

```
$ brew install libtiff libjpeg webp little-cms2
```

Install Pillow with:

```
$ pip install Pillow
```

1.6 Windows installation

We provide binaries for Windows in the form of Python Eggs and Python Wheels:

1.6.1 Python Eggs

Note: **pip** does not support Python Eggs; use **easy_install** instead.

```
$ easy_install Pillow
```

1.6.2 Python Wheels

Note: Experimental. Requires setuptools >=0.8 and pip >=1.4.1

```
$ pip install --use-wheel Pillow
```

If the above does not work, it's likely because we haven't uploaded a wheel for the latest version of Pillow. In that case, try pinning it to a specific version:

```
$ pip install --use-wheel Pillow==2.6.1
```

1.7 FreeBSD installation

Note: Only FreeBSD 10 tested

Make sure you have Python's development libraries installed.:

```
$ sudo pkg install python2
```

Or for Python 3:

```
$ sudo pkg install python3
```

Prerequisites are installed on **FreeBSD 10** with:

```
$ sudo pkg install jpeg tiff webp lcms2 freetype2
```

1.8 Platform support

Current platform support for Pillow. Binary distributions are contributed for each release on a volunteer basis, but the source should compile and run everywhere platform support is listed. In general, we aim to support all current versions of Linux, OS X, and Windows.

Note: Contributors please test on your platform, edit this document, and send a pull request.

Operating system	Supported	Tested Python versions	Tested Pillow versions	Tested processors
Mac OS X 10.10 Yosemite				x86-64
Mac OS X 10.9 Mavericks	Yes	2.7,3.4	2.6.1	x86-64
Mac OS X 10.8 Mountain Lion	Yes	2.6,2.7,3.2,3.3		x86-64
Redhat Linux 6	Yes	2.6		x86
CentOS 6.3	Yes	2.7,3.3		x86
Fedora 20	Yes	2.7,3.3	2.3.0	x86-64
Ubuntu Linux 10.04 LTS	Yes	2.6	2.3.0	x86,x86-64
Ubuntu Linux 12.04 LTS	Yes	2.6,2.7,3.2,3.3,PyPy2.4, PyPy3,v2.3 2.7,3.2	2.6.1 2.6.1	x86,x86-64 ppc
Ubuntu Linux 14.04 LTS	Yes	2.7,3.2,3.3,3.4	2.3.0	x86
Raspian Wheezy	Yes	2.7,3.2	2.3.0	arm
Gentoo Linux	Yes	2.7,3.2	2.1.0	x86-64
FreeBSD 10	Yes	2.7,3.4	2.4,2.3.1	x86-64
Windows 7 Pro	Yes	2.7,3.2,3.3	2.2.1	x86-64
Windows Server 2008 R2 Enterprise	Yes	3.3		x86-64
Windows 8 Pro	Yes	2.6,2.7,3.2,3.3,3.4a3	2.2.0	x86,x86-64
Windows 8.1 Pro	Yes	2.6,2.7,3.2,3.3,3.4	2.3.0, 2.4.0	x86,x86-64

1.9 Old Versions

You can download old distributions from PyPI. Only the latest 1.x and 2.x releases are visible, but all releases are available by direct URL access e.g. https://pypi.python.org/pypi/Pillow/1.0.

CHAPTER 2

About Pillow

2.1 Goals

The fork authors' goal is to foster active development of PIL through:

- Continuous integration testing via Travis CI
- Publicized development activity on GitHub
- Regular releases to the Python Package Index

2.2 License

Like PIL itself, Pillow is licensed under the MIT-like *PIL Software License <http://www.pythonware.com/products/pil/license.htm>*:

```
Software License

The Python Imaging Library (PIL) is

    Copyright © 1997-2011 by Secret Labs AB
    Copyright © 1995-2011 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that

Permission to use, copy, modify, and distribute this software and its associated documentation for an

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IN
```

2.3 Why a fork?

PIL is not setuptools compatible. Please see this Image-SIG post for a more detailed explanation. Also, PIL's current bi-yearly (or greater) release schedule is too infrequent to accommodate the large number and frequency of issues reported.

2.4 What about PIL?

Note: Prior to Pillow 2.0.0, very few image code changes were made. Pillow 2.0.0 added Python 3 support and includes many bug fixes from many contributors.

As more time passes since the last PIL release, the likelihood of a new PIL release decreases. However, we've yet to hear an official "PIL is dead" announcement. So if you still want to support PIL, please report issues here first, then open the corresponding Pillow tickets here.

Please provide a link to the PIL ticket so we can track the issue(s) upstream.

CHAPTER 3

Guides

3.1 Overview

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

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

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

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

3.1.1 Image Archives

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

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

3.1.2 Image Display

The current release includes Tk *PhotoImage* and *BitmapImage* interfaces, as well as a *Windows DIB interface* that can be used with PythonWin and other Windows-based toolkits. Many other GUI toolkits come with some kind of PIL support.

For debugging, there's also a `show()` method which saves an image to disk, and calls an external display utility.

3.1.3 Image Processing

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

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

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

3.2 Tutorial

3.2.1 Using the Image class

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

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

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

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

```
>>> from __future__ import print_function
>>> print(im.format, im.size, im.mode)
PPM (512, 512) RGB
```

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

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

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

3.2.2 Reading and writing images

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

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

Convert files to JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
```

```
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

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

Create JPEG thumbnails

```
from __future__ import print_function
import os, sys
from PIL import Image

size = (128, 128)

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

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

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

Identify Image Files

```
from __future__ import print_function
import sys
from PIL import Image

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

3.2.3 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.

Copying a subrectangle from an image

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

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

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

Processing a subrectangle, and pasting it back

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

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

Here's an additional example:

Rolling an image

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

    xsize, ysize = image.size

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

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

    return image
```

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

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

Splitting and merging bands

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

Note that for a single-band image, *split()* returns the image itself. To work with individual color bands, you may want to convert the image to "RGB" first.

3.2.4 Geometrical transforms

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

Simple geometry transforms

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

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

Transposing an image

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

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

A more general form of image transformations can be carried out via the *transform()* method.

3.2.5 Color transforms

The Python Imaging Library allows you to convert images between different pixel representations using the *convert()* method.

Converting between modes

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

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

3.2.6 Image enhancement

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

Filters

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

Applying filters

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

Point Operations

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

Applying point transforms

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

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

Processing individual bands

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

R, G, B = 0, 1, 2

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

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

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

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

Note the syntax used to create the mask:

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

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

Enhancement

For more advanced image enhancement, you can 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, color balance and sharpness in this way.

Enhancing images

```
from PIL import ImageEnhance

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

3.2.7 Image sequences

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

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

Reading sequences

```
from PIL import Image

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

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

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

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

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

A sequence iterator class

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

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

3.2.8 Postscript printing

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

Drawing Postscript

```
from PIL import Image
from PIL 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 title
ps.setfont("HelveticaNarrow-Bold", 36)
ps.text((3*72, 4*72), title)

ps.end_document()
```

3.2.9 More on reading images

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

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

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

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

Reading from an open file

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

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

Reading from a string

```
import StringIO

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

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

Reading from a tar archive

```
from PIL import TarIO

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

3.2.10 Controlling the decoder

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

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

Reading in draft mode

```
from __future__ import print_function
im = Image.open(file)
print("original =", im.mode, im.size)

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

This prints something like:

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

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

3.3 Concepts

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

3.3.1 Bands

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

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

3.3.2 Modes

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

- `1` (1-bit pixels, black and white, stored with one pixel per byte)
- `L` (8-bit pixels, black and white)
- `P` (8-bit pixels, mapped to any other mode using a color palette)
- `RGB` (3x8-bit pixels, true color)
- `RGBA` (4x8-bit pixels, true color with transparency mask)
- `CMYK` (4x8-bit pixels, color separation)
- `YCbCr` (3x8-bit pixels, color video format)
- `LAB` (3x8-bit pixels, the L*a*b color space)
- `HSV` (3x8-bit pixels, Hue, Saturation, Value color space)
- `I` (32-bit signed integer pixels)
- `F` (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including `LA` (L with alpha), `RGBX` (true color with padding) and `RGBa` (true color with premultiplied alpha). However, PIL doesn't support user-defined modes; if you to handle band combinations that are not listed above, use a sequence of Image objects.

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

3.3.3 Size

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

3.3.4 Coordinate System

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

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

3.3.5 Palette

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

3.3.6 Info

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

How such information is handled when loading and saving image files is up to the file format handler (see the chapter on *Image file formats*). Most handlers add properties to the `info` attribute when loading an image, but ignore it when saving images.

3.3.7 Filters

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

NEAREST Pick the nearest pixel from the input image. Ignore all other input pixels.

BILINEAR For resize calculate the output pixel value using linear interpolation on all pixels that may contribute to the output value. For other transformations linear interpolation over a 2x2 environment in the input image is used.

BICUBIC For resize calculate the output pixel value using cubic interpolation on all pixels that may contribute to the output value. For other transformations cubic interpolation over a 4x4 environment in the input image is used.

LANCZOS Calculate the output pixel value using a high-quality Lanczos filter (a truncated sinc) on all pixels that may contribute to the output value. In the current version of PIL, this filter can only be used with the resize and thumbnail methods.

New in version 1.1.3.

3.4 Porting existing PIL-based code to Pillow

Pillow is a functional drop-in replacement for the Python Imaging Library. To run your existing PIL-compatible code with Pillow, it needs to be modified to import the `Image` module from the `PIL` namespace *instead* of the global namespace. Change this:

```
import Image
```

to this:

```
from PIL import Image
```

The `_imaging` module has been moved. You can now import it like this:

```
from PIL.Image import core as _imaging
```

The image plugin loading mechanism has changed. Pillow no longer automatically imports any file in the Python path with a name ending in `ImagePlugin.py`. You will need to import your image plugin manually.

Pillow will raise an exception if the core extension can't be loaded for any reason, including a version mismatch between the Python and extension code. Previously PIL allowed Python only code to run if the core extension was not available.

3.5 Developer

Note: When committing only trivial changes, please include [ci skip] in the commit message to avoid running tests on Travis-CI. Thank you!

3.5.1 Release

Details about making a Pillow release.

Version number

The version number is currently stored in 3 places:

```
PIL/__init__.py _imaging.c setup.py
```

CHAPTER 4

Reference

4.1 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.

4.1.1 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)
    im.save(file + ".thumbnail", "JPEG")
```

4.1.2 Functions

PIL.Image.**open**(*fp*, *mode='r'*)
Opens and identifies the given image file.

This is a lazy operation; this function identifies the file, but the file remains open and the actual image data is not read from the file until you try to process the data (or call the *load()* method). See *new()*.

Parameters

- **file** – A filename (string) or a file object. The file object must implement `read()`, `seek()`, and `tell()` methods, and be opened in binary mode.
- **mode** – The mode. If given, this argument must be "r".

Returns An *Image* object.

Raises IOError If the file cannot be found, or the image cannot be opened and identified.

Warning: To protect against potential DOS attacks caused by "decompression bombs" (i.e. malicious files which decompress into a huge amount of data and are designed to crash or cause disruption by using up a lot of memory), Pillow will issue a *DecompressionBombWarning* if the image is over a certain limit. If desired, the warning can be turned into an error with *warnings.simplefilter('error', Image.DecompressionBombWarning)* or suppressed entirely with *warnings.simplefilter('ignore', Image.DecompressionBombWarning)*. See also the logging documentation to have warnings output to the logging facility instead of stderr.

Image processing

PIL.Image.**alpha_composite**(*im1*, *im2*)
Alpha composite im2 over im1.

Parameters

- **im1** – The first image.
- **im2** – The second image. Must have the same mode and size as the first image.

Returns An *Image* object.

PIL.Image.**blend**(*im1*, *im2*, *alpha*)
Creates a new image by interpolating between two input images, using a constant alpha.:

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

Parameters

- **im1** – The first image.
- **im2** – The second image. Must have the same mode and size as the first image.
- **alpha** – The interpolation alpha factor. 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.

Returns An *Image* object.

PIL.Image.**composite**(*image1*, *image2*, *mask*)
Create composite image by blending images using a transparency mask.

Parameters

- **image1** – The first image.
- **image2** – The second image. Must have the same mode and size as the first image.

- **mask** – A mask image. This image can have mode "1", "L", or "RGBA", and must have the same size as the other two images.

PIL.Image.**eval**(*image*, **args*)
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.

Parameters

- **image** – The input image.
- **function** – A function object, taking one integer argument.

Returns An *Image* object.

PIL.Image.**merge**(*mode*, *bands*)
Merge a set of single band images into a new multiband image.

Parameters

- **mode** – The mode to use for the output image. See: *Modes*.
- **bands** – A sequence containing one single-band image for each band in the output image. All bands must have the same size.

Returns An *Image* object.

Constructing images

PIL.Image.**new**(*mode*, *size*, *color=0*)
Creates a new image with the given mode and size.

Parameters

- **mode** – The mode to use for the new image. See: *Modes*.
- **size** – A 2-tuple, containing (width, height) in pixels.
- **color** – What color to use for the image. Default is black. If given, this should be a single integer or floating point value for single-band modes, and a tuple for multi-band modes (one value per band). When creating RGB images, you can also use color strings as supported by the ImageColor module. If the color is None, the image is not initialised.

Returns An *Image* object.

PIL.Image.**fromarray**(*obj*, *mode=None*)
Creates an image memory from an object exporting the array interface (using the buffer protocol).

If obj is not contiguous, then the tobytes method is called and *frombuffer()* is used.

Parameters

- **obj** – Object with array interface
- **mode** – Mode to use (will be determined from type if None) See: *Modes*.

Returns An image object.

New in version 1.1.6.

PIL.Image.**frombytes**(*mode*, *size*, *data*, *decoder_name='raw'*, **args*)
Creates a copy of an image memory from pixel data in a buffer.

In its simplest form, this function takes three arguments (mode, size, and unpacked pixel data).

You can also 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 in a string, wrap it in a `BytesIO` object, and use *`open()`* to load it.

Parameters

- **mode** – The image mode. See: *Modes*.
- **size** – The image size.
- **data** – A byte buffer containing raw data for the given mode.
- **decoder_name** – What decoder to use.
- **args** – Additional parameters for the given decoder.

Returns An *`Image`* object.

`PIL.Image.`**`fromstring`**(**args*, ***kw*)
Deprecated alias to frombytes.

Deprecated since version 2.0.

`PIL.Image.`**`frombuffer`**(*mode*, *size*, *data*, *decoder_name='raw'*, **args*)
Creates an image memory referencing pixel data in a byte buffer.

This function is similar to *`frombytes()`*, but uses data in the byte buffer, where possible. This means that changes to the original buffer object are reflected in this image). Not all modes can share memory; supported modes include "L", "RGBX", "RGBA", and "CMYK".

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 **BytesIO** object, and use *`open()`* to load it.

In the current version, the default parameters used for the "raw" decoder differs from that used for *`fromstring()`*. This is a bug, and will probably be fixed in a future release. The current release issues a warning if you do this; to disable the warning, you should provide the full set of parameters. See below for details.

Parameters

- **mode** – The image mode. See: *Modes*.
- **size** – The image size.
- **data** – A bytes or other buffer object containing raw data for the given mode.
- **decoder_name** – What decoder to use.
- **args** – Additional parameters for the given decoder. For the default encoder ("raw"), it's recommended that you provide the full set of parameters:

```
frombuffer(mode, size, data, "raw", mode, 0, 1)
```

Returns An *`Image`* object.

New in version 1.1.4.

Registering plugins

Note: These functions are for use by plugin authors. Application authors can ignore them.

PIL.Image.**register_open**(*id*, *factory*, *accept=None*)
Register an image file plugin. This function should not be used in application code.

Parameters

- **id** – An image format identifier.
- **factory** – An image file factory method.
- **accept** – An optional function that can be used to quickly reject images having another format.

PIL.Image.**register_mime**(*id*, *mimetype*)
Registers an image MIME type. This function should not be used in application code.

Parameters

- **id** – An image format identifier.
- **mimetype** – The image MIME type for this format.

PIL.Image.**register_save**(*id*, *driver*)
Registers an image save function. This function should not be used in application code.

Parameters

- **id** – An image format identifier.
- **driver** – A function to save images in this format.

PIL.Image.**register_extension**(*id*, *extension*)
Registers an image extension. This function should not be used in application code.

Parameters

- **id** – An image format identifier.
- **extension** – An extension used for this format.

4.1.3 The Image Class

class PIL.Image.**Image**
This class represents an image object. To create *Image* objects, use the appropriate factory functions. There's hardly ever any reason to call the Image constructor directly.

- *open()*
- *new()*
- *frombytes()*

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.

Image.**convert**(*mode=None*, *matrix=None*, *dither=None*, *palette=0*, *colors=256*)
Returns a converted copy of this image. For the "P" mode, this method translates pixels through the palette. If mode is omitted, a mode is chosen so that all information in the image and the palette can be represented without a palette.

The current version supports all possible conversions between "L", "RGB" and "CMYK." The **matrix** argument only supports "L" and "RGB".

When translating a color 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
```

The default method of converting a greyscale ("L") or "RGB" image into a bilevel (mode "1") image uses Floyd-Steinberg dither to approximate the original image luminosity levels. If dither is NONE, all non-zero values are set to 255 (white). To use other thresholds, use the *point()* method.

