# CONTENTS

1 Overview  
   1.1 Installation ..................................................... 3  
   1.2 Handbook ....................................................... 12  
   1.3 Reference ....................................................... 49  
   1.4 Porting .......................................................... 159  
   1.5 About ............................................................. 160  
   1.6 Release Notes .................................................... 160  
   1.7 Deprecations and removals ................................. 201  

2 Indices and tables ............................................. 207

Python Module Index ............................................... 209

Index ............................................................... 211
Pillow is the friendly PIL fork by Alex Clark and Contributors. PIL is the Python Imaging Library by Fredrik Lundh and Contributors.

Pillow for enterprise is available via the Tidelift Subscription. Learn more.
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.

1.1 Installation

1.1.1 Warnings

**Warning:** Pillow and PIL cannot co-exist in the same environment. Before installing Pillow, please uninstall PIL.  

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

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

1.1.2 Python Support

Pillow supports these Python versions.

<table>
<thead>
<tr>
<th>Python</th>
<th>3.9</th>
<th>3.8</th>
<th>3.7</th>
<th>3.6</th>
<th>3.5</th>
<th>3.4</th>
<th>3.3</th>
<th>3.2</th>
<th>2.7</th>
<th>2.6</th>
<th>2.5</th>
<th>2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pillow &gt;= 8.0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 7.0 - 7.2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 6.2.1 - 6.2.2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 6.0 - 6.2.0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 5.2 - 5.4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 5.0 - 5.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow 2 - 3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pillow &lt; 2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
1.1.3 Basic Installation

**Note:** The following instructions will install Pillow with support for most common image formats. See *External Libraries* for a full list of external libraries supported.

Install Pillow with **pip**:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

**Windows Installation**

We provide Pillow binaries for Windows compiled for the matrix of supported Pythons in both 32 and 64-bit versions in the wheel format. These binaries have all of the optional libraries included except for raqm, libimagequant, and libxcb:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

To install Pillow in MSYS2, see *Building on Windows using MSYS2/MinGW*.

**macOS Installation**

We provide binaries for macOS for each of the supported Python versions in the wheel format. These include support for all optional libraries except libimagequant and libxcb. Raqm support requires libraqm, fribidi, and harfbuzz to be installed separately:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

**Linux Installation**

We provide binaries for Linux for each of the supported Python versions in the manylinux wheel format. These include support for all optional libraries except libimagequant. Raqm support requires libraqm, fribidi, and harfbuzz to be installed separately:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

Most major Linux distributions, including Fedora, Debian/Ubuntu and ArchLinux also include Pillow in packages that previously contained PIL e.g. *python-imaging*. 
**FreeBSD Installation**

Pillow can be installed on FreeBSD via the official Ports or Packages systems:

**Ports:**

```
cd /usr/ports/graphics/py-pillow && make install clean
```

**Packages:**

```
pkg install py36-pillow
```

**Note:** The Pillow FreeBSD port and packages are tested by the ports team with all supported FreeBSD versions.

### 1.1.4 Building From Source

Download and extract the compressed archive from PyPI.

**External Libraries**

**Note:** You do not need to install all supported external libraries to use Pillow’s basic features. **Zlib** and **libjpeg** are required by default.

**Note:** There are Dockerfiles in our Docker images repo to install the dependencies for some operating systems.

Many of Pillow’s features require external libraries:

- **libjpeg** provides JPEG functionality.
  - Pillow has been tested with libjpeg versions 6b, 8, 9-9d and libjpeg-turbo version 8.
  - Starting with Pillow 3.0.0, libjpeg is required by default, but may be disabled with the `--disable-jpeg` flag.

- **zlib** provides access to compressed PNGs
  - Starting with Pillow 3.0.0, zlib is required by default, but may be disabled with the `--disable-zlib` flag.

- **libtiff** provides compressed TIFF functionality
  - Pillow has been tested with libtiff versions 3.x and 4.0-4.1

- **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 2.7-2.11.

- **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 above 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, 2.1.0** and **2.3.1**.
  - Pillow does not support the earlier **1.5** series which ships with Debian Jessie.