Parameters

- **mode** – The requested mode. See: *Modes*.
- **matrix** – An optional conversion matrix. If given, this should be 4- or 16-tuple containing floating point values.
- **dither** – Dithering method, used when converting from mode "RGB" to "P" or from "RGB" or "L" to "1". Available methods are NONE or FLOYDSTEINBERG (default).
- **palette** – Palette to use when converting from mode "RGB" to "P". Available palettes are WEB or ADAPTIVE.
- **colors** – Number of colors to use for the ADAPTIVE palette. Defaults to 256.

Return type *Image*

Returns An *Image* object.

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

```
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)
```

Image.**copy**()

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

Return type *Image*

Returns An *Image* object.

Image.**crop**(*box=None*)

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

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

Parameters **box** – The crop rectangle, as a (left, upper, right, lower)-tuple.

Return type *Image*

Returns An *Image* object.

Image.**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 color 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.

Parameters

- **mode** – The requested mode.

- **size** – The requested size.

Image.**filter**(*filter*)
Filters this image using the given filter. For a list of available filters, see the *ImageFilter* module.

Parameters **filter** – Filter kernel.

Returns An *Image* object.

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

Returns A tuple containing band names.

Return type tuple

Image.**getbbox**()
Calculates the bounding box of the non-zero regions in the image.

Returns 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.

Image.**getcolors**(*maxcolors=256*)
Returns a list of colors used in this image.

Parameters **maxcolors** – Maximum number of colors. If this number is exceeded, this method returns None. The default limit is 256 colors.

Returns An unsorted list of (count, pixel) values.

Image.**getdata**(*band=None*)
Returns the contents of this image as a sequence object containing pixel values. The sequence object is flattened, so that values for line one follow directly after the values of line zero, and so on.

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

Parameters **band** – What band to return. The default is to return all bands. To return a single band, pass in the index value (e.g. 0 to get the "R" band from an "RGB" image).

Returns A sequence-like object.

Image.**getextrema**()
Gets the the minimum and maximum pixel values for each band in the image.

Returns For a single-band image, a 2-tuple containing the minimum and maximum pixel value. For a multi-band image, a tuple containing one 2-tuple for each band.

Image.**getpalette**()
Returns the image palette as a list.

Returns A list of color values [r, g, b, ...], or None if the image has no palette.

Image.**getpixel**(*xy*)
Returns the pixel value at a given position.

Parameters **xy** – The coordinate, given as (x, y).

Returns The pixel value. If the image is a multi-layer image, this method returns a tuple.

Image.**histogram**(*mask=None*, *extrema=None*)
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.

If a mask is provided, the method 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").

Parameters **mask** – An optional mask.

Returns A list containing pixel counts.

Image.**offset**(*xoffset*, *yoffset=None*)

Deprecated since version 2.0.

Note: New code should use *PIL.ImageChops.offset()*.

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**.

Parameters

- **xoffset** – The horizontal distance.
- **yoffset** – The vertical distance. If omitted, both distances are set to the same value.

Returns An *Image* object.

Image.**paste**(*im*, *box=None*, *mask=None*)

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).

Instead of an image, the source can be a integer or tuple containing pixel values. The method then fills the region with the given color. When creating RGB images, you can also use color strings as supported by the ImageColor module.

If a mask is given, this method 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.

Parameters

- **im** – Source image or pixel value (integer or tuple).
- **box** – An optional 4-tuple giving the region to paste into. If a 2-tuple is used instead, it's treated as the upper left corner. If omitted or None, the source is pasted into the upper left corner.

  If an image is given as the second argument and there is no third, the box defaults to (0, 0), and the second argument is interpreted as a mask image.
- **mask** – An optional mask image.

Image.**point**(*lut*, *mode=None*)

Maps this image through a lookup table or function.

Parameters

- **lut** – A lookup table, containing 256 (or 65336 if self.mode==”I” and mode == “L”) values per band in the image. A function can be 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.
- **mode** – Output mode (default is same as input). In the current version, this can only be used if the source image has mode “L” or “P”, and the output has mode “1” or the source image mode is “I” and the output mode is “L”.

Returns An *Image* object.

Image.**putalpha**(*alpha*)

Adds or replaces the alpha layer in this image. If the image does not have an alpha layer, it’s converted to “LA” or “RGBA”. The new layer must be either “L” or “1”.

Parameters **alpha** – The new alpha layer. This can either be an “L” or “1” image having the same size as this image, or an integer or other color value.

Image.**putdata**(*data*, *scale=1.0*, *offset=0.0*)

Copies pixel data to this image. This method copies data from a sequence object into the image, starting at the upper left corner (0, 0), and continuing until either the image or the sequence ends. The scale and offset values are used to adjust the sequence values: **pixel = value*scale + offset**.

Parameters

- **data** – A sequence object.
- **scale** – An optional scale value. The default is 1.0.
- **offset** – An optional offset value. The default is 0.0.

Image.**putpalette**(*data*, *rawmode='RGB'*)

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

Parameters **data** – A palette sequence (either a list or a string).

Image.**putpixel**(*xy*, *value*)

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

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

See:

- *paste()*
- *putdata()*
- *ImageDraw*

Parameters

- **xy** – The pixel coordinate, given as (x, y).
- **value** – The pixel value.

Image.**quantize**(*colors=256*, *method=None*, *kmeans=0*, *palette=None*)

Convert the image to ‘P’ mode with the specified number of colors.

Parameters

- **colors** – The desired number of colors, <= 256
- **method** – 0 = median cut 1 = maximum coverage 2 = fast octree
- **kmeans** – Integer
- **palette** – Quantize to the `PIL.ImagingPalette` palette.

Returns A new image

`Image.`**`resize`**(*size*, *resample=0*)

Returns a resized copy of this image.

Parameters

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **resample** – An optional resampling filter. This can be one of `PIL.Image.NEAREST` (use nearest neighbour), `PIL.Image.BILINEAR` (linear interpolation), `PIL.Image.BICUBIC` (cubic spline interpolation), or `PIL.Image.LANCZOS` (a high-quality downsampling filter). If omitted, or if the image has mode "1" or "P", it is set `PIL.Image.NEAREST`.

Returns An *Image* object.

`Image.`**`rotate`**(*angle*, *resample=0*, *expand=0*)

Returns a rotated copy of this image. This method returns a copy of this image, rotated the given number of degrees counter clockwise around its centre.

Parameters

- **angle** – In degrees counter clockwise.
- **resample** – An optional resampling filter. This can be one of `PIL.Image.NEAREST` (use nearest neighbour), `PIL.Image.BILINEAR` (linear interpolation in a 2x2 environment), or `PIL.Image.BICUBIC` (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode "1" or "P", it is set `PIL.Image.NEAREST`.
- **expand** – Optional expansion flag. If true, expands the output image to make it large enough to hold the entire rotated image. If false or omitted, make the output image the same size as the input image.

Returns An *Image* object.

`Image.`**`save`**(*fp*, *format=None*, ***params*)

Saves this image under the given filename. If no format is specified, the format to use is determined from the filename extension, if possible.

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 in the image format documentation for each writer.

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.

Parameters

- **fp** – File name or file object.
- **format** – Optional format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter should always be used.
- **options** – Extra parameters to the image writer.

Returns None

Raises

- **KeyError** – If the output format could not be determined from the file name. Use the format option to solve this.
- **IOError** – If the file could not be written. The file may have been created, and may contain partial data.

Image.**seek**(*frame*)

Seeks to the given frame in this 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.

See *tell()*.

Parameters **frame** – Frame number, starting at 0.

Raises EOFError If the call attempts to seek beyond the end of the sequence.

Image.**show**(*title=None*, *command=None*)

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

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

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

Parameters

- **title** – Optional title to use for the image window, where possible.
- **command** – command used to show the image

Image.**split**()

Split this image into individual bands. This method 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).

Returns A tuple containing bands.

Image.**tell**()

Returns the current frame number. See *seek()*.

Returns Frame number, starting with 0.

Image.**thumbnail**(*size*, *resample=3*)

Make this image into a thumbnail. This method 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.

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.

Parameters

- **size** – Requested size.
- **resample** – Optional resampling filter. This can be one of PIL.Image.NEAREST, PIL.Image.BILINEAR, PIL.Image.BICUBIC, or PIL.Image.LANCZOS. If omitted, it defaults to PIL.Image.BICUBIC. (was PIL.Image.NEAREST prior to version 2.5.0)

Returns None

Image.**tobitmap**(*name='image'*)
Returns the image converted to an X11 bitmap.

Note: This method only works for mode "1" images.

Parameters **name** – The name prefix to use for the bitmap variables.

Returns A string containing an X11 bitmap.

Raises ValueError If the mode is not "1"

Image.**tobytes**(*encoder_name='raw'*, **args*)
Return image as a bytes object

Parameters

- **encoder_name** – What encoder to use. The default is to use the standard "raw" encoder.
- **args** – Extra arguments to the encoder.

Return type A bytes object.

Image.**tostring**(**args*, ***kw*)
Deprecated alias to tobytes.

Deprecated since version 2.0.

Image.**transform**(*size*, *method*, *data=None*, *resample=0*, *fill=1*)
Transforms this image. This method 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.

Parameters

- **size** – The output size.
- **method** – The transformation method. This is one of PIL.Image.EXTENT (cut out a rectangular subregion), PIL.Image.AFFINE (affine transform), PIL.Image.PERSPECTIVE (perspective transform), PIL.Image.QUAD (map a quadrilateral to a rectangle), or PIL.Image.MESH (map a number of source quadrilaterals in one operation).
- **data** – Extra data to the transformation method.
- **resample** – Optional resampling filter. It can be one of PIL.Image.NEAREST (use nearest neighbour), PIL.Image.BILINEAR (linear interpolation in a 2x2 environment), or PIL.Image.BICUBIC (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode "1" or "P", it is set to PIL.Image.NEAREST.

Returns An *Image* object.

Image.**transpose**(*method*)
Transpose image (flip or rotate in 90 degree steps)

Parameters **method** – One of PIL.Image.FLIP_LEFT_RIGHT, PIL.Image.FLIP_TOP_BOTTOM, PIL.Image.ROTATE_90, PIL.Image.ROTATE_180, PIL.Image.ROTATE_270 or PIL.Image.TRANSPOSE.

Returns Returns a flipped or rotated copy of this image.

Image.**verify**()
Verifies the contents of a file. For data read from a file, this method 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.

Image.**fromstring**(**args*, ***kw*)
Deprecated alias to frombytes.

Deprecated since version 2.0.

Image.**load**()
Allocates storage for the image and loads the pixel data. 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. This method will close the file associated with the image.

Returns An image access object.

Return type *PixelAccess Class* or *PIL.PyAccess*

Image.**close**()
Closes the file pointer, if possible.

This operation will destroy the image core and release its memory. The image data will be unusable afterward.

This function is only required to close images that have not had their file read and closed by the *load()* method.

4.1.4 Attributes

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

PIL.Image.**format**
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.

Type string or None

PIL.Image.**mode**
Image mode. This is a string specifying the pixel format used by the image. Typical values are "1", "L", "RGB", or "CMYK." See *Modes* for a full list.

Type string

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

Type (width, height)

PIL.Image.**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.

Type *ImagePalette* or None

PIL.Image.**info**
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.

Unless noted elsewhere, this dictionary does not affect saving files.

Type dict

4.2 ImageChops ("Channel Operations") Module

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

For more pre-made operations, see ImageOps.

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

4.2.1 Functions

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

PIL.ImageChops.**add**(*image1*, *image2*, *scale=1.0*, *offset=0*)
Adds two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

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

Return type *Image*

PIL.ImageChops.**add_modulo**(*image1*, *image2*)
Add two images, without clipping the result.

```
out = ((image1 + image2) % MAX)
```

Return type *Image*

PIL.ImageChops.**blend**(*image1*, *image2*, *alpha*)
Blend images using constant transparency weight. Alias for PIL.Image.Image.blend().

Return type *Image*

PIL.ImageChops.**composite**(*image1*, *image2*, *mask*)
Create composite using transparency mask. Alias for PIL.Image.Image.composite().

Return type *Image*

PIL.ImageChops.**constant**(*image*, *value*)
Fill a channel with a given grey level.

Return type *Image*

PIL.ImageChops.**darker**(*image1*, *image2*)
Compares the two images, pixel by pixel, and returns a new image containing the darker values.

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

Return type *Image*

PIL.ImageChops.**difference**(*image1*, *image2*)
Returns the absolute value of the pixel-by-pixel difference between the two images.

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

Return type *Image*

PIL.ImageChops.**duplicate**(*image*)
Copy a channel. Alias for *PIL.Image.Image.copy()*.

Return type *Image*

PIL.ImageChops.**invert**(*image*)
Invert an image (channel).

```
out = MAX - image
```

Return type *Image*

PIL.ImageChops.**lighter**(*image1*, *image2*)
Compares the two images, pixel by pixel, and returns a new image containing the lighter values.

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

Return type *Image*

PIL.ImageChops.**logical_and**(*image1*, *image2*)
Logical AND between two images.

```
out = ((image1 and image2) % MAX)
```

Return type *Image*

PIL.ImageChops.**logical_or**(*image1*, *image2*)
Logical OR between two images.

```
out = ((image1 or image2) % MAX)
```

Return type *Image*

PIL.ImageChops.**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
```

Return type *Image*

PIL.ImageChops.**offset**(*image*, *xoffset*, *yoffset=None*)
Returns a copy of the image where data has been offset by the given distances. Data wraps around the edges. If **yoffset** is omitted, it is assumed to be equal to **xoffset**.

Parameters

- **xoffset** – The horizontal distance.
- **yoffset** – The vertical distance. If omitted, both distances are set to the same value.

Return type *Image*

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

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

Return type *Image*

PIL.ImageChops.**subtract**(*image1*, *image2*, *scale=1.0*, *offset=0*)
Subtracts two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

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

Return type *Image*

PIL.ImageChops.**subtract_modulo**(*image1*, *image2*)
Subtract two images, without clipping the result.

```
out = ((image1 - image2) % MAX)
```

Return type *Image*

4.3 ImageColor Module

The ImageColor module contains color tables and converters from CSS3-style color specifiers to RGB tuples. This module is used by PIL.Image.Image.new() and the *ImageDraw* module, among others.

4.3.1 Color Names

The ImageColor module supports the following string formats:

- Hexadecimal color specifiers, given as #rgb or #rrggbb. For example, #ff0000 specifies pure red.
- RGB functions, given as rgb(red, green, blue) where the color values are integers in the range 0 to 255. Alternatively, the color values can be given as three percentages (0% to 100%). For example, rgb(255,0,0) and rgb(100%,0%,0%) both specify pure red.
- Hue-Saturation-Lightness (HSL) functions, given as hsl(hue, saturation%, lightness%) where hue is the color given as an angle between 0 and 360 (red=0, green=120, blue=240), saturation is a value between 0% and 100% (gray=0%, full color=100%), and lightness is a value between 0% and 100% (black=0%, normal=50%, white=100%). For example, hsl(0,100%,50%) is pure red.
- Common HTML color names. The *ImageColor* module provides some 140 standard color names, based on the colors supported by the X Window system and most web browsers. color names are case insensitive. For example, red and Red both specify pure red.

4.3.2 Functions

PIL.ImageColor.**getrgb**(*color*)

Convert a color string to an RGB tuple. If the string cannot be parsed, this function raises a ValueError exception.

New in version 1.1.4.

Parameters **color** – A color string

Returns (red, green, blue[, alpha])

PIL.ImageColor.**getcolor**(*color*, *mode*)
Same as *getrgb()*, but converts the RGB value to a greyscale value if the mode is not color or a palette image. If the string cannot be parsed, this function raises a ValueError exception.

New in version 1.1.4.

Parameters **color** – A color string

Returns (graylevel [, alpha]) or (red, green, blue[, alpha])

4.4 ImageCms Module

The ImageCms module provides color profile management support using the LittleCMS2 color management engine, based on Kevin Cazabon's PyCMS library.

exception PIL.ImageCms.**PyCMSError**
(pyCMS) Exception class. This is used for all errors in the pyCMS API.

PIL.ImageCms.**applyTransform**(*im*, *transform*, *inPlace=0*)
(pyCMS) Applies a transform to a given image.

If im.mode != transform.inMode, a PyCMSError is raised.

If inPlace == TRUE and transform.inMode != transform.outMode, a PyCMSError is raised.

If im.mode, transfer.inMode, or transfer.outMode is not supported by pyCMSdll or the profiles you used for the transform, a PyCMSError is raised.

If an error occurs while the transform is being applied, a PyCMSError is raised.

This function applies a pre-calculated transform (from ImageCms.buildTransform() or ImageCms.buildTransformFromOpenProfiles()) to an image. The transform can be used for multiple images, saving considerable calcuation time if doing the same conversion multiple times.

If you want to modify im in-place instead of receiving a new image as the return value, set inPlace to TRUE. This can only be done if transform.inMode and transform.outMode are the same, because we can't change the mode in-place (the buffer sizes for some modes are different). The default behavior is to return a new Image object of the same dimensions in mode transform.outMode.

Parameters

- **im** – A PIL Image object, and im.mode must be the same as the inMode supported by the transform.
- **transform** – A valid CmsTransform class object
- **inPlace** – Bool (1 == True, 0 or None == False). If True, im is modified in place and None is returned, if False, a new Image object with the transform applied is returned (and im is not changed). The default is False.

Returns Either None, or a new PIL Image object, depending on the value of inPlace. The profile will be returned in the image's info['icc_profile'].

Raises PyCMSError

PIL.ImageCms.**buildProofTransform**(*inputProfile*, *outputProfile*, *proofProfile*, *inMode*, *outMode*, *renderingIntent=0*, *proofRenderingIntent=3*, *flags=16384*)

(pyCMS) Builds an ICC transform mapping from the inputProfile to the outputProfile, but tries to simulate the result that would be obtained on the proofProfile device.

If the input, output, or proof profiles specified are not valid filenames, a PyCMSError will be raised.

If an error occurs during creation of the transform, a PyCMSError will be raised.

If inMode or outMode are not a mode supported by the outputProfile (or by pyCMS), a PyCMSError will be raised.

This function builds and returns an ICC transform from the inputProfile to the outputProfile, but tries to simulate the result that would be obtained on the proofProfile device using renderingIntent and proofRenderingIntent to determine what to do with out-of-gamut colors. This is known as "soft-proofing". It will ONLY work for converting images that are in inMode to images that are in outMode color format (PIL mode, i.e. "RGB", "RGBA", "CMYK", etc.).

Usage of the resulting transform object is exactly the same as with ImageCms.buildTransform().

Proof profiling is generally used when using an output device to get a good idea of what the final printed/displayed image would look like on the proofProfile device when it's quicker and easier to use the output device for judging color. Generally, this means that the output device is a monitor, or a dye-sub printer (etc.), and the simulated device is something more expensive, complicated, or time consuming (making it difficult to make a real print for color judgement purposes).

Soft-proofing basically functions by adjusting the colors on the output device to match the colors of the device being simulated. However, when the simulated device has a much wider gamut than the output device, you may obtain marginal results.

Parameters

- **inputProfile** – String, as a valid filename path to the ICC input profile you wish to use for this transform, or a profile object
- **outputProfile** – String, as a valid filename path to the ICC output (monitor, usually) profile you wish to use for this transform, or a profile object
- **proofProfile** – String, as a valid filename path to the ICC proof profile you wish to use for this transform, or a profile object
- **inMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **outMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **renderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for the input->proof (simulated) transform

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.
- **proofRenderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for proof->output transform

INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

see the pyCMS documentation for details on rendering intents and what they do.

- **flags** – Integer (0-...) specifying additional flags

Returns A CmsTransform class object.

Raises PyCMSError

PIL.ImageCms.**buildProofTransformFromOpenProfiles**(*inputProfile*, *outputProfile*, *proofProfile*, *inMode*, *outMode*, *renderingIntent=0*, *proofRenderingIntent=3*, *flags=16384*)

(pyCMS) Builds an ICC transform mapping from the inputProfile to the outputProfile, but tries to simulate the result that would be obtained on the proofProfile device.

If the input, output, or proof profiles specified are not valid filenames, a PyCMSError will be raised.

If an error occurs during creation of the transform, a PyCMSError will be raised.

If inMode or outMode are not a mode supported by the outputProfile (or by pyCMS), a PyCMSError will be raised.

This function builds and returns an ICC transform from the inputProfile to the outputProfile, but tries to simulate the result that would be obtained on the proofProfile device using renderingIntent and proofRenderingIntent to determine what to do with out-of-gamut colors. This is known as "soft-proofing". It will ONLY work for converting images that are in inMode to images that are in outMode color format (PIL mode, i.e. "RGB", "RGBA", "CMYK", etc.).

Usage of the resulting transform object is exactly the same as with ImageCms.buildTransform().

Proof profiling is generally used when using an output device to get a good idea of what the final printed/displayed image would look like on the proofProfile device when it's quicker and easier to use the output device for judging color. Generally, this means that the output device is a monitor, or a dye-sub printer (etc.), and the simulated device is something more expensive, complicated, or time consuming (making it difficult to make a real print for color judgement purposes).

Soft-proofing basically functions by adjusting the colors on the output device to match the colors of the device being simulated. However, when the simulated device has a much wider gamut than the output device, you may obtain marginal results.

Parameters

- **inputProfile** – String, as a valid filename path to the ICC input profile you wish to use for this transform, or a profile object
- **outputProfile** – String, as a valid filename path to the ICC output (monitor, usually) profile you wish to use for this transform, or a profile object
- **proofProfile** – String, as a valid filename path to the ICC proof profile you wish to use for this transform, or a profile object
- **inMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **outMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)

- **renderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for the input->proof (simulated) transform

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.

- **proofRenderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for proof->output transform

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.

- **flags** – Integer (0-...) specifying additional flags

Returns A CmsTransform class object.

Raises PyCMSError

PIL.ImageCms.**buildTransform**(*inputProfile*, *outputProfile*, *inMode*, *outMode*, *renderingIntent=0*, *flags=0*)

(pyCMS) Builds an ICC transform mapping from the inputProfile to the outputProfile. Use applyTransform to apply the transform to a given image.

If the input or output profiles specified are not valid filenames, a PyCMSError will be raised. If an error occurs during creation of the transform, a PyCMSError will be raised.

If inMode or outMode are not a mode supported by the outputProfile (or by pyCMS), a PyCMSError will be raised.

This function builds and returns an ICC transform from the inputProfile to the outputProfile using the renderingIntent to determine what to do with out-of-gamut colors. It will ONLY work for converting images that are in inMode to images that are in outMode color format (PIL mode, i.e. "RGB", "RGBA", "CMYK", etc.).

Building the transform is a fair part of the overhead in ImageCms.profileToProfile(), so if you're planning on converting multiple images using the same input/output settings, this can save you time. Once you have a transform object, it can be used with ImageCms.applyProfile() to convert images without the need to re-compute the lookup table for the transform.

The reason pyCMS returns a class object rather than a handle directly to the transform is that it needs to keep track of the PIL input/output modes that the transform is meant for. These attributes are stored in the "inMode" and "outMode" attributes of the object (which can be manually overridden if you really want to, but I don't know of any time that would be of use, or would even work).

Parameters

- **inputProfile** – String, as a valid filename path to the ICC input profile you wish to use for this transform, or a profile object
- **outputProfile** – String, as a valid filename path to the ICC output profile you wish to use for this transform, or a profile object
- **inMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)

- **outMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **renderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for the transform

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.
- **flags** – Integer (0-...) specifying additional flags

Returns A CmsTransform class object.

Raises PyCMSError

PIL.ImageCms.**buildTransformFromOpenProfiles**(*inputProfile*, *outputProfile*, *inMode*, *outMode*, *renderingIntent=0*, *flags=0*)

(pyCMS) Builds an ICC transform mapping from the inputProfile to the outputProfile. Use applyTransform to apply the transform to a given image.

If the input or output profiles specified are not valid filenames, a PyCMSError will be raised. If an error occurs during creation of the transform, a PyCMSError will be raised.

If inMode or outMode are not a mode supported by the outputProfile (or by pyCMS), a PyCMSError will be raised.

This function builds and returns an ICC transform from the inputProfile to the outputProfile using the renderingIntent to determine what to do with out-of-gamut colors. It will ONLY work for converting images that are in inMode to images that are in outMode color format (PIL mode, i.e. "RGB", "RGBA", "CMYK", etc.).

Building the transform is a fair part of the overhead in ImageCms.profileToProfile(), so if you're planning on converting multiple images using the same input/output settings, this can save you time. Once you have a transform object, it can be used with ImageCms.applyProfile() to convert images without the need to re-compute the lookup table for the transform.

The reason pyCMS returns a class object rather than a handle directly to the transform is that it needs to keep track of the PIL input/output modes that the transform is meant for. These attributes are stored in the "inMode" and "outMode" attributes of the object (which can be manually overridden if you really want to, but I don't know of any time that would be of use, or would even work).

Parameters

- **inputProfile** – String, as a valid filename path to the ICC input profile you wish to use for this transform, or a profile object
- **outputProfile** – String, as a valid filename path to the ICC output profile you wish to use for this transform, or a profile object
- **inMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **outMode** – String, as a valid PIL mode that the appropriate profile also supports (i.e. "RGB", "RGBA", "CMYK", etc.)
- **renderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for the transform

INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

see the pyCMS documentation for details on rendering intents and what they do.

- **flags** – Integer (0-...) specifying additional flags

Returns A CmsTransform class object.

Raises PyCMSError

PIL.ImageCms.**createProfile**(*colorSpace*, *colorTemp=-1*)
(pyCMS) Creates a profile.

If colorSpace not in ["LAB", "XYZ", "sRGB"], a PyCMSError is raised

If using LAB and colorTemp != a positive integer, a PyCMSError is raised.

If an error occurs while creating the profile, a PyCMSError is raised.

Use this function to create common profiles on-the-fly instead of having to supply a profile on disk and knowing the path to it. It returns a normal CmsProfile object that can be passed to ImageCms.buildTransformFromOpenProfiles() to create a transform to apply to images.

Parameters

- **colorSpace** – String, the color space of the profile you wish to create. Currently only "LAB", "XYZ", and "sRGB" are supported.
- **colorTemp** – Positive integer for the white point for the profile, in degrees Kelvin (i.e. 5000, 6500, 9600, etc.). The default is for D50 illuminant if omitted (5000k). colorTemp is ONLY applied to LAB profiles, and is ignored for XYZ and sRGB.

Returns A CmsProfile class object

Raises PyCMSError

PIL.ImageCms.**getDefaultIntent**(*profile*)
(pyCMS) Gets the default intent name for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the default intent, a PyCMSError is raised.

Use this function to determine the default (and usually best optomized) rendering intent for this profile. Most profiles support multiple rendering intents, but are intended mostly for one type of conversion. If you wish to use a different intent than returned, use ImageCms.isIntentSupported() to verify it will work first.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns

Integer 0-3 specifying the default rendering intent for this profile.

INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

see the pyCMS documentation for details on rendering intents and what they do.

Raises PyCMSError

PIL.ImageCms.**getOpenProfile**(*profileFilename*)
(pyCMS) Opens an ICC profile file.

The PyCMSProfile object can be passed back into pyCMS for use in creating transforms and such (as in ImageCms.buildTransformFromOpenProfiles()).

If profileFilename is not a vaild filename for an ICC profile, a PyCMSError will be raised.

Parameters **profileFilename** – String, as a valid filename path to the ICC profile you wish to open, or a file-like object.

Returns A CmsProfile class object.

Raises PyCMSError

PIL.ImageCms.**getProfileCopyright**(*profile*)
(pyCMS) Gets the copyright for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the copyright tag, a PyCMSError is raised

Use this function to obtain the information stored in the profile's copyright tag.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal profile information stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**getProfileDescription**(*profile*)
(pyCMS) Gets the description for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the description tag, a PyCMSError is raised

Use this function to obtain the information stored in the profile's description tag.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal profile information stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**getProfileInfo**(*profile*)
(pyCMS) Gets the internal product information for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the info tag, a PyCMSError is raised

Use this function to obtain the information stored in the profile's info tag. This often contains details about the profile, and how it was created, as supplied by the creator.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal profile information stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**getProfileManufacturer**(*profile*)
(pyCMS) Gets the manufacturer for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the manufacturer tag, a PyCMSError is raised

Use this function to obtain the information stored in the profile's manufacturer tag.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal profile information stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**getProfileModel**(*profile*)
(pyCMS) Gets the model for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised.

If an error occurs while trying to obtain the model tag, a PyCMSError is raised

Use this function to obtain the information stored in the profile's model tag.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal profile information stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**getProfileName**(*profile*)
(pyCMS) Gets the internal product name for the given profile.

If profile isn't a valid CmsProfile object or filename to a profile, a PyCMSError is raised If an error occurs while trying to obtain the name tag, a PyCMSError is raised.

Use this function to obtain the INTERNAL name of the profile (stored in an ICC tag in the profile itself), usually the one used when the profile was originally created. Sometimes this tag also contains additional information supplied by the creator.

Parameters **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.

Returns A string containing the internal name of the profile as stored in an ICC tag.

Raises PyCMSError

PIL.ImageCms.**get_display_profile**(*handle=None*)
(experimental) Fetches the profile for the current display device. :returns: None if the profile is not known.

PIL.ImageCms.**isIntentSupported**(*profile*, *intent*, *direction*)
(pyCMS) Checks if a given intent is supported.

Use this function to verify that you can use your desired renderingIntent with profile, and that profile can be used for the input/output/proof profile as you desire.

Some profiles are created specifically for one "direction", can cannot be used for others. Some profiles can only be used for certain rendering intents... so it's best to either verify this before trying to create a transform with them (using this function), or catch the potential PyCMSError that will occur if they don't support the modes you select.

Parameters

- **profile** – EITHER a valid CmsProfile object, OR a string of the filename of an ICC profile.
- **intent** – Integer (0-3) specifying the rendering intent you wish to use with this profile

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.

- **direction** – Integer specifing if the profile is to be used for input, output, or proof

  INPUT = 0 (or use ImageCms.DIRECTION_INPUT) OUTPUT = 1 (or use ImageCms.DIRECTION_OUTPUT) PROOF = 2 (or use ImageCms.DIRECTION_PROOF)

Returns 1 if the intent/direction are supported, -1 if they are not.

Raises PyCMSError

PIL.ImageCms.**profileToProfile**(*im*, *inputProfile*, *outputProfile*, *renderingIntent=0*, *outputMode=None*, *inPlace=0*, *flags=0*)

(pyCMS) Applies an ICC transformation to a given image, mapping from inputProfile to outputProfile.

If the input or output profiles specified are not valid filenames, a PyCMSError will be raised. If inPlace == TRUE and outputMode != im.mode, a PyCMSError will be raised. If an error occurs during application of the profiles, a PyCMSError will be raised. If outputMode is not a mode supported by the outputProfile (or by pyCMS), a PyCMSError will be raised.

This function applies an ICC transformation to im from inputProfile's color space to outputProfile's color space using the specified rendering intent to decide how to handle out-of-gamut colors.

OutputMode can be used to specify that a color mode conversion is to be done using these profiles, but the specified profiles must be able to handle that mode. I.e., if converting im from RGB to CMYK using profiles, the input profile must handle RGB data, and the output profile must handle CMYK data.

Parameters

- **im** – An open PIL image object (i.e. Image.new(...) or Image.open(...), etc.)
- **inputProfile** – String, as a valid filename path to the ICC input profile you wish to use for this image, or a profile object
- **outputProfile** – String, as a valid filename path to the ICC output profile you wish to use for this image, or a profile object
- **renderingIntent** – Integer (0-3) specifying the rendering intent you wish to use for the transform

  INTENT_PERCEPTUAL = 0 (DEFAULT) (ImageCms.INTENT_PERCEPTUAL) INTENT_RELATIVE_COLORIMETRIC = 1 (ImageCms.INTENT_RELATIVE_COLORIMETRIC) INTENT_SATURATION = 2 (ImageCms.INTENT_SATURATION) INTENT_ABSOLUTE_COLORIMETRIC = 3 (ImageCms.INTENT_ABSOLUTE_COLORIMETRIC)

  see the pyCMS documentation for details on rendering intents and what they do.

- **outputMode** – A valid PIL mode for the output image (i.e. "RGB", "CMYK", etc.). Note: if rendering the image "inPlace", outputMode MUST be the same mode as the input,

or omitted completely. If omitted, the outputMode will be the same as the mode of the input image (im.mode)

- **inPlace** – Boolean (1 = True, None or 0 = False). If True, the original image is modified in-place, and None is returned. If False (default), a new Image object is returned with the transform applied.
- **flags** – Integer (0-...) specifying additional flags

Returns Either None or a new PIL image object, depending on value of inPlace

Raises PyCMSError

PIL.ImageCms.**versions**()
(pyCMS) Fetches versions.

4.5 ImageDraw Module

The ImageDraw module provide simple 2D graphics for *Image* objects. You can use this module to create new images, annotate or retouch existing images, and to generate graphics on the fly for web use.

For a more advanced drawing library for PIL, see the aggdraw module.

4.5.1 Example: Draw a gray cross over an image

```
from PIL import Image, ImageDraw

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

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

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

4.5.2 Concepts

Coordinates

The graphics interface uses the same coordinate system as PIL itself, with (0, 0) in the upper left corner.

Colors

To specify colors, you can use numbers or tuples just as you would use with PIL.Image.Image.new() or *PIL.Image.Image.putpixel()*. For “1”, “L”, and “I” images, use integers. For “RGB” images, use a 3-tuple containing integer values. For “F” images, use integer or floating point values.

For palette images (mode “P”), use integers as color indexes. In 1.1.4 and later, you can also use RGB 3-tuples or color names (see below). The drawing layer will automatically assign color indexes, as long as you don’t draw with more than 256 colors.

Color Names

See *Color Names* for the color names supported by Pillow.

Fonts

PIL can use bitmap fonts or OpenType/TrueType fonts.

Bitmap fonts are stored in PIL's own format, where each font typically consists of a two files, one named .pil and the other usually named .pbm. The former contains font metrics, the latter raster data.

To load a bitmap font, use the load functions in the *ImageFont* module.

To load a OpenType/TrueType font, use the truetype function in the *ImageFont* module. Note that this function depends on third-party libraries, and may not available in all PIL builds.

4.5.3 Example: Draw Partial Opacity Text

```
from PIL import Image, ImageDraw, ImageFont
# get an image
base = Image.open('Pillow/Tests/images/lena.png').convert('RGBA')

# make a blank image for the text, initialized to transparent text color
txt = Image.new('RGBA', base.size, (255,255,255,0))

# get a font
fnt = ImageFont.truetype('Pillow/Tests/fonts/FreeMono.ttf', 40)
# get a drawing context
d = ImageDraw.Draw(txt)

# draw text, half opacity
d.text((10,10), "Hello", font=fnt, fill=(255,255,255,128))
# draw text, full opacity
d.text((10,60), "World", font=fnt, fill=(255,255,255,255))

out = Image.alpha_composite(base, txt)

out.show()
```

4.5.4 Functions

class PIL.ImageDraw.**Draw**(*im*, *mode=None*)

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

Note that the image will be modified in place.