• **libimagequant** provides improved color quantization
  - Pillow has been tested with libimagequant **2.6-2.12.6**
  - Libimagequant is licensed GPLv3, which is more restrictive than the Pillow license, therefore we will not be distributing binaries with libimagequant support enabled.

• **libraqm** provides complex text layout support.
  - libraqm provides bidirectional text support (using FriBiDi), shaping (using HarfBuzz), and proper script itemization. As a result, Raqm can support most writing systems covered by Unicode.
  - libraqm depends on the following libraries: FreeType, HarfBuzz, FriBiDi, make sure that you install them before installing libraqm if not available as package in your system.
  - setting text direction or font features is not supported without libraqm.
  - libraqm is dynamically loaded in Pillow 5.0.0 and above, so support is available if all the libraries are installed.
  - Windows support: Raqm is not included in prebuilt wheels

• **libxcb** provides X11 screengrab support.

Once you have installed the prerequisites, run:

```bash
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade 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:

```bash
CFLAGS="-I/usr/pkg/include" python3 -m pip install --upgrade Pillow
```

If Pillow has been previously built without the required prerequisites, it may be necessary to manually clear the pip cache or build without cache using the `--no-cache-dir` option to force a build with newly installed external libraries.

### Build Options

- **Environment variable**: `MAX_CONCURRENCY=n`. Pillow can use multiprocessing to build the extension. Setting `MAX_CONCURRENCY` sets the number of CPUs to use, or can disable parallel building by using a setting of 1. By default, it uses 4 CPUs, or if 4 are not available, as many as are present.

- **Build flags**: `--disable-zlib`, `--disable-jpeg`, `--disable-tiff`, `--disable-freetype`, `--disable-lcms`, `--disable-webp`, `--disable-webpmux`, `--disable-jpeg2000`, `--disable-imagequant`, `--disable-xcb`. 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-lcms`, `--enable-webp`, `--enable-webpmux`, `--enable-jpeg2000`, `--enable-imagequant`, `--enable-xcb`. 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.

- **Build flag: --disable-platform-guessing.** Skips all of the platform dependent guessing of include and library directories for automated build systems that configure the proper paths in the environment variables (e.g. Buildroot).
- **Build flag: --debug.** Adds a debugging flag to the include and library search process to dump all paths searched for and found to stdout.

Sample usage:

```
MAX_CONCURRENCY=1 python3 setup.py build_ext --enable-[feature] install
```

or using pip:

```
python3 -m pip install --upgrade Pillow --global-option="build_ext" --global-option="-\-\-enable-[feature]"
```

### Building on macOS

The Xcode command line tools are required to compile portions of Pillow. The tools are installed by running `xcode-select --install` from the command line. The command line tools are required even if you have the full Xcode package installed. It may be necessary to run `sudo xcodebuild -license` to accept the license prior to using the tools.

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

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

To install libraqm on macOS use Homebrew to install its dependencies:

```
brew install freetype harfbuzz fribidi
```

Then see `depends/install_raqm_cmake.sh` to install libraqm.

Now install Pillow with:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

or from within the uncompressed source directory:

```
python3 setup.py install
```

### Building on Windows

We recommend you use prebuilt wheels from PyPI. If you wish to compile Pillow manually, you can use the build scripts in the `winbuild` directory used for CI testing and development. These scripts require Visual Studio 2017 or newer and NASM.
Building on Windows using MSYS2/MinGW

To build Pillow using MSYS2, make sure you run the MSYS2 MinGW 32-bit or MSYS2 MinGW 64-bit console, not MSYS2 directly.

The following instructions target the 64-bit build, for 32-bit replace all occurrences of `mingw-w64-x86_64-` with `mingw-w64-i686-`.

Make sure you have Python and GCC installed:

```bash
pacman -S \
  mingw-w64-x86_64-gcc \
  mingw-w64-x86_64-python3 \
  mingw-w64-x86_64-python3-pip \
  mingw-w64-x86_64-python3-setuptools
```

Prerequisites are installed on MSYS2 MinGW 64-bit with:

```bash
pacman -S \
  mingw-w64-x86_64-libjpeg-turbo \
  mingw-w64-x86_64-zlib \
  mingw-w64-x86_64-libtiff \
  mingw-w64-x86_64-freetype \
  mingw-w64-x86_64-lcms2 \
  mingw-w64-x86_64-libwebp \
  mingw-w64-x86_64-openjpeg2 \
  mingw-w64-x86_64-libimagequant \
  mingw-w64-x86_64-libraqm
```

Now install Pillow with:

```bash
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

Building on FreeBSD

**Note:** Only FreeBSD 10 and 11 tested

Make sure you have Python’s development libraries installed:

```bash
sudo pkg install python3
```

Prerequisites are installed on FreeBSD 10 or 11 with:

```bash
sudo pkg install jpeg-turbo tiff webp lcms2 freetype2 openjpeg harfbuzz fribidi libxcb
```

Then see `depends/install_raqm_cmake.sh` to install libraqm.
Building on Linux

If you didn’t build Python from source, make sure you have Python’s development libraries installed.

In Debian or Ubuntu:

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

In Fedora, the command is:

```bash
sudo dnf install python3-devel redhat-rpm-config
```

**Note:** redhat-rpm-config is required on Fedora 23, but not earlier versions.

Prerequisites for **Ubuntu 16.04 LTS - 20.04 LTS** are installed with:

```bash
sudo apt-get install libtiff5-dev libjpeg8-dev libopenjp2-7-dev zlib-devel \ 
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev python3-tk \ 
    libharfuzz-dev libfribidi-dev libxcb1-devel
```

Then see `depends/install_raqm.sh` to install libraqm.

Prerequisites are installed on recent **Red Hat**, **CentOS** or **Fedora** with:

```bash
sudo dnf install libtiff-devel libjpeg-devel openjpeg2-devel zlib-devel \ 
    freetype-devel lcms2-devel libwebp-devel tcl-devel tk-devel \ 
    harfbuzz-devel fribidi-devel libraqm-devel libimagequant-devel libxcb-devel
```

Note that the package manager may be yum or DNF, depending on the exact distribution.

See also the Dockerfiles in the Test Infrastructure repo (https://github.com/python-pillow/docker-images) for a known working install process for other tested distros.

Building on Android

Basic Android support has been added for compilation within the Termux environment. The dependencies can be installed by:

```bash
pkg install -y python ndk-sysroot clang make \ 
    libjpeg-turbo
```

This has been tested within the Termux app on ChromeOS, on x86.

1.1.5 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, macOS, and Windows.
Continuous Integration Targets

These platforms are built and tested for every change.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Tested Python versions</th>
<th>Tested architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpine</td>
<td>3.8</td>
<td>x86-64</td>
</tr>
<tr>
<td>Arch</td>
<td>3.8</td>
<td>x86-64</td>
</tr>
<tr>
<td>Amazon Linux 1</td>
<td>3.6</td>
<td>x86-64</td>
</tr>
<tr>
<td>Amazon Linux 2</td>
<td>3.7</td>
<td>x86-64</td>
</tr>
<tr>
<td>CentOS 6</td>
<td>3.6</td>
<td>x86-64</td>
</tr>
<tr>
<td>CentOS 7</td>
<td>3.6</td>
<td>x86-64</td>
</tr>
<tr>
<td>CentOS 8</td>
<td>3.6</td>
<td>x86-64</td>
</tr>
<tr>
<td>Debian 10 Buster</td>
<td>3.7</td>
<td>x86</td>
</tr>
<tr>
<td>Fedora 31</td>
<td>3.7</td>
<td>x86-64</td>
</tr>
<tr>
<td>Fedora 32</td>
<td>3.8</td>
<td>x86-64</td>
</tr>
<tr>
<td>macOS 10.15 Catalina</td>
<td>3.6, 3.7, 3.8, PyPy3</td>
<td>x86-64</td>
</tr>
<tr>
<td>Ubuntu Linux 16.04 LTS (Xenial)</td>
<td>3.6, 3.7, 3.8, PyPy3</td>
<td>x86-64</td>
</tr>
<tr>
<td>Ubuntu Linux 18.04 LTS (Bionic)</td>
<td>3.6, 3.7, 3.8, PyPy3</td>
<td>x86-64</td>
</tr>
<tr>
<td>Ubuntu Linux 20.04 LTS (Focal)</td>
<td>3.8</td>
<td>x86-64</td>
</tr>
<tr>
<td>Windows Server 2016</td>
<td>3.8</td>
<td>x86</td>
</tr>
<tr>
<td></td>
<td>3.6</td>
<td>x86-64</td>
</tr>
<tr>
<td>Windows Server 2019</td>
<td>3.6, 3.7, 3.8</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td></td>
<td>PyPy3</td>
<td>x86</td>
</tr>
<tr>
<td></td>
<td>3.8/MinGW</td>
<td>x86, x86-64</td>
</tr>
</tbody>
</table>