Parameters

- **im** – The image to draw in.
- **mode** – Optional mode to use for color values. For RGB images, this argument can be RGB or RGBA (to blend the drawing into the image). For all other modes, this argument must be the same as the image mode. If omitted, the mode defaults to the mode of the image.

4.5.5 Methods

PIL.ImageDraw.Draw.**arc**(*xy*, *start*, *end*, *fill=None*)
Draws an arc (a portion of a circle outline) between the start and end angles, inside the given bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **start** – Starting angle, in degrees. Angles are measured from 3 o'clock, increasing clockwise.
- **end** – Ending angle, in degrees.
- **fill** – Color to use for the arc.

PIL.ImageDraw.Draw.**bitmap**(*xy*, *bitmap*, *fill=None*)
Draws a bitmap (mask) at the given position, using the current fill color for the non-zero portions. The bitmap should be a valid transparency mask (mode "1") or matte (mode "L" or "RGBA").

This is equivalent to doing `image.paste(xy, color, bitmap)`.

To paste pixel data into an image, use the *paste()* method on the image itself.

PIL.ImageDraw.Draw.**chord**(*xy*, *start*, *end*, *fill=None*, *outline=None*)
Same as *arc()*, but connects the end points with a straight line.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

PIL.ImageDraw.Draw.**ellipse**(*xy*, *fill=None*, *outline=None*)
Draws an ellipse inside the given bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

PIL.ImageDraw.Draw.**line**(*xy*, *fill=None*, *width=0*)
Draws a line between the coordinates in the **xy** list.

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **fill** – Color to use for the line.
- **width** – The line width, in pixels. Note that line joins are not handled well, so wide polylines will not look good.

  New in version 1.1.5.

  Note: This option was broken until version 1.1.6.

PIL.ImageDraw.Draw.**pieslice**(*xy*, *start*, *end*, *fill=None*, *outline=None*)
Same as arc, but also draws straight lines between the end points and the center of the bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

PIL.ImageDraw.Draw.**point**(*xy*, *fill=None*)
Draws points (individual pixels) at the given coordinates.

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **fill** – Color to use for the point.

PIL.ImageDraw.Draw.**polygon**(*xy*, *fill=None*, *outline=None*)
Draws a polygon.

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

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

PIL.ImageDraw.Draw.**rectangle**(*xy*, *fill=None*, *outline=None*)
Draws a rectangle.

Parameters

- **xy** – Four points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`. The second point is just outside the drawn rectangle.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

PIL.ImageDraw.Draw.**shape**(*shape*, *fill=None*, *outline=None*)

Warning: This method is experimental.

Draw a shape.

PIL.ImageDraw.Draw.**text**(*xy*, *text*, *fill=None*, *font=None*, *anchor=None*)
Draws the string at the given position.

Parameters

- **xy** – Top left corner of the text.
- **text** – Text to be drawn.
- **font** – An `ImageFont` instance.
- **fill** – Color to use for the text.

PIL.ImageDraw.Draw.**textsize**(*text*, *font=None*)
Return the size of the given string, in pixels.

Parameters

- **text** – Text to be measured.
- **font** – An ImageFont instance.

4.5.6 Legacy API

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

PIL.ImageDraw.**ImageDraw**(*image*)

Return type *Draw*

PIL.ImageDraw.Draw.**setink**(*ink*)
Deprecated since version 1.1.5.

Sets the color to use for subsequent draw and fill operations.

PIL.ImageDraw.Draw.**setfill**(*fill*)
Deprecated since version 1.1.5.

Sets the fill mode.

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

PIL.ImageDraw.Draw.**setfont**(*font*)
Deprecated since version 1.1.5.

Sets the default font to use for the text method.

Parameters **font** – An ImageFont instance.

4.6 ImageEnhance Module

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

4.6.1 Example: Vary the sharpness of an image

```
from PIL 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.

4.6.2 Classes

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

class PIL.ImageEnhance.**_Enhance**

enhance(*factor*)
Returns an enhanced image.

Parameters **factor** – A floating point value controlling the enhancement. Factor 1.0 always returns a copy of the original image, lower factors mean less color (brightness, contrast, etc), and higher values more. There are no restrictions on this value.

Return type *Image*

class PIL.ImageEnhance.**Color**(*image*)
Adjust image color balance.

This class can be used to adjust the colour balance of an image, in a manner similar to the controls on a colour TV set. An enhancement factor of 0.0 gives a black and white image. A factor of 1.0 gives the original image.

class PIL.ImageEnhance.**Contrast**(*image*)
Adjust image contrast.

This class can be used to control the contrast of an image, similar to the contrast control on a TV set. An enhancement factor of 0.0 gives a solid grey image. A factor of 1.0 gives the original image.

class PIL.ImageEnhance.**Brightness**(*image*)
Adjust image brightness.

This class can be used to control the brightness of an image. An enhancement factor of 0.0 gives a black image. A factor of 1.0 gives the original image.

class PIL.ImageEnhance.**Sharpness**(*image*)
Adjust image sharpness.

This class can be used to adjust the sharpness of an image. An enhancement factor of 0.0 gives a blurred image, a factor of 1.0 gives the original image, and a factor of 2.0 gives a sharpened image.

4.7 ImageFile Module

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

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

4.7.1 Example: Parse an image

```
from PIL import ImageFile

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

p = ImageFile.Parser()

while 1:
    s = fp.read(1024)
```

```
    if not s:
        break
    p.feed(s)

im = p.close()

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

4.7.2 Parser

class PIL.ImageFile.**Parser**

Incremental image parser. This class implements the standard feed/close consumer interface.

In Python 2.x, this is an old-style class.

close()

(Consumer) Close the stream.

Returns An image object.

Raises IOError If the parser failed to parse the image file either because it cannot be identified or cannot be decoded.

feed(*data*)

(Consumer) Feed data to the parser.

Parameters **data** – A string buffer.

Raises IOError If the parser failed to parse the image file.

reset()

(Consumer) Reset the parser. Note that you can only call this method immediately after you've created a parser; parser instances cannot be reused.

4.8 ImageFilter Module

The ImageFilter module contains definitions for a pre-defined set of filters, which can be be used with the *Image.filter()* method.

4.8.1 Example: Filter an image

```
from PIL import ImageFilter

im1 = im.filter(ImageFilter.BLUR)

im2 = im.filter(ImageFilter.MinFilter(3))
im3 = im.filter(ImageFilter.MinFilter)  # same as MinFilter(3)
```

4.8.2 Filters

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

- **BLUR**
- **CONTOUR**

- **DETAIL**
- **EDGE_ENHANCE**
- **EDGE_ENHANCE_MORE**
- **EMBOSS**
- **FIND_EDGES**
- **SMOOTH**
- **SMOOTH_MORE**
- **SHARPEN**

class PIL.ImageFilter.**GaussianBlur**(*radius=2*)
Gaussian blur filter.

Parameters **radius** – Blur radius.

class PIL.ImageFilter.**UnsharpMask**(*radius=2*, *percent=150*, *threshold=3*)
Unsharp mask filter.

See Wikipedia's entry on digital unsharp masking for an explanation of the parameters.

Parameters

- **radius** – Blur Radius
- **percent** – Unsharp strength, in percent
- **threshold** – Threshold controls the minimum brightness change that will be sharpened

class PIL.ImageFilter.**Kernel**(*size*, *kernel*, *scale=None*, *offset=0*)
Create a convolution kernel. The current version only supports 3x3 and 5x5 integer and floating point kernels.

In the current version, kernels can only be applied to "L" and "RGB" images.

Parameters

- **size** – Kernel size, given as (width, height). In the current version, this must be (3,3) or (5,5).
- **kernel** – A sequence containing kernel weights.
- **scale** – Scale factor. If given, the result for each pixel is divided by this value. the default is the sum of the kernel weights.
- **offset** – Offset. If given, this value is added to the result, after it has been divided by the scale factor.

class PIL.ImageFilter.**RankFilter**(*size*, *rank*)
Create a rank filter. The rank filter sorts all pixels in a window of the given size, and returns the **rank**'th value.

Parameters

- **size** – The kernel size, in pixels.
- **rank** – What pixel value to pick. Use 0 for a min filter, `size * size / 2` for a median filter, `size * size - 1` for a max filter, etc.

class PIL.ImageFilter.**MedianFilter**(*size=3*)
Create a median filter. Picks the median pixel value in a window with the given size.

Parameters **size** – The kernel size, in pixels.

class PIL.ImageFilter.**MinFilter**(*size=3*)
Create a min filter. Picks the lowest pixel value in a window with the given size.

Parameters **size** – The kernel size, in pixels.

class PIL.ImageFilter.**MaxFilter**(*size=3*)
Create a max filter. Picks the largest pixel value in a window with the given size.

Parameters **size** – The kernel size, in pixels.

class PIL.ImageFilter.**ModeFilter**(*size=3*)
Create a mode filter. Picks the most frequent pixel value in a box with the given size. Pixel values that occur only once or twice are ignored; if no pixel value occurs more than twice, the original pixel value is preserved.

Parameters **size** – The kernel size, in pixels.

4.9 ImageFont Module

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

PIL uses its own font file format to store bitmap fonts. You can use the :command‘pilfont‘ utility to convert BDF and PCF font descriptors (X window font formats) to this format.

Starting with version 1.1.4, PIL can be configured to support TrueType and OpenType fonts (as well as other font formats supported by the FreeType library). For earlier versions, TrueType support is only available as part of the imToolkit package

4.9.1 Example

```
from PIL import ImageFont, ImageDraw

draw = ImageDraw.Draw(image)

# use a bitmap font
font = ImageFont.load("arial.pil")

draw.text((10, 10), "hello", font=font)

# use a truetype font
font = ImageFont.truetype("arial.ttf", 15)

draw.text((10, 25), "world", font=font)
```

4.9.2 Functions

PIL.ImageFont.**load**(*filename*)
Load a font file. This function loads a font object from the given bitmap font file, and returns the corresponding font object.

Parameters **filename** – Name of font file.

Returns A font object.

Raises IOError If the file could not be read.

PIL.ImageFont.**load_path**(*filename*)
Load font file. Same as *load()*, but searches for a bitmap font along the Python path.

Parameters **filename** – Name of font file.

Returns A font object.

Raises IOError If the file could not be read.

PIL.ImageFont.**truetype**(*font=None*, *size=10*, *index=0*, *encoding=''*, *filename=None*)
Load a TrueType or OpenType font file, and create a font object. This function loads a font object from the given file, and creates a font object for a font of the given size.

This function requires the _imagingft service.

Parameters

- **filename** – A truetype font file. Under Windows, if the file is not found in this filename, the loader also looks in Windows `fonts/` directory.
- **size** – The requested size, in points.
- **index** – Which font face to load (default is first available face).
- **encoding** – Which font encoding to use (default is Unicode). Common encodings are "unic" (Unicode), "symb" (Microsoft Symbol), "ADOB" (Adobe Standard), "ADBE" (Adobe Expert), and "armn" (Apple Roman). See the FreeType documentation for more information.

Returns A font object.

Raises IOError If the file could not be read.

PIL.ImageFont.**load_default**()
Load a "better than nothing" default font.

New in version 1.1.4.

Returns A font object.

4.9.3 Methods

PIL.ImageFont.ImageFont.**getsize**(*text*)

Returns (width, height)

PIL.ImageFont.ImageFont.**getmask**(*text*, *mode=''*)
Create a bitmap for the text.

If the font uses antialiasing, the bitmap should have mode "L" and use a maximum value of 255. Otherwise, it should have mode "1".

Parameters

- **text** – Text to render.
- **mode** – Used by some graphics drivers to indicate what mode the driver prefers; if empty, the renderer may return either mode. Note that the mode is always a string, to simplify C-level implementations.

  New in version 1.1.5.

Returns An internal PIL storage memory instance as defined by the PIL.Image.core interface module.

4.10 ImageGrab Module (Windows-only)

The ImageGrab module can be used to copy the contents of the screen or the clipboard to a PIL image memory.

Note: The current version works on Windows only.

New in version 1.1.3.

PIL.ImageGrab.**grab**(*bbox=None*)
Take a snapshot of the screen. The pixels inside the bounding box are returned as an "RGB" image. If the bounding box is omitted, the entire screen is copied.

New in version 1.1.3.

Parameters **bbox** – What region to copy. Default is the entire screen.

Returns An image

PIL.ImageGrab.**grabclipboard**()
Take a snapshot of the clipboard image, if any.

New in version 1.1.4.

Returns An image, a list of filenames, or None if the clipboard does not contain image data or filenames. Note that if a list is returned, the filenames may not represent image files.

4.11 ImageMath Module

The ImageMath module can be used to evaluate "image expressions". The module provides a single eval function, which takes an expression string and one or more images.

4.11.1 Example: Using the ImageMath module

```
import Image, ImageMath

im1 = Image.open("image1.jpg")
im2 = Image.open("image2.jpg")

out = ImageMath.eval("convert(min(a, b), 'L')", a=im1, b=im2)
out.save("result.png")
```

PIL.ImageMath.**eval**(*expression*, *environment*)
Evaluate expression in the given environment.

In the current version, *ImageMath* only supports single-layer images. To process multi-band images, use the *split()* method or *merge()* function.

Parameters

- **expression** – A string which uses the standard Python expression syntax. In addition to the standard operators, you can also use the functions described below.
- **environment** – A dictionary that maps image names to Image instances. You can use one or more keyword arguments instead of a dictionary, as shown in the above example. Note that the names must be valid Python identifiers.

Returns An image, an integer value, a floating point value, or a pixel tuple, depending on the expression.

4.11.2 Expression syntax

Expressions are standard Python expressions, but they're evaluated in a non-standard environment. You can use PIL methods as usual, plus the following set of operators and functions:

Standard Operators

You can use standard arithmetical operators for addition (+), subtraction (-), multiplication (*), and division (/).

The module also supports unary minus (-), modulo (%), and power (**) operators.

Note that all operations are done with 32-bit integers or 32-bit floating point values, as necessary. For example, if you add two 8-bit images, the result will be a 32-bit integer image. If you add a floating point constant to an 8-bit image, the result will be a 32-bit floating point image.

You can force conversion using the `convert()`, `float()`, and `int()` functions described below.

Bitwise Operators

The module also provides operations that operate on individual bits. This includes and (&), or (|), and exclusive or (^). You can also invert (~) all pixel bits.

Note that the operands are converted to 32-bit signed integers before the bitwise operation is applied. This means that you'll get negative values if you invert an ordinary greyscale image. You can use the and (&) operator to mask off unwanted bits.

Bitwise operators don't work on floating point images.

Logical Operators

Logical operators like `and`, `or`, and `not` work on entire images, rather than individual pixels.

An empty image (all pixels zero) is treated as false. All other images are treated as true.

Note that `and` and `or` return the last evaluated operand, while not always returns a boolean value.

Built-in Functions

These functions are applied to each individual pixel.

abs(*image*)
Absolute value.

convert(*image*, *mode*)
Convert image to the given mode. The mode must be given as a string constant.

float(*image*)
Convert image to 32-bit floating point. This is equivalent to convert(image, "F").

int(*image*)
Convert image to 32-bit integer. This is equivalent to convert(image, "I").

Note that 1-bit and 8-bit images are automatically converted to 32-bit integers if necessary to get a correct result.

max(*image1*, *image2*)
Maximum value.

min(*image1*, *image2*)
Minimum value.

4.12 ImageMorph Module

The ImageMorph module provides morphology operations on images.

class PIL.ImageMorph.**LutBuilder**(*patterns=None*, *op_name=None*)
A class for building a MorphLut from a descriptive language

The input patterns is a list of a strings sequences like these:

```
4:(...
   .1.
   111)->1
```

(whitespaces including linebreaks are ignored). The option 4 describes a series of symmetry operations (in this case a 4-rotation), the pattern is described by:

- . or X - Ignore
- 1 - Pixel is on
- 0 - Pixel is off

The result of the operation is described after "->" string.

The default is to return the current pixel value, which is returned if no other match is found.

Operations:

- 4 - 4 way rotation
- N - Negate
- 1 - Dummy op for no other operation (an op must always be given)
- M - Mirroring

Example:

```
lb = LutBuilder(patterns = ["4:(... .1. 111)->1"])
lut = lb.build_lut()
```

add_patterns(*patterns*)

build_default_lut()

build_lut()
Compile all patterns into a morphology lut.

TBD :Build based on (file) morphlut:modify_lut

get_lut()

class PIL.ImageMorph.**MorphOp**(*lut=None*, *op_name=None*, *patterns=None*)
A class for binary morphological operators

apply(*image*)
Run a single morphological operation on an image

Returns a tuple of the number of changed pixels and the morphed image

get_on_pixels(*image*)
Get a list of all turned on pixels in a binary image

Returns a list of tuples of (x,y) coordinates of all matching pixels.

load_lut(*filename*)
Load an operator from an mrl file

match(*image*)
Get a list of coordinates matching the morphological operation on an image.

Returns a list of tuples of (x,y) coordinates of all matching pixels.

save_lut(*filename*)
Save an operator to an mrl file

set_lut(*lut*)
Set the lut from an external source

4.13 ImageOps Module

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

Only bug fixes have been added since the Pillow fork.

New in version 1.1.3.

PIL.ImageOps.**autocontrast**(*image*, *cutoff=0*, *ignore=None*)
Maximize (normalize) image contrast. This function calculates a histogram of the input image, removes **cutoff** percent of the lightest and darkest pixels from the histogram, and remaps the image so that the darkest pixel becomes black (0), and the lightest becomes white (255).

Parameters

- **image** – The image to process.
- **cutoff** – How many percent to cut off from the histogram.
- **ignore** – The background pixel value (use None for no background).

Returns An image.

PIL.ImageOps.**colorize**(*image*, *black*, *white*)
Colorize grayscale image. The **black** and **white** arguments should be RGB tuples; this function calculates a color wedge mapping all black pixels in the source image to the first color, and all white pixels to the second color.

Parameters

- **image** – The image to colorize.
- **black** – The color to use for black input pixels.
- **white** – The color to use for white input pixels.

Returns An image.

PIL.ImageOps.**crop**(*image*, *border=0*)
Remove border from image. The same amount of pixels are removed from all four sides. This function works on all image modes.

See also:

crop()

Parameters

- **image** – The image to crop.
- **border** – The number of pixels to remove.

Returns An image.

PIL.ImageOps.**deform**(*image*, *deformer*, *resample=2*)
Deform the image.

Parameters

- **image** – The image to deform.
- **deformer** – A deformer object. Any object that implements a **getmesh** method can be used.
- **resample** – What resampling filter to use.

Returns An image.

PIL.ImageOps.**equalize**(*image*, *mask=None*)
Equalize the image histogram. This function applies a non-linear mapping to the input image, in order to create a uniform distribution of grayscale values in the output image.

Parameters

- **image** – The image to equalize.
- **mask** – An optional mask. If given, only the pixels selected by the mask are included in the analysis.

Returns An image.

PIL.ImageOps.**expand**(*image*, *border=0*, *fill=0*)
Add border to the image

Parameters

- **image** – The image to expand.
- **border** – Border width, in pixels.
- **fill** – Pixel fill value (a color value). Default is 0 (black).

Returns An image.

PIL.ImageOps.**fit**(*image*, *size*, *method=0*, *bleed=0.0*, *centering=(0.5*, *0.5)*)
Returns a sized and cropped version of the image, cropped to the requested aspect ratio and size.

This function was contributed by Kevin Cazabon.

Parameters

- **size** – The requested output size in pixels, given as a (width, height) tuple.
- **method** – What resampling method to use. Default is PIL.Image.NEAREST.
- **bleed** – Remove a border around the outside of the image (from all four edges. The value is a decimal percentage (use 0.01 for one percent). The default value is 0 (no border).

- **centering** – Control the cropping position. Use (0.5, 0.5) for center cropping (e.g. if cropping the width, take 50% off of the left side, and therefore 50% off the right side). (0.0, 0.0) will crop from the top left corner (i.e. if cropping the width, take all of the crop off of the right side, and if cropping the height, take all of it off the bottom). (1.0, 0.0) will crop from the bottom left corner, etc. (i.e. if cropping the width, take all of the crop off the left side, and if cropping the height take none from the top, and therefore all off the bottom).

Returns An image.

PIL.ImageOps.**flip**(*image*)
Flip the image vertically (top to bottom).

Parameters **image** – The image to flip.

Returns An image.

PIL.ImageOps.**grayscale**(*image*)
Convert the image to grayscale.

Parameters **image** – The image to convert.

Returns An image.

PIL.ImageOps.**invert**(*image*)
Invert (negate) the image.

Parameters **image** – The image to invert.

Returns An image.

PIL.ImageOps.**mirror**(*image*)
Flip image horizontally (left to right).

Parameters **image** – The image to mirror.

Returns An image.

PIL.ImageOps.**posterize**(*image*, *bits*)
Reduce the number of bits for each color channel.

Parameters

- **image** – The image to posterize.
- **bits** – The number of bits to keep for each channel (1-8).

Returns An image.

PIL.ImageOps.**solarize**(*image*, *threshold=128*)
Invert all pixel values above a threshold.

Parameters

- **image** – The image to solarize.
- **threshold** – All pixels above this greyscale level are inverted.

Returns An image.

4.14 ImagePalette Module

The ImagePalette module contains a class of the same name to represent the color palette of palette mapped images.

Note: This module was never well-documented. It hasn't changed since 2001, though, so it's probably safe for you to read the source code and puzzle out the internals if you need to.

The *ImagePalette* class has several methods, but they are all marked as "experimental." Read that as you will. The [source] link is there for a reason.

---

class PIL.ImagePalette.**ImagePalette**(*mode='RGB'*, *palette=None*, *size=0*)

Color palette for palette mapped images

getcolor(*color*)

Given an rgb tuple, allocate palette entry.

> **Warning:** This method is experimental.

getdata()

Get palette contents in format suitable # for the low-level im.putpalette primitive.

> **Warning:** This method is experimental.

save(*fp*)

Save palette to text file.

> **Warning:** This method is experimental.

tobytes()

Convert palette to bytes.

> **Warning:** This method is experimental.

tostring()

Convert palette to bytes.

> **Warning:** This method is experimental.

4.15 ImagePath Module

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

class PIL.ImagePath.**Path**

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

You can also create a path object from another path object.

In 1.1.6 and later, you can also pass in any object that implements Python's buffer API. The buffer should provide read access, and contain C floats in machine byte order.

The path object implements most parts of the Python sequence interface, and behaves like a list of (x, y) pairs. You can use len(), item access, and slicing as usual. However, the current version does not support slice assignment, or item and slice deletion.

Parameters **xy** – A sequence. The sequence can contain 2-tuples [(x, y), ...] or a flat list of numbers [x, y, ...].

PIL.ImagePath.Path.**compact**(*distance=2*)
Compacts the path, by removing points that are close to each other. This method modifies the path in place, and returns the number of points left in the path.

distance is measured as Manhattan distance and defaults to two pixels.

PIL.ImagePath.Path.**getbbox**()
Gets the bounding box of the path.

Returns `(x0, y0, x1, y1)`

PIL.ImagePath.Path.**map**(*function*)
Maps the path through a function.

PIL.ImagePath.Path.**tolist**(*flat=0*)
Converts the path to a Python list [(x, y), . . .].

Parameters **flat** – By default, this function returns a list of 2-tuples [(x, y), ...]. If this argument is `True`, it returns a flat list [x, y, ...] instead.

Returns A list of coordinates. See **flat**.

PIL.ImagePath.Path.**transform**(*matrix*)
Transforms the path in place, using an affine transform. The matrix is a 6-tuple (a, b, c, d, e, f), and each point is mapped as follows:

```
xOut = xIn * a + yIn * b + c
yOut = xIn * d + yIn * e + f
```

4.16 ImageQt Module

The `ImageQt` module contains support for creating PyQt4 or PyQt5 QImage objects from PIL images.

New in version 1.1.6.

class ImageQt.**ImageQt**(*image*)
Creates an `ImageQt` object from a PIL *Image* object. This class is a subclass of QtGui.QImage, which means that you can pass the resulting objects directly to PyQt4/5 API functions and methods.

This operation is currently supported for mode 1, L, P, RGB, and RGBA images. To handle other modes, you need to convert the image first.

4.17 ImageSequence Module

The `ImageSequence` module contains a wrapper class that lets you iterate over the frames of an image sequence.

4.17.1 Extracting frames from an animation

```
from PIL import Image, ImageSequence

im = Image.open("animation.fli")

index = 1
```

```
for frame in ImageSequence.Iterator(im):
    frame.save("frame%d.png" % index)
    index = index + 1
```

4.17.2 The `Iterator` class

class `PIL.ImageSequence.`**`Iterator`**(*im*)
This class implements an iterator object that can be used to loop over an image sequence.

You can use the `[]` operator to access elements by index. This operator will raise an `IndexError` if you try to access a nonexistent frame.

Parameters **`im`** – An image object.

4.18 `ImageStat` Module

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

class `PIL.ImageStat.`**`Stat`**(*image_or_list*, *mask=None*)
Calculate statistics for the given image. If a mask is included, only the regions covered by that mask are included in the statistics. You can also pass in a previously calculated histogram.

Parameters

- **`image`** – A PIL image, or a precalculated histogram.
- **`mask`** – An optional mask.

`extrema`
Min/max values for each band in the image.

`count`
Total number of pixels for each band in the image.

`sum`
Sum of all pixels for each band in the image.

`sum2`
Squared sum of all pixels for each band in the image.

`mean`
Average (arithmetic mean) pixel level for each band in the image.

`median`
Median pixel level for each band in the image.

`rms`
RMS (root-mean-square) for each band in the image.

`var`
Variance for each band in the image.

`stddev`
Standard deviation for each band in the image.

4.19 ImageTk Module

The ImageTk module contains support to create and modify Tkinter BitmapImage and PhotoImage objects from PIL images.

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

class PIL.ImageTk.**BitmapImage**(*image=None*, ***kw*)

A Tkinter-compatible bitmap image. This can be used everywhere Tkinter expects an image object.

The given image must have mode "1". Pixels having value 0 are treated as transparent. Options, if any, are passed on to Tkinter. The most commonly used option is **foreground**, which is used to specify the color for the non-transparent parts. See the Tkinter documentation for information on how to specify colours.

Parameters **image** – A PIL image.

height()

Get the height of the image.

Returns The height, in pixels.

width()

Get the width of the image.

Returns The width, in pixels.

class PIL.ImageTk.**PhotoImage**(*image=None*, *size=None*, ***kw*)

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

The constructor takes either a PIL image, or a mode and a size. Alternatively, you can use the **file** or **data** options to initialize the photo image object.

Parameters

- **image** – Either a PIL image, or a mode string. If a mode string is used, a size must also be given.
- **size** – If the first argument is a mode string, this defines the size of the image.
- **file** – A filename to load the image from (using Image.open(file)).
- **data** – An 8-bit string containing image data (as loaded from an image file).

height()

Get the height of the image.

Returns The height, in pixels.

paste(*im*, *box=None*)

Paste a PIL image into the photo image. Note that this can be very slow if the photo image is displayed.

Parameters

- **im** – A PIL image. The size must match the target region. If the mode does not match, the image is converted to the mode of the bitmap image.
- **box** – 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.

width()

Get the width of the image.

Returns The width, in pixels.

4.20 ImageWin Module (Windows-only)

The ImageWin module contains support to create and display images on Windows.

ImageWin can be used with PythonWin and other user interface toolkits that provide access to Windows device contexts or window handles. For example, Tkinter makes the window handle available via the winfo_id method:

```
from PIL import ImageWin

dib = ImageWin.Dib(...)

hwnd = ImageWin.HWND(widget.winfo_id())
dib.draw(hwnd, xy)
```

class PIL.ImageWin.**Dib**(*image*, *size=None*)

A Windows bitmap with the given mode and size. The mode can be one of "1", "L", "P", 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.

Parameters

- **image** – Either a PIL image, or a mode string. If a mode string is used, a size must also be given. The mode can be one of "1", "L", "P", or "RGB".
- **size** – If the first argument is a mode string, this defines the size of the image.

draw(*handle*, *dst*, *src=None*)

Same as expose, but allows you to specify where to draw the image, and what part of it to draw.

The destination and source areas are given as 4-tuple rectangles. If the source is omitted, the entire image is copied. If the source and the destination have different sizes, the image is resized as necessary.

expose(*handle*)

Copy the bitmap contents to a device context.

Parameters **handle** – Device context (HDC), cast to a Python integer, or an HDC or HWND instance. In PythonWin, you can use the CDC.GetHandleAttrib() to get a suitable handle.

frombytes(*buffer*)

Load display memory contents from byte data.

Parameters **buffer** – A buffer containing display data (usually data returned from tobytes)

paste(*im*, *box=None*)

Paste a PIL image into the bitmap image.

Parameters

- **im** – A PIL image. The size must match the target region. If the mode does not match, the image is converted to the mode of the bitmap image.
- **box** – 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.

query_palette(*handle*)

Installs the palette associated with the image in the given device context.

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.

Parameters **handle** – Device context (HDC), cast to a Python integer, or an HDC or HWND instance.

Returns A true value if one or more entries were changed (this indicates that the image should be redrawn).

tobytes()
Copy display memory contents to bytes object.

Returns A bytes object containing display data.

class PIL.ImageWin.**HDC**(*dc*)
Wraps an HDC integer. The resulting object can be passed to the *draw()* and *expose()* methods.

class PIL.ImageWin.**HWND**(*wnd*)
Wraps an HWND integer. The resulting object can be passed to the *draw()* and *expose()* methods, instead of a DC.

4.21 ExifTags Module

The ExifTags module exposes two dictionaries which provide constants and clear-text names for various well-known EXIF tags.

class PIL.ExifTags.**TAGS**
The TAG dictionary maps 16-bit integer EXIF tag enumerations to descriptive string names. For instance:

```
>>> from PIL.ExifTags import TAGS
>>> TAGS[0x010e]
'ImageDescription'
```

class PIL.ExifTags.**GPSTAGS**
The GPSTAGS dictionary maps 8-bit integer EXIF gps enumerations to descriptive string names. For instance:

```
>>> from PIL.ExifTags import GPSTAGS
>>> GPSTAGS[20]
'GPSDestLatitude'
```

4.22 OleFileIO Module

The OleFileIO module reads Microsoft OLE2 files (also called Structured Storage or Microsoft Compound Document File Format), such as Microsoft Office documents, Image Composer and FlashPix files, and Outlook messages.

This module is the OleFileIO_PL project by Philippe Lagadec, v0.30, merged back into Pillow.

4.22.1 How to use this module

For more information, see also the file **PIL/OleFileIO.py**, sample code at the end of the module itself, and docstrings within the code.

About the structure of OLE files

An OLE file can be seen as a mini file system or a Zip archive: It contains **streams** of data that look like files embedded within the OLE file. Each stream has a name. For example, the main stream of a MS Word document containing its text is named "WordDocument".

An OLE file can also contain **storages**. A storage is a folder that contains streams or other storages. For example, a MS Word document with VBA macros has a storage called "Macros".

Special streams can contain **properties**. A property is a specific value that can be used to store information such as the metadata of a document (title, author, creation date, etc). Property stream names usually start with the character '05'.

For example, a typical MS Word document may look like this:

```
\x05DocumentSummaryInformation (stream)
\x05SummaryInformation (stream)
WordDocument (stream)
Macros (storage)
    PROJECT (stream)
    PROJECTwm (stream)
    VBA (storage)
        Module1 (stream)
        ThisDocument (stream)
        _VBA_PROJECT (stream)
        dir (stream)
ObjectPool (storage)
```

Test if a file is an OLE container

Use isOleFile to check if the first bytes of the file contain the Magic for OLE files, before opening it. isOleFile returns True if it is an OLE file, False otherwise.

```
assert OleFileIO.isOleFile('myfile.doc')
```

Open an OLE file from disk

Create an OleFileIO object with the file path as parameter:

```
ole = OleFileIO.OleFileIO('myfile.doc')
```

Open an OLE file from a file-like object

This is useful if the file is not on disk, e.g. already stored in a string or as a file-like object.

```
ole = OleFileIO.OleFileIO(f)
```

For example the code below reads a file into a string, then uses BytesIO to turn it into a file-like object.

```
data = open('myfile.doc', 'rb').read()
f = io.BytesIO(data) # or StringIO.StringIO for Python 2.x
ole = OleFileIO.OleFileIO(f)
```

How to handle malformed OLE files

By default, the parser is configured to be as robust and permissive as possible, allowing to parse most malformed OLE files. Only fatal errors will raise an exception. It is possible to tell the parser to be more strict in order to raise exceptions for files that do not fully conform to the OLE specifications, using the raise_defect option:

```
ole = OleFileIO.OleFileIO('myfile.doc', raise_defects=DEFECT_INCORRECT)
```

When the parsing is done, the list of non-fatal issues detected is available as a list in the parsing_issues attribute of the OleFileIO object:

```
print('Non-fatal issues raised during parsing:')
if ole.parsing_issues:
    for exctype, msg in ole.parsing_issues:
        print('- %s: %s' % (exctype.__name__, msg))
else:
    print('None')
```

Syntax for stream and storage path

Two different syntaxes are allowed for methods that need or return the path of streams and storages:

1. Either a **list of strings** including all the storages from the root up to the stream/storage name. For example a stream called "WordDocument" at the root will have ['WordDocument'] as full path. A stream called "ThisDocument" located in the storage "Macros/VBA" will be ['Macros', 'VBA', 'ThisDocument']. This is the original syntax from PIL. While hard to read and not very convenient, this syntax works in all cases.
2. Or a **single string with slashes** to separate storage and stream names (similar to the Unix path syntax). The previous examples would be 'WordDocument' and 'Macros/VBA/ThisDocument'. This syntax is easier, but may fail if a stream or storage name contains a slash.

Both are case-insensitive.

Switching between the two is easy:

```
slash_path = '/'.join(list_path)
list_path  = slash_path.split('/')
```

Get the list of streams

listdir() returns a list of all the streams contained in the OLE file, including those stored in storages. Each stream is listed itself as a list, as described above.

```
print(ole.listdir())
```

Sample result:

```
[['\x01CompObj'], ['\x05DocumentSummaryInformation'], ['\x05SummaryInformation']
, ['1Table'], ['Macros', 'PROJECT'], ['Macros', 'PROJECTwm'], ['Macros', 'VBA',
'Module1'], ['Macros', 'VBA', 'ThisDocument'], ['Macros', 'VBA', '_VBA_PROJECT']
, ['Macros', 'VBA', 'dir'], ['ObjectPool'], ['WordDocument']]
```

As an option it is possible to choose if storages should also be listed, with or without streams:

```
ole.listdir (streams=False, storages=True)
```

Test if known streams/storages exist:

exists(path) checks if a given stream or storage exists in the OLE file.

```
if ole.exists('worddocument'):
    print("This is a Word document.")
    if ole.exists('macros/vba'):
        print("This document seems to contain VBA macros.")
```

Read data from a stream

openstream(path) opens a stream as a file-like object.