Other Platforms

These platforms have been reported to work at the versions mentioned.

Note: Contributors please test Pillow on your platform then update this document and send a pull request.
### Operating system | Tested Python versions | Latest tested Pillow version | Tested processors |
---|---|---|---|
macOS 10.15 Catalina | 3.5, 3.6, 3.7, 3.8 | 7.2.0 | x86-64 |
macOS 10.14 Mojave | 3.5, 3.6, 3.7, 3.8 | 7.2.0 | x86-64 |
macOS 10.13 High Sierra | 2.7, 3.4, 3.5, 3.6 | 4.2.1 | x86-64 |
macOS 10.12 Sierra | 2.7, 3.4, 3.5, 3.6 | 4.1.1 | x86-64 |
Mac OS X 10.11 El Capitan | 2.7, 3.4, 3.5, 3.6, 3.7 | 5.4.1 | x86-64 |
Mac OS X 10.9 Mavericks | 2.7, 3.2, 3.3, 3.4 | 3.0.0 | x86-64 |
Mac OS X 10.8 Mountain Lion | 2.6, 2.7, 3.2, 3.3 | 3.1.0 | x86-64 |
Redhat Linux 6 | 2.6 | | x86 |
CentOS 6.3 | 2.7, 3.3 | | x86 |
Fedora 23 | 2.7, 3.4 | 3.1.0 | x86-64 |
Ubuntu Linux 12.04 LTS (Precise) | 2.6, 3.2, 3.3, 3.4, 3.5 PyPy5.3.1, PyPy3 v2.4.0 | 3.4.1 | x86,x86-64 |
Ubuntu Linux 10.04 LTS (Lucid) | 2.6 | 2.3.0 | x86,x86-64 |
Debian 8.2 Jessie | 2.7, 3.4 | 3.1.0 | x86-64 |
Raspbian Jessie | 2.7, 3.4 | 3.1.0 | arm |
Raspbian Stretch | 2.7, 3.5 | 4.0.0 | arm |
Gentoo Linux | 2.7, 3.2 | 2.1.0 | x86-64 |
FreeBSD 11.1 | 2.7, 3.4, 3.5, 3.6 | 4.3.0 | x86-64 |
FreeBSD 10.3 | 2.7, 3.4, 3.5 | 4.2.0 | x86-64 |
FreeBSD 10.2 | 2.7, 3.4 | 3.1.0 | x86-64 |
Windows 10 | 3.7 | 7.1.0 | x86-64 |
Windows 8.1 Pro | 2.6, 2.7, 3.2, 3.3, 3.4 | 2.4.0 | x86,x86-64 |
Windows 8 Pro | 2.6, 2.7, 3.2, 3.3, 3.4a3 | 2.2.0 | x86,x86-64 |
Windows 7 Professional | 3.7 | 7.0.0 | x86,x86-64 |
Windows Server 2008 R2 Enterprise | 3.3 | | x86-64 |

### 1.1.6 Old Versions

You can download old distributions from the release history at PyPI and by direct URL access eg. [https://pypi.org/project/Pillow/1.0/](https://pypi.org/project/Pillow/1.0/).
1.2 Handbook

1.2.1 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.

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.

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.

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.