The following example extracts the "Pictures" stream from a PPT file:

```
pics = ole.openstream('Pictures')
data = pics.read()
```

Get information about a stream/storage

Several methods can provide the size, type and timestamps of a given stream/storage:

get_size(path) returns the size of a stream in bytes:

```
s = ole.get_size('WordDocument')
```

get_type(path) returns the type of a stream/storage, as one of the following constants: STGTY_STREAM for a stream, STGTY_STORAGE for a storage, STGTY_ROOT for the root entry, and False for a non existing path.

```
t = ole.get_type('WordDocument')
```

get_ctime(path) and get_mtime(path) return the creation and modification timestamps of a stream/storage, as a Python datetime object with UTC timezone. Please note that these timestamps are only present if the application that created the OLE file explicitly stored them, which is rarely the case. When not present, these methods return None.

```
c = ole.get_ctime('WordDocument')
m = ole.get_mtime('WordDocument')
```

The root storage is a special case: You can get its creation and modification timestamps using the OleFileIO.root attribute:

```
c = ole.root.getctime()
m = ole.root.getmtime()
```

Extract metadata

get_metadata() will check if standard property streams exist, parse all the properties they contain, and return an OleMetadata object with the found properties as attributes.

```
meta = ole.get_metadata()
print('Author:', meta.author)
print('Title:', meta.title)
print('Creation date:', meta.create_time)
# print all metadata:
meta.dump()
```

Available attributes include:

```
codepage, title, subject, author, keywords, comments, template,
last_saved_by, revision_number, total_edit_time, last_printed, create_time,
last_saved_time, num_pages, num_words, num_chars, thumbnail,
creating_application, security, codepage_doc, category, presentation_target,
bytes, lines, paragraphs, slides, notes, hidden_slides, mm_clips,
scale_crop, heading_pairs, titles_of_parts, manager, company, links_dirty,
chars_with_spaces, unused, shared_doc, link_base, hlinks, hlinks_changed,
version, dig_sig, content_type, content_status, language, doc_version
```

See the source code of the OleMetadata class for more information.

Parse a property stream

get_properties(path) can be used to parse any property stream that is not handled by get_metadata. It returns a dictionary indexed by integers. Each integer is the index of the property, pointing to its value. For example in the standard property stream '05SummaryInformation', the document title is property #2, and the subject is #3.

```
p = ole.getproperties('specialprops')
```

By default as in the original PIL version, timestamp properties are converted into a number of seconds since Jan 1,1601. With the option convert_time, you can obtain more convenient Python datetime objects (UTC timezone). If some time properties should not be converted (such as total editing time in '05SummaryInformation'), the list of indexes can be passed as no_conversion:

```
p = ole.getproperties('specialprops', convert_time=True, no_conversion=[10])
```

Close the OLE file

Unless your application is a simple script that terminates after processing an OLE file, do not forget to close each OleFileIO object after parsing to close the file on disk.

```
ole.close()
```

Use OleFileIO as a script

OleFileIO can also be used as a script from the command-line to display the structure of an OLE file and its metadata, for example:

```
PIL/OleFileIO.py myfile.doc
```

You can use the option -c to check that all streams can be read fully, and -d to generate very verbose debugging information.

4.22.2 How to contribute

The code is available in a Mercurial repository on bitbucket. You may use it to submit enhancements or to report any issue.

If you would like to help us improve this module, or simply provide feedback, please contact me. You can help in many ways:

- test this module on different platforms / Python versions

- find and report bugs
- improve documentation, code samples, docstrings
- write unittest test cases
- provide tricky malformed files

4.22.3 How to report bugs

To report a bug, for example a normal file which is not parsed correctly, please use the issue reporting page, or if you prefer to do it privately, use this contact form. Please provide all the information about the context and how to reproduce the bug.

If possible please join the debugging output of OleFileIO. For this, launch the following command :

```
PIL/OleFileIO.py -d -c file >debug.txt
```

4.22.4 Classes and Methods

class PIL.OleFileIO.**OleFileIO**(*filename=None*, *raise_defects=40*)
OLE container object

This class encapsulates the interface to an OLE 2 structured storage file. Use the `listdir()` and `openstream()` methods to access the contents of this file.

Object names are given as a list of strings, one for each subentry level. The root entry should be omitted. For example, the following code extracts all image streams from a Microsoft Image Composer file:

```
ole = OleFileIO("fan.mic")

for entry in ole.listdir():
    if entry[1:2] == "Image":
        fin = ole.openstream(entry)
        fout = open(entry[0:1], "wb")
        while True:
            s = fin.read(8192)
            if not s:
                break
            fout.write(s)
```

You can use the viewer application provided with the Python Imaging Library to view the resulting files (which happens to be standard TIFF files).

close()
close the OLE file, to release the file object

dumpdirectory()
Dump directory (for debugging only)

dumpfat(*fat*, *firstindex=0*)
Displays a part of FAT in human-readable form for debugging purpose

dumpsect(*sector*, *firstindex=0*)
Displays a sector in a human-readable form, for debugging purpose.

exists(*filename*)
Test if given filename exists as a stream or a storage in the OLE container.

Parameters **filename** – path of stream in storage tree. (see openstream for syntax)

Returns True if object exist, else False.

get_metadata()
Parse standard properties streams, return an OleMetadata object containing all the available metadata. (also stored in the metadata attribute of the OleFileIO object)

new in version 0.25

get_rootentry_name()
Return root entry name. Should usually be 'Root Entry' or 'R' in most implementations.

get_size(*filename*)
Return size of a stream in the OLE container, in bytes.

Parameters **filename** – path of stream in storage tree (see openstream for syntax)

Returns size in bytes (long integer)

Raises

- **IOError** – if file not found
- **TypeError** – if this is not a stream

get_type(*filename*)
Test if given filename exists as a stream or a storage in the OLE container, and return its type.

Parameters **filename** – path of stream in storage tree. (see openstream for syntax)

Returns

False if object does not exist, its entry type (>0) otherwise:

- STGTY_STREAM: a stream
- STGTY_STORAGE: a storage
- STGTY_ROOT: the root entry

getctime(*filename*)
Return creation time of a stream/storage.

Parameters **filename** – path of stream/storage in storage tree. (see openstream for syntax)

Returns None if creation time is null, a python datetime object otherwise (UTC timezone)

new in version 0.26

getmtime(*filename*)
Return modification time of a stream/storage.

Parameters **filename** – path of stream/storage in storage tree. (see openstream for syntax)

Returns None if modification time is null, a python datetime object otherwise (UTC timezone)

new in version 0.26

getproperties(*filename*, *convert_time=False*, *no_conversion=None*)
Return properties described in substream.

Parameters

- **filename** – path of stream in storage tree (see openstream for syntax)
- **convert_time** – bool, if True timestamps will be converted to Python datetime
- **no_conversion** – None or list of int, timestamps not to be converted (for example total editing time is not a real timestamp)

Returns a dictionary of values indexed by id (integer)

getsect(*sect*)
Read given sector from file on disk.

Parameters **sect** – sector index

Returns a string containing the sector data.

listdir(*streams=True*, *storages=False*)
Return a list of streams stored in this file

Parameters

- **streams** – bool, include streams if True (True by default) - new in v0.26
- **storages** – bool, include storages if True (False by default) - new in v0.26 (note: the root storage is never included)

loaddirectory(*sect*)
Load the directory.

Parameters **sect** – sector index of directory stream.

loadfat(*header*)
Load the FAT table.

loadfat_sect(*sect*)
Adds the indexes of the given sector to the FAT

Parameters **sect** – string containing the first FAT sector, or array of long integers

Returns index of last FAT sector.

loadminifat()
Load the MiniFAT table.

open(*filename*)
Open an OLE2 file. Reads the header, FAT and directory.

Parameters **filename** – string-like or file-like object

openstream(*filename*)
Open a stream as a read-only file object (BytesIO).

Parameters **filename** – path of stream in storage tree (except root entry), either:

- a string using Unix path syntax, for example: 'storage_1/storage_1.2/stream'
- a list of storage filenames, path to the desired stream/storage. Example: ['storage_1', 'storage_1.2', 'stream']

Returns file object (read-only)

Raises IOError if filename not found, or if this is not a stream.

sect2array(*sect*)
convert a sector to an array of 32 bits unsigned integers, swapping bytes on big endian CPUs such as PowerPC (old Macs)

PIL.OleFileIO.**isOleFile**(*filename*)
Test if file is an OLE container (according to its header).

Parameters **filename** – file name or path (str, unicode)

Returns True if OLE, False otherwise.

4.23 PSDraw Module

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

class PIL.PSDraw.**PSDraw**(*fp=None*)
Sets up printing to the given file. If **file** is omitted, sys.stdout is assumed.

begin_document(*id=None*)
Set up printing of a document. (Write Postscript DSC header.)

end_document()
Ends printing. (Write Postscript DSC footer.)

image(*box*, *im*, *dpi=None*)
Draw a PIL image, centered in the given box.

line(*xy0*, *xy1*)
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(*box*)
Draws a rectangle.

Parameters **box** – A 4-tuple of integers whose order and function is currently undocumented.

Hint: the tuple is passed into this format string:

```
%d %d M %d %d 0 Vr
```

setfont(*font*, *size*)
Selects which font to use.

Parameters

- **font** – A Postscript font name
- **size** – Size in points.

text(*xy*, *text*)
Draws text at the given position. You must use *setfont()* before calling this method.

4.24 PixelAccess Class

The PixelAccess class provides read and write access to *PIL.Image* data at a pixel level.

Note: Accessing individual pixels is fairly slow. If you are looping over all of the pixels in an image, there is likely a faster way using other parts of the Pillow API.

4.24.1 Example

The following script loads an image, accesses one pixel from it, then changes it.

```
from PIL import Image
im = Image.open('hopper.jpg')
px = im.load()
print (px[4,4])
```

```
px[4,4] = (0,0,0)
print (px[4,4])
```

Results in the following:

```
(23, 24, 68)
(0, 0, 0)
```

4.24.2 `PixelAccess` Class

class `PixelAccess`

`__setitem__(self, xy, color):`
Modifies the pixel at x,y. The color is given as a single numerical value for single band images, and a tuple for multi-band images

Parameters

- **`xy`** – The pixel coordinate, given as (x, y).
- **`value`** – The pixel value.

`__getitem__(self, xy):`

Returns the pixel at x,y. The pixel is returned as a single value for single band images or a tuple for multiple band images

param xy The pixel coordinate, given as (x, y).

returns a pixel value for single band images, a tuple of pixel values for multiband images.

`putpixel(self, xy, color):`
Modifies the pixel at x,y. The color is given as a single numerical value for single band images, and a tuple for multi-band images

Parameters

- **`xy`** – The pixel coordinate, given as (x, y).
- **`value`** – The pixel value.

`getpixel(self, xy):`

Returns the pixel at x,y. The pixel is returned as a single value for single band images or a tuple for multiple band images

param xy The pixel coordinate, given as (x, y).

returns a pixel value for single band images, a tuple of pixel values for multiband images.

4.25 `PyAccess` Module

The `PyAccess` module provides a CFFI/Python implementation of the *PixelAccess Class*. This implementation is far faster on PyPy than the PixelAccess version.

Note: Accessing individual pixels is fairly slow. If you are looping over all of the pixels in an image, there is likely a faster way using other parts of the Pillow API.

4.25.1 Example

The following script loads an image, accesses one pixel from it, then changes it.

```
from PIL import Image
im = Image.open('hopper.jpg')
px = im.load()
print (px[4,4])
px[4,4] = (0,0,0)
print (px[4,4])
```

Results in the following:

```
(23, 24, 68)
(0, 0, 0)
```

4.25.2 `PyAccess` Class

4.26 PIL Package (autodoc of remaining modules)

Reference for modules whose documentation has not yet been ported or written can be found here.

4.26.1 `BdfFontFile` Module

class PIL.BdfFontFile.**BdfFontFile**(*fp*)

Bases: *PIL.FontFile.FontFile*

PIL.BdfFontFile.**bdf_char**(*f*)

4.26.2 `ContainerIO` Module

class PIL.ContainerIO.**ContainerIO**(*file*, *offset*, *length*)

isatty()

read(*n=0*)

readline()

readlines()

seek(*offset*, *mode=0*)

tell()

4.26.3 `FontFile` Module

class PIL.FontFile.**FontFile**

bitmap = None

compile()

Create metrics and bitmap

save (*filename*)
Save font

PIL.FontFile.**puti16** (*fp*, *values*)

4.26.4 GdImageFile Module

class PIL.GdImageFile.**GdImageFile** (*fp=None*, *filename=None*)
Bases: PIL.ImageFile.ImageFile

format = 'GD'

format_description = 'GD uncompressed images'

PIL.GdImageFile.**open** (*fp*, *mode='r'*)

4.26.5 GimpGradientFile Module

class PIL.GimpGradientFile.**GimpGradientFile** (*fp*)
Bases: *PIL.GimpGradientFile.GradientFile*

class PIL.GimpGradientFile.**GradientFile**

getpalette (*entries=256*)

gradient = None

PIL.GimpGradientFile.**curved** (*middle*, *pos*)

PIL.GimpGradientFile.**linear** (*middle*, *pos*)

PIL.GimpGradientFile.**sine** (*middle*, *pos*)

PIL.GimpGradientFile.**sphere_decreasing** (*middle*, *pos*)

PIL.GimpGradientFile.**sphere_increasing** (*middle*, *pos*)

4.26.6 GimpPaletteFile Module

class PIL.GimpPaletteFile.**GimpPaletteFile** (*fp*)

getpalette ()

rawmode = 'RGB'

4.26.7 ImageDraw2 Module

class PIL.ImageDraw2.**Brush** (*color*, *opacity=255*)

class PIL.ImageDraw2.**Draw** (*image*, *size=None*, *color=None*)

arc (*xy*, *start*, *end*, **options*)

chord (*xy*, *start*, *end*, **options*)

ellipse (*xy*, **options*)

flush()

line(*xy*, **options*)

pieslice(*xy*, *start*, *end*, **options*)

polygon(*xy*, **options*)

rectangle(*xy*, **options*)

render(*op*, *xy*, *pen*, *brush=None*)

settransform(*offset*)

symbol(*xy*, *symbol*, **options*)

text(*xy*, *text*, *font*)

textsize(*text*, *font*)

class PIL.ImageDraw2.**Font**(*color*, *file*, *size=12*)

class PIL.ImageDraw2.**Pen**(*color*, *width=1*, *opacity=255*)

4.26.8 ImageFileIO Module

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

Deprecated. New code should use the *PIL.ImageFile.Parser* class in the *PIL.ImageFile* module instead.

See also:

modules *PIL.ImageFile.Parser*

class PIL.ImageFileIO.**ImageFileIO**(*fp*)
Bases: _io.BytesIO

4.26.9 ImageShow Module

class PIL.ImageShow.**DisplayViewer**
Bases: *PIL.ImageShow.UnixViewer*

get_command_ex(*file*, ***options*)

class PIL.ImageShow.**UnixViewer**
Bases: *PIL.ImageShow.Viewer*

show_file(*file*, ***options*)

class PIL.ImageShow.**Viewer**

format = None

get_command(*file*, ***options*)

get_format(*image*)

save_image(*image*)

show(*image*, ***options*)

show_file(*file*, ***options*)

show_image(*image*, ***options*)

class PIL.ImageShow.**XVViewer**
Bases: *PIL.ImageShow.UnixViewer*

get_command_ex(*file*, *title=None*, ***options*)

PIL.ImageShow.**register**(*viewer*, *order=1*)

PIL.ImageShow.**show**(*image*, *title=None*, ***options*)

PIL.ImageShow.**which**(*executable*)

4.26.10 ImageTransform Module

class PIL.ImageTransform.**AffineTransform**(*data*)
Bases: *PIL.ImageTransform.Transform*

method = 0

class PIL.ImageTransform.**ExtentTransform**(*data*)
Bases: *PIL.ImageTransform.Transform*

method = 1

class PIL.ImageTransform.**MeshTransform**(*data*)
Bases: *PIL.ImageTransform.Transform*

method = 4

class PIL.ImageTransform.**QuadTransform**(*data*)
Bases: *PIL.ImageTransform.Transform*

method = 3

class PIL.ImageTransform.**Transform**(*data*)
Bases: PIL.Image.ImageTransformHandler

getdata()

transform(*size*, *image*, ***options*)

4.26.11 JpegPresets Module

JPEG quality settings equivalent to the Photoshop settings.

More presets can be added to the presets dict if needed.

Can be use when saving JPEG file.

To apply the preset, specify:

```
quality="preset_name"
```

To apply only the quantization table:

```
qtables="preset_name"
```

To apply only the subsampling setting:

```
subsampling="preset_name"
```

Example:

```
im.save("image_name.jpg", quality="web_high")
```

Subsampling

Subsampling is the practice of encoding images by implementing less resolution for chroma information than for luma information. (ref.: http://en.wikipedia.org/wiki/Chroma_subsampling)

Possible subsampling values are 0, 1 and 2 that correspond to 4:4:4, 4:2:2 and 4:1:1 (or 4:2:0?).

You can get the subsampling of a JPEG with the *JpegImagePlugin.get_subsampling(im)* function.

Quantization tables

They are values use by the DCT (Discrete cosine transform) to remove *unnecessary* information from the image (the lossy part of the compression). (ref.: http://en.wikipedia.org/wiki/Quantization_matrix#Quantization_matrices, http://en.wikipedia.org/wiki/JPEG#Quantization)

You can get the quantization tables of a JPEG with:

```
im.quantization
```

This will return a dict with a number of arrays. You can pass this dict directly as the qtables argument when saving a JPEG.

The tables format between im.quantization and quantization in presets differ in 3 ways:

1. The base container of the preset is a list with sublists instead of dict. dict[0] -> list[0], dict[1] -> list[1], ...
2. Each table in a preset is a list instead of an array.
3. The zigzag order is remove in the preset (needed by libjpeg >= 6a).

You can convert the dict format to the preset format with the *JpegImagePlugin.convert_dict_qtables(dict_qtables)* function.

Libjpeg ref.: http://www.jpegcameras.com/libjpeg/libjpeg-3.html

4.26.12 `PaletteFile` Module

class PIL.PaletteFile.**PaletteFile**(*fp*)

getpalette()

rawmode = 'RGB'

4.26.13 `PcfFontFile` Module

class PIL.PcfFontFile.**PcfFontFile**(*fp*)

Bases: *PIL.FontFile.FontFile*

name = 'name'

PIL.PcfFontFile.**sz**(*s*, *o*)

4.26.14 PngImagePlugin.iTXt Class

class PIL.PngImagePlugin.**iTXt**
Bases: str

Subclass of string to allow iTXt chunks to look like strings while keeping their extra information

__new__(*cls*, *text*, *lang*, *tkey*)

Parameters

- **value** – value for this key
- **lang** – language code
- **tkey** – UTF-8 version of the key name

4.26.15 PngImagePlugin.PngInfo Class

class PIL.PngImagePlugin.**PngInfo**
PNG chunk container (for use with save(pnginfo=))

add(*cid*, *data*)
Appends an arbitrary chunk. Use with caution.

Parameters

- **cid** – a byte string, 4 bytes long.
- **data** – a byte string of the encoded data

add_itxt(*key*, *value*, *lang=''*, *tkey=''*, *zip=False*)
Appends an iTXt chunk.

Parameters

- **key** – latin-1 encodable text key name
- **value** – value for this key
- **lang** – language code
- **tkey** – UTF-8 version of the key name
- **zip** – compression flag

add_text(*key*, *value*, *zip=0*)
Appends a text chunk.

Parameters

- **key** – latin-1 encodable text key name
- **value** – value for this key, text or an *PIL.PngImagePlugin.iTXt* instance
- **zip** – compression flag

4.26.16 TarIO Module

class PIL.TarIO.**TarIO**(*tarfile*, *file*)
Bases: *PIL.ContainerIO.ContainerIO*

4.26.17 TiffTags Module

4.26.18 WalImageFile Module

PIL.WalImageFile.**open**(*filename*)

4.26.19 _binary Module

PIL._binary.**i16be**(*c*, *o=0*)

PIL._binary.**i16le**(*c*, *o=0*)
Converts a 2-bytes (16 bits) string to an integer.

c: string containing bytes to convert o: offset of bytes to convert in string

PIL._binary.**i32be**(*c*, *o=0*)

PIL._binary.**i32le**(*c*, *o=0*)
Converts a 4-bytes (32 bits) string to an integer.

c: string containing bytes to convert o: offset of bytes to convert in string

PIL._binary.**i8**(*c*)

PIL._binary.**o16be**(*i*)

PIL._binary.**o16le**(*i*)

PIL._binary.**o32be**(*i*)

PIL._binary.**o32le**(*i*)

PIL._binary.**o8**(*i*)

CHAPTER 5

Appendices

5.1 Image file formats

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

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

5.1.1 Fully supported formats

BMP

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

The `open()` method sets the following `info` properties:

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

EPS

PIL identifies EPS files containing image data, and can read files that contain embedded raster images (ImageData descriptors). If Ghostscript is available, other EPS files can be read as well. The EPS driver can also write EPS images.

If Ghostscript is available, you can call the *`load()`* method with the following parameter to affect how Ghostscript renders the EPS

scale Affects the scale of the resultant rasterized image. If the EPS suggests that the image be rendered at 100px x 100px, setting this parameter to 2 will make the Ghostscript render a 200px x 200px image instead. The relative position of the bounding box is maintained:

```
im = Image.open(...)
im.size #(100,100)
im.load(scale=2)
im.size #(200,200)
```

GIF

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

The `open()` method sets the following `info` properties:

background Default background color (a palette color index).

duration Time between frames in an animation (in milliseconds).

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

version Version (either `GIF87a` or `GIF89a`).

Reading sequences

The GIF loader supports the `seek()` and `tell()` methods. You can seek to the next frame (`im.seek(im.tell() + 1)`), or rewind the file by seeking to the first frame. Random access is not supported.

Reading local images

The GIF loader creates an image memory the same size as the GIF file's *logical screen size*, and pastes the actual pixel data (the *local image*) into this image. If you only want the actual pixel rectangle, you can manipulate the `size` and `tile` attributes before loading the file:

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

if im.tile[0][0] == "gif":
    # only read the first "local image" from this GIF file
    tag, (x0, y0, x1, y1), offset, extra = im.tile[0]
    im.size = (x1 - x0, y1 - y0)
    im.tile = [(tag, (0, 0) + im.size, offset, extra)]
```

IM

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

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

JPEG

PIL reads JPEG, JFIF, and Adobe JPEG files containing `L`, `RGB`, or `CMYK` data. It writes standard and progressive JFIF files.

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

The `open()` method may set the following `info` properties if available:

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

jfif_version A tuple representing the jfif version, (major version, minor version).

jfif_density A tuple representing the pixel density of the image, in units specified by jfif_unit.

jfif_unit Units for the jfif_density:

- 0 - No Units
- 1 - Pixels per Inch
- 2 - Pixels per Centimeter

dpi A tuple representing the reported pixel density in pixels per inch, if the file is a jfif file and the units are in inches.

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

adobe_transform Vendor Specific Tag.

progression Indicates that this is a progressive JPEG file.

icc-profile The ICC color profile for the image.

exif Raw EXIF data from the image.

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

quality The image quality, on a scale from 1 (worst) to 95 (best). The default is 75. Values above 95 should be avoided; 100 disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality.

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

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

dpi A tuple of integers representing the pixel density, `(x,y)`.

icc-profile If present, the image is stored with the provided ICC profile. If this parameter is not provided, the image will be saved with no profile attached. To preserve the existing profile:

```
im.save(filename, 'jpeg', icc_profile=im.info.get('icc_profile'))
```

exif If present, the image will be stored with the provided raw EXIF data.

subsampling If present, sets the subsampling for the encoder.

- `keep`: Only valid for JPEG files, will retain the original image setting.
- `4:4:4`, `4:2:2`, `4:1:1`: Specific sampling values
- `-1`: equivalent to `keep`
- `0`: equivalent to `4:4:4`
- `1`: equivalent to `4:2:2`
- `2`: equivalent to `4:1:1`

qtables If present, sets the qtables for the encoder. This is listed as an advanced option for wizards in the JPEG documentation. Use with caution. `qtables` can be one of several types of values:

- a string, naming a preset, e.g. `keep`, `web_low`, or `web_high`
- a list, tuple, or dictionary (with integer keys = range(len(keys))) of lists of 64 integers. There must be between 2 and 4 tables.

New in version 2.5.0.

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.

JPEG 2000

New in version 2.4.0.

PIL reads and writes JPEG 2000 files containing `L`, `LA`, `RGB` or `RGBA` data. It can also read files containing `YCbCr` data, which it converts on read into `RGB` or `RGBA` depending on whether or not there is an alpha channel. PIL supports JPEG 2000 raw codestreams (`.j2k` files), as well as boxed JPEG 2000 files (`.j2p` or `.jpx` files). PIL does *not* support files whose components have different sampling frequencies.

When loading, if you set the `mode` on the image prior to the *`load()`* method being invoked, you can ask PIL to convert the image to either `RGB` or `RGBA` rather than choosing for itself. It is also possible to set `reduce` to the number of resolutions to discard (each one reduces the size of the resulting image by a factor of 2), and `layers` to specify the number of quality layers to load.

The *`save()`* method supports the following options:

offset The image offset, as a tuple of integers, e.g. (16, 16)

tile_offset The tile offset, again as a 2-tuple of integers.

tile_size The tile size as a 2-tuple. If not specified, or if set to None, the image will be saved without tiling.

quality_mode Either *"rates"* or *"dB"* depending on the units you want to use to specify image quality.

quality_layers A sequence of numbers, each of which represents either an approximate size reduction (if quality mode is *"rates"*) or a signal to noise ratio value in decibels. If not specified, defaults to a single layer of full quality.

num_resolutions The number of different image resolutions to be stored (which corresponds to the number of Discrete Wavelet Transform decompositions plus one).

codeblock_size The code-block size as a 2-tuple. Minimum size is 4 x 4, maximum is 1024 x 1024, with the additional restriction that no code-block may have more than 4096 coefficients (i.e. the product of the two numbers must be no greater than 4096).

precinct_size The precinct size as a 2-tuple. Must be a power of two along both axes, and must be greater than the code-block size.

irreversible If `True`, use the lossy Irreversible Color Transformation followed by DWT 9-7. Defaults to `False`, which means to use the Reversible Color Transformation with DWT 5-3.

progression Controls the progression order; must be one of `"LRCP"`, `"RLCP"`, `"RPCL"`, `"PCRL"`, `"CPRL"`. The letters stand for Component, Position, Resolution and Layer respectively and control the order of encoding, the idea being that e.g. an image encoded using LRCP mode can have its quality layers decoded as they arrive at the decoder, while one encoded using RLCP mode will have increasing resolutions decoded as they arrive, and so on.

cinema_mode Set the encoder to produce output compliant with the digital cinema specifications. The options here are `"no"` (the default), `"cinema2k-24"` for 24fps 2K, `"cinema2k-48"` for 48fps 2K, and `"cinema4k-24"` for 24fps 4K. Note that for compliant 2K files, *at least one* of your image dimensions must match 2048 x 1080, while for compliant 4K files, *at least one* of the dimensions must match 4096 x 2160.

Note: To enable JPEG 2000 support, you need to build and install the OpenJPEG library, version 2.0.0 or higher, before building the Python Imaging Library.

Windows users can install the OpenJPEG binaries available on the OpenJPEG website, but must add them to their PATH in order to use PIL (if you fail to do this, you will get errors about not being able to load the `_imaging` DLL).

MSP

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

PCX

PIL reads and writes PCX files containing `1`, `L`, `P`, or `RGB` data.

PNG

PIL identifies, reads, and writes PNG files containing `1`, `L`, `P`, `RGB`, or `RGBA` data. Interlaced files are supported as of v1.1.7.

The `open()` method sets the following `info` properties, when appropriate:

gamma Gamma, given as a floating point number.

transparency Transparency color index. This key is omitted if the image is not a transparent palette image.

`Open` also sets `Image.text` to a list of the values of the `tEXt`, `zTXt`, and `iTXt` chunks of the PNG image. Individual compressed chunks are limited to a decompressed size of `PngImagePlugin.MAX_TEXT_CHUNK`, by default 1MB, to prevent decompression bombs. Additionally, the total size of all of the text chunks is limited to `PngImagePlugin.MAX_TEXT_MEMORY`, defaulting to 64MB.

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

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

transparency For `P`, `L`, and `RGB` images, this option controls what color image to mark as transparent.

dpi A tuple of two numbers corresponding to the desired dpi in each direction.

pnginfo A *PIL.PngImagePlugin.PngInfo* instance containing text tags.

bits (experimental) For `P` images, this option controls how many bits to store. If omitted, the PNG writer uses 8 bits (256 colors).

dictionary (experimental) Set the ZLIB encoder dictionary.

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

PPM

PIL reads and writes PBM, PGM and PPM files containing `1`, `L` or `RGB` data.

SPIDER

PIL reads and writes SPIDER image files of 32-bit floating point data ("F;32F").

PIL also reads SPIDER stack files containing sequences of SPIDER images. The `seek()` and `tell()` methods are supported, and random access is allowed.

The `open()` method sets the following attributes:

format Set to `SPIDER`

istack Set to 1 if the file is an image stack, else 0.

nimages Set to the number of images in the stack.

A convenience method, `convert2byte()`, is provided for converting floating point data to byte data (mode `L`):

```
im = Image.open('image001.spi').convert2byte()
```

Writing files in SPIDER format

The extension of SPIDER files may be any 3 alphanumeric characters. Therefore the output format must be specified explicitly:

```
im.save('newimage.spi', format='SPIDER')
```

For more information about the SPIDER image processing package, see the SPIDER homepage at Wadsworth Center.

TIFF

PIL reads and writes TIFF files. It can read both striped and tiled images, pixel and plane interleaved multi-band images, and either uncompressed, or Packbits, LZW, or JPEG compressed images.

If you have libtiff and its headers installed, PIL can read and write many more kinds of compressed TIFF files. If not, PIL will always write uncompressed files.

The `open()` method sets the following `info` properties:

compression Compression mode.

dpi Image resolution as an (xdpi, ydpi) tuple, where applicable. You can use the `tag` attribute to get more detailed information about the image resolution.

New in version 1.1.5.

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

Saving Tiff Images

The *`save()`* method can take the following keyword arguments:

tiffinfo A `ImageFileDirectory` object or dict object containing tiff tags and values. The TIFF field type is autodetected for Numeric and string values, any other types require using an `ImageFileDirectory` object and setting the type in `tagtype` with the appropriate numerical value from `TiffTags.TYPES`.

New in version 2.3.0.

compression

A string containing the desired compression method for the file. (valid only with libtiff installed) Valid compression methods are: `[None, "tiff_ccitt", "group3", "group4", "tiff_jpeg", "tiff_adobe_deflate", "tiff_thunderscan", "tiff_deflate", "tiff_sgilog", "tiff_sgilog24", "tiff_raw_16"]`

These arguments to set the tiff header fields are an alternative to using the general tags available through tiffinfo.

description

software

date_time

artist

copyright Strings

resolution_unit A string of "inch", "centimeter" or "cm"

resolution

x_resolution

y_resolution

dpi Either a Float, Integer, or 2 tuple of (numerator, denominator). Resolution implies an equal x and y resolution, dpi also implies a unit of inches.

WebP

PIL reads and writes WebP files. The specifics of PIL's capabilities with this format are currently undocumented.

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

lossless If present, instructs the WEBP writer to use lossless compression.

quality Integer, 1-100, Defaults to 80. Sets the quality level for lossy compression.

icc_procfile The ICC Profile to include in the saved file. Only supported if the system webp library was built with webpmux support.

exif The exif data to include in the saved file. Only supported if the system webp library was built with webpmux support.

XBM

PIL reads and writes X bitmap files (mode `1`).

XV Thumbnails

PIL can read XV thumbnail files.

5.1.2 Read-only formats

CUR

CUR is used to store cursors on Windows. The CUR decoder reads the largest available cursor. 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 DCX decoder can read files containing `1`, `L`, `P`, or `RGB` data.

When the file is opened, only the first image is read. You can use `seek()` or *ImageSequence* to read other images.

FLI, FLC

PIL reads Autodesk FLI and FLC animations.

The `open()` method sets the following `info` properties:

duration The delay (in milliseconds) between each frame.

FPX

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

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

GBR

The GBR decoder reads GIMP brush files.

The `open()` method sets the following `info` properties:

description The brush name.

GD

PIL reads uncompressed GD files. Note that this file format cannot be automatically identified, so you must use *`PIL.GdImageFile.open()`* to read such a file.

The `open()` method sets the following `info` properties:

transparency Transparency color 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.

The *`save()`* method supports the following options:

sizes A list of sizes including in this ico file; these are a 2-tuple, `(width, height)`; Default to `[(16, 16), (24, 24), (32, 32), (48, 48), (64, 64), (128, 128), (255, 255)]`. Any size is bigger then the original size or 255 will be ignored.

ICNS

PIL reads Mac OS X `.icns` files. By default, the largest available icon is read, though you can override this by setting the `size` property before calling *`load()`*. The `open()` method sets the following `info` property:

sizes A list of supported sizes found in this icon file; these are a 3-tuple, `(width, height, scale)`, where `scale` is 2 for a retina icon and 1 for a standard icon. You *are* permitted to use this 3-tuple format for the `size` property if you set it before calling *`load()`*; after loading, the size will be reset to a 2-tuple containing pixel dimensions (so, e.g. if you ask for `(512, 512, 2)`, the final value of `size` will be `(1024, 1024)`).

IMT

PIL reads Image Tools images containing `L` data.

IPTC/NAA

PIL provides limited read support for IPTC/NAA newsphoto files.

MCIDAS

PIL identifies and reads 8-bit McIdas area files.

MIC (read only)

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

MPO

Pillow identifies and reads Multi Picture Object (MPO) files, loading the primary image when first opened. The `seek()` and `tell()` methods may be used to read other pictures from the file. The pictures are zero-indexed and random access is supported.

MIC (read only)

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

PCD

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

PSD

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

SGI

PIL reads uncompressed `L`, `RGB`, and `RGBA` files.

TGA

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

WAL

New in version 1.1.4.

PIL reads Quake2 WAL texture files.

Note that this file format cannot be automatically identified, so you must use the open function in the *WalImageFile* module to read files in this format.

By default, a Quake2 standard palette is attached to the texture. To override the palette, use the putpalette method.

XPM

PIL reads X pixmap files (mode `P`) with 256 colors or less.

The `open()` method sets the following `info` properties:

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

5.1.3 Write-only formats

PALM

PIL provides write-only support for PALM pixmap files.

The format code is `Palm`, the extension is `.palm`.

PDF

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

PIXAR (read only)

PIL provides limited support for PIXAR raster files. The library can identify and read "dumped" RGB files.

The format code is `PIXAR`.

5.1.4 Identify-only formats

BUFR

New in version 1.1.3.

PIL provides a stub driver for BUFR files.

To add read or write support to your application, use `PIL.BufrStubImagePlugin.register_handler()`.

FITS

New in version 1.1.5.

PIL provides a stub driver for FITS files.

To add read or write support to your application, use `PIL.FitsStubImagePlugin.register_handler()`.

GRIB

New in version 1.1.5.

PIL provides a stub driver for GRIB files.

The driver requires the file to start with a GRIB header. If you have files with embedded GRIB data, or files with multiple GRIB fields, your application has to seek to the header before passing the file handle to PIL.

To add read or write support to your application, use `PIL.GribStubImagePlugin.register_handler()`.

HDF5

New in version 1.1.5.

PIL provides a stub driver for HDF5 files.

To add read or write support to your application, use `PIL.Hdf5StubImagePlugin.register_handler()`.

MPEG

PIL identifies MPEG files.

WMF

PIL can identify placable WMF files.

In PIL 1.1.4 and earlier, the WMF driver provides some limited rendering support, but not enough to be useful for any real application.

In PIL 1.1.5 and later, the WMF driver is a stub driver. To add WMF read or write support to your application, use `PIL.WmfImagePlugin.register_handler()` to register a WMF handler.

```
from PIL import Image
from PIL import WmfImagePlugin

class WmfHandler:
    def open(self, im):
        ...
    def load(self, im):
        ...
        return image
    def save(self, im, fp, filename):
        ...

wmf_handler = WmfHandler()

WmfImagePlugin.register_handler(wmf_handler)

im = Image.open("sample.wmf")
```

5.2 Writing your own file decoder

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

unique format name (usually an abbreviation).

Warning: Pillow >= 2.1.0 no longer automatically imports any file in the Python path with a name ending in `ImagePlugin.py`. You will need to import your decoder manually.

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

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

5.2.1 Example

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

SpamImagePlugin.py:

```
from PIL import Image, ImageFile
import string

class SpamImageFile(ImageFile.ImageFile):

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

    def _open(self):

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

        header = string.split(header)

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

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

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

```
Image.register_open("SPAM", SpamImageFile)

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

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

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

5.2.2 The `tile` attribute

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

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

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

The fields are used as follows:

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

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

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

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

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

The `raw` decoder

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

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

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

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

The fields are used as follows:

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

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

orientation

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

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

mode	description
`1`	1-bit bilevel, stored with the leftmost pixel in the most significant bit. 0 means black, 1 means white.
`1;I`	1-bit inverted bilevel, stored with the leftmost pixel in the most significant bit. 0 means white, 1 means black.
`1;R`	1-bit reversed bilevel, stored with the leftmost pixel in the least significant bit. 0 means black, 1 means white.
`L`	8-bit greyscale. 0 means black, 255 means white.
`L;I`	8-bit inverted greyscale. 0 means white, 255 means black.
`P`	8-bit palette-mapped image.
`RGB`	24-bit true colour, stored as (red, green, blue).
`BGR`	24-bit true colour, stored as (blue, green, red).
`RGBX`	24-bit true colour, stored as (blue, green, red, pad).
`RGB;L`	24-bit true colour, line interleaved (first all red pixels, the all green pixels, finally all blue pixels).

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

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

5.2.3 Decoding floating point data

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

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

mode	description
F	32-bit native floating point.
F;8	8-bit unsigned integer.
F;8S	8-bit signed integer.
F;16	16-bit little endian unsigned integer.
F;16S	16-bit little endian signed integer.
F;16B	16-bit big endian unsigned integer.
F;16BS	16-bit big endian signed integer.
F;16N	16-bit native unsigned integer.
F;16NS	16-bit native signed integer.
F;32	32-bit little endian unsigned integer.
F;32S	32-bit little endian signed integer.
F;32B	32-bit big endian unsigned integer.
F;32BS	32-bit big endian signed integer.
F;32N	32-bit native unsigned integer.
F;32NS	32-bit native signed integer.
F;32F	32-bit little endian floating point.
F;32BF	32-bit big endian floating point.
F;32NF	32-bit native floating point.
F;64F	64-bit little endian floating point.
F;64BF	64-bit big endian floating point.
F;64NF	64-bit native floating point.

5.2.4 The bit decoder

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

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

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

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

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

The fields are used as follows:

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

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

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

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

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

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

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

If omitted, the fill order defaults to 0.

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

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

CHAPTER 6

Release Notes

6.1 Pillow 2.7.0

6.1.1 Png text chunk size limits

To prevent potential denial of service attacks using compressed text chunks, there are now limits to the decompressed size of text chunks decoded from PNG images. If the limits are exceeded when opening a PNG image a `ValueError` will be raised.

Individual text chunks are limited to `PIL.PngImagePlugin.MAX_TEXT_CHUNK`, set to 1MB by default. The total decompressed size of all text chunks is limited to `PIL.PngImagePlugin.MAX_TEXT_MEMORY`, which defaults to 64MB. These values can be changed prior to opening PNG images if you know that there are large text blocks that are desired.

6.1.2 Image resizing filters

Image resizing methods *`resize()`* and *`thumbnail()`* take a *resample* argument, which tells which filter should be used for resampling. Possible values are: `PIL.Image.NEAREST`, `PIL.Image.BILINEAR`, `PIL.Image.BICUBIC` and `PIL.Image.ANTIALIAS`. Almost all of them were changed in this version.

Bicubic and bilinear downscaling

From the beginning `BILINEAR` and `BICUBIC` filters were based on affine transformations and used a fixed number of pixels from the source image for every destination pixel (2x2 pixels for `BILINEAR` and 4x4 for `BICUBIC`). This gave an unsatisfactory result for downscaling. At the same time, a high quality convolutions-based algorithm with flexible kernel was used for `ANTIALIAS` filter.

Starting from Pillow 2.7.0, a high quality convolutions-based algorithm is used for all of these three filters.

If you have previously used any tricks to maintain quality when downscaling with `BILINEAR` and `BICUBIC` filters (for example, reducing within several steps), they are unnecessary now.

Antialias renamed to Lanczos

A new `PIL.Image.LANCZOS` constant was added instead of `ANTIALIAS`.

When `ANTIALIAS` was initially added, it was the only high-quality filter based on convolutions. It's name was supposed to reflect this. Starting from Pillow 2.7.0 all resize method are based on convolutions. All of them are antialias from now on. And the real name of the `ANTIALIAS` filter is Lanczos filter.

The ANTIALIAS constant is left for backward compatibility and is an alias for LANCZOS.

Lanczos upscaling quality

The image upscaling quality with LANCZOS filter was almost the same as BILINEAR due to bug. This has been fixed.

Bicubic upscaling quality

The BICUBIC filter for affine transformations produced sharp, slightly pixelated image for upscaling. Bicubic for convolutions is more soft.

Resize performance

In most cases, convolution is more a expensive algorithm for downscaling because it takes into account all the pixels of source image. Therefore BILINEAR and BICUBIC filters' performance can be lower than before. On the other hand the quality of BILINEAR and BICUBIC was close to NEAREST. So if such quality is suitable for your tasks you can switch to NEAREST filter for downscaling, which will give a huge improvement in performance.

At the same time performance of convolution resampling for downscaling has been improved by around a factor of two compared to the previous version. The upscaling performance of the LANCZOS filter has remained the same. For BILINEAR filter it has improved by 1.5 times and for BICUBIC by four times.

Default filter for thumbnails

In Pillow 2.5 the default filter for *thumbnail()* was changed from NEAREST to ANTIALIAS. Antialias was chosen because all the other filters gave poor quality for reduction. Starting from Pillow 2.7.0, ANTIALIAS has been replaced with BICUBIC, because it's faster and ANTIALIAS doesn't give any advantages after downscaling with libjpeg, which uses supersampling internally, not convolutions.

6.1.3 Image transposition

A new method PIL.Image.TRANSPOSE has been added for the *transpose()* operation in addition to FLIP_LEFT_RIGHT, FLIP_TOP_BOTTOM, ROTATE_90, ROTATE_180, ROTATE_270. TRANSPOSE is an algebra transpose, with an image reflected across its main diagonal.

The speed of ROTATE_90, ROTATE_270 and TRANSPOSE has been significantly improved for large images which don't fit in the processor cache.

6.1.4 Gaussian blur and unsharp mask

The *GaussianBlur()* implementation has been replaced with a sequential application of box filters. The new implementation is based on "Theoretical foundations of Gaussian convolution by extended box filtering" from the Mathematical Image Analysis Group. As *UnsharpMask()* implementations use Gaussian blur internally, all changes from this chapter are also applicable to it.

Blur radius

There was an error in the previous version of Pillow, where blur radius (the standard deviation of Gaussian) actually meant blur diameter. For example, to blur an image with actual radius 5 you were forced to use value 10. This has been fixed. Now the meaning of the radius is the same as in other software.

If you used a Gaussian blur with some radius value, you need to divide this value by two.

Blur performance

Box filter computation time is constant relative to the radius and depends on source image size only. Because the new Gaussian blur implementation is based on box filter, its computation time also doesn't depends on the blur radius.

For example, previously, if the execution time for a given test image was 1 second for radius 1, 3.6 seconds for radius 10 and 17 seconds for 50, now blur with any radius on same image is executed for 0.2 seconds.

Blur quality

The previous implementation takes into account only source pixels within 2 * standard deviation radius for every destination pixel. This was not enough, so the quality was worse compared to other Gaussian blur software.

The new implementation does not have this drawback.

6.1.5 TFF Parameter Changes

Several kwarg parameters for saving TIFF images were previously specified as strings with included spaces (e.g. 'x resolution'). This was difficult to use as kwargs without constructing and passing a dictionary. These parameters now use the underscore character instead of space. (e.g. 'x_resolution')

CHAPTER 7

Original PIL README

What follows is the original PIL 1.1.7 README file contents.

```
The Python Imaging Library
$Id$

Release 1.1.7 (November 15, 2009)

====================================================================
The Python Imaging Library 1.1.7
====================================================================

Contents
--------

+ Introduction
+ Support Options
  - Commercial support
  - Free support
+ Software License
+ Build instructions (all platforms)
  - Additional notes for Mac OS X
  - Additional notes for Windows

--------------------------------------------------------------------
Introduction
--------------------------------------------------------------------

The Python Imaging Library (PIL) adds image processing capabilities
to your Python environment.  This library provides extensive file
format support, an efficient internal representation, and powerful
image processing capabilities.

This source kit has been built and tested with Python 2.0 and newer,
on Windows, Mac OS X, and major Unix platforms.  Large parts of the
library also work on 1.5.2 and 1.6.

The main distribution site for this software is:

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

That site also contains information about free and commercial support
options, PIL add-ons, answers to frequently asked questions, and more.
```

```
Development versions (alphas, betas) are available here:

        http://effbot.org/downloads/


The PIL handbook is not included in this distribution; to get the
latest version, check:

        http://www.pythonware.com/library/
        http://effbot.org/books/imagingbook/ (drafts)


For installation and licensing details, see below.


--------------------------------------------------------------------
Support Options
--------------------------------------------------------------------

+ Commercial Support

Secret Labs (PythonWare) offers support contracts for companies using
the Python Imaging Library in commercial applications, and in mission-
critical environments.  The support contract includes technical support,
bug fixes, extensions to the PIL library, sample applications, and more.

For the full story, check:

        http://www.pythonware.com/products/pil/support.htm


+ Free Support

For support and general questions on the Python Imaging Library, send
e-mail to the 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
send your questions to the Python mailing list, python-list@python.org,
or post them to the newsgroup comp.lang.python.  DO NOT SEND SUPPORT
QUESTIONS TO PYTHONWARE ADDRESSES.


--------------------------------------------------------------------
Software License
--------------------------------------------------------------------

The Python Imaging Library is

Copyright (c) 1997-2009 by Secret Labs AB
Copyright (c) 1995-2009 by Fredrik Lundh
```

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

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

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


--------------------------------------------------------------------
Build instructions (all platforms)
--------------------------------------------------------------------

For a list of changes in this release, see the CHANGES document.

0. If you're in a hurry, try this:

        $ tar xvfz Imaging-1.1.7.tar.gz
        $ cd Imaging-1.1.7
        $ python setup.py install

   If you prefer to know what you're doing, read on.


1. Prerequisites.

   If you need any of the features described below, make sure you
   have the necessary libraries before building PIL.

   feature              library
   ----------------------------------------------------------------
   JPEG support         libjpeg (6a or 6b)

                        http://www.ijg.org
                        http://www.ijg.org/files/jpegsrc.v6b.tar.gz
                        ftp://ftp.uu.net/graphics/jpeg/

   PNG support          zlib (1.2.3 or later is recommended)

                        http://www.gzip.org/zlib/

   OpenType/TrueType    freetype2 (2.3.9 or later is recommended)
   support
                        http://www.freetype.org
                        http://freetype.sourceforge.net
```

```
   CMS support          littleCMS (1.1.5 or later is recommended)
   support
                        http://www.littlecms.com/

   If you have a recent Linux version, the libraries provided with the
   operating system usually work just fine.  If some library is
   missing, installing a prebuilt version (jpeg-devel, zlib-devel,
   etc) is usually easier than building from source.  For example, for
   Ubuntu 9.10 (karmic), you can install the following libraries:

       sudo apt-get install libjpeg62-dev
       sudo apt-get install zlib1g-dev
       sudo apt-get install libfreetype6-dev
       sudo apt-get install liblcms1-dev

   If you're using Mac OS X, you can use the 'fink' tool to install
   missing libraries (also see the Mac OS X section below).

   Similar tools are available for many other platforms.


2. To build under Python 1.5.2, you need to install the stand-alone
   version of the distutils library:

       http://www.python.org/sigs/distutils-sig/download.html

   You can fetch distutils 1.0.2 from the Python source repository:

       svn export http://svn.python.org/projects/python/tags/Distutils-1_0_2/Lib/distutils/

   For newer releases, the distutils library is included in the
   Python standard library.

   NOTE: Version 1.1.7 is not fully compatible with 1.5.2.  Some
   more recent additions to the library may not work, but the core
   functionality is available.


3. If you didn't build Python from sources, make sure you have
   Python's build support files on your machine.  If you've down-
   loaded a prebuilt package (e.g. a Linux RPM), you probably
   need additional developer packages.  Look for packages named
   "python-dev", "python-devel", or similar.  For example, for
   Ubuntu 9.10 (karmic), use the following command:

       sudo apt-get install python-dev


4. When you have everything you need, unpack the PIL distribution
   (the file Imaging-1.1.7.tar.gz) in a suitable work directory:

        $ cd MyExtensions # example
        $ gunzip Imaging-1.1.7.tar.gz
        $ tar xvf Imaging-1.1.7.tar


5. Build the library.  We recommend that you do an in-place build,
   and run the self test before installing.
```

```
        $ cd Imaging-1.1.7
        $ python setup.py build_ext -i
        $ python selftest.py

   During the build process, the setup.py will display a summary
   report that lists what external components it found.  The self-
   test will display a similar report, with what external components
   the tests found in the actual build files:

        --------------------------------------------------------------------
        PIL 1.1.7 SETUP SUMMARY
        --------------------------------------------------------------------
        *** TKINTER support not available (Tcl/Tk 8.5 libraries needed)
        --- JPEG support available
        --- ZLIB (PNG/ZIP) support available
        --- FREETYPE support available
        --------------------------------------------------------------------

   Make sure that the optional components you need are included.

   If the build script won't find a given component, you can edit the
   setup.py file and set the appropriate ROOT variable.  For details,
   see instructions in the file.

   If the build script finds the component, but the tests cannot
   identify it, try rebuilding *all* modules:

        $ python setup.py clean
        $ python setup.py build_ext -i


6. If the setup.py and selftest.py commands finish without any
   errors, you're ready to install the library:

        $ python setup.py install

   (depending on how Python has been installed on your machine,
   you might have to log in as a superuser to run the 'install'
   command, or use the 'sudo' command to run 'install'.)


--------------------------------------------------------------------
Additional notes for Mac OS X
--------------------------------------------------------------------

On Mac OS X you will usually install additional software such as
libjpeg or freetype with the "fink" tool, and then it ends up in
"/sw".  If you have installed the libraries elsewhere, you may have
to tweak the "setup.py" file before building.


--------------------------------------------------------------------
Additional notes for Windows
--------------------------------------------------------------------

On Windows, you need to tweak the ROOT settings in the "setup.py"
file, to make it find the external libraries.  See comments in the
file for details.
```

```
Make sure to build PIL and the external libraries with the same
runtime linking options as was used for the Python interpreter
(usually /MD, under Visual Studio).


Note that most Python distributions for Windows include libraries
compiled for Microsoft Visual Studio.  You can get the free Express
edition of Visual Studio from:

    http://www.microsoft.com/Express/

To build extensions using other tool chains, see the "Using
non-Microsoft compilers on Windows" section in the distutils handbook:

    http://www.python.org/doc/current/inst/non-ms-compilers.html

For additional information on how to build extensions using the
popular MinGW compiler, see:

    http://mingw.org (compiler)
    http://sebsauvage.net/python/mingw.html (build instructions)
    http://sourceforge.net/projects/gnuwin32 (prebuilt libraries)
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X