1.2.2 Tutorial

Using the Image class

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

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

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

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

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

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

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

## Reading and writing images

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

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

### Convert files to JPEG

```python
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:
            with Image.open(infile) as im:
                im.save(outfile)
        except OSError:
            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

```python
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:
            with Image.open(infile) as im:
                im.thumbnail(size)
                im.save(outfile, "JPEG")
        except OSError:
            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

import sys
from PIL import Image

for infile in sys.argv[1:]:
    try:
        with Image.open(infile) as im:
            print(infile, im.format, f"{im.size}x{im.mode}")
    except OSError:
        pass
```

Cutting, pasting, and merging images

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

**Copying a subrectangle from an image**

```python
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

```python
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

```python
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(part1, (xsize-delta, 0, xsize, ysize))
    image.paste(part2, (0, 0, xsize-delta, 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. For example, pasting an RGBA image and also using it as the mask would paste the opaque portion of the image but not its transparent background.

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

Splitting and merging bands

```python
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.
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

```python
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

```python
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)
```

`transpose(ROTATE)` operations can also be performed identically with `rotate()` operations, provided the `expand` flag is true, to provide for the same changes to the image’s size.

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

Color transforms

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

Converting between modes

```python
from PIL import Image
with Image.open("hopper.ppm") as im:
    im = im.convert("L")
```

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

Image enhancement

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

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

Applying filters

```python
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

```python
# 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

```python
# 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:

```python
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

```python
from PIL import ImageEnhance

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

Image sequences

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

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

Reading sequences

```python
from PIL import Image

with Image.open("animation.gif") as im:
    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.

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

Using the `ImageSequence` Iterator class

```python
from PIL import ImageSequence

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

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

Drawing PostScript

```python
from PIL import Image
from PIL import PSDraw

with Image.open("hopper.ppm") as im:
    title = "hopper"
    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()
```

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. `Image.open()` can be used as a context manager:

```python
from PIL import Image
with Image.open("hopper.ppm") as im:
    ...
```

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

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

Reading from an open file

```python
from PIL import Image
with open("hopper.ppm", "rb") as fp:
    im = Image.open(fp)
```

To read an image from binary data, use the `BytesIO` class:
Reading from binary data

```python
from PIL import Image, io
im = Image.open(io.BytesIO(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

```python
from PIL import Image, TarIO
fp = TarIO.TarIO("Tests/images/hopper.tar", "hopper.jpg")
im = Image.open(fp)
```

Controlling the decoder

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

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

Reading in draft mode

This is only available for JPEG and MPO files.

```python
from PIL import Image

with Image.open(file) as im:
    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.
1.2.3 Concepts

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

Bands

An image can consist of one or more bands of data. The Python Imaging Library allows you to store several bands in a single image, provided they all have the same dimensions and depth. For example, a PNG image might have ‘R’, ‘G’, ‘B’, and ‘A’ bands for the red, green, blue, and alpha transparency values. Many operations act on each band separately, e.g., histograms. It is often useful to think of each pixel as having one value per band.

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

Modes

The mode of an image is a string which defines the type and depth of a pixel in the image. Each pixel uses the full range of the bit depth. So a 1-bit pixel has a range of 0-1, an 8-bit pixel has a range of 0-255 and so on. 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)
  - Note that this refers to the JPEG, and not the ITU-R BT.2020, standard
- 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)

Pillow also provides limited support for a few special modes, including:

- LA (L with alpha)
- PA (P with alpha)
- RGBX (true color with padding)
- RGBA (true color with premultiplied alpha)
- La (L with premultiplied alpha)
- I;16 (16-bit unsigned integer pixels)
- I;16L (16-bit little endian unsigned integer pixels)
- I;16B (16-bit big endian unsigned integer pixels)
- I;16N (16-bit native endian unsigned integer pixels)
- BGR;15 (15-bit reversed true colour)
• BGR;16 (16-bit reversed true colour)
• BGR;24 (24-bit reversed true colour)
• BGR;32 (32-bit reversed true colour)

However, Pillow doesn’t support user-defined modes; if you need 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.

### Size

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

### Coordinate System

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

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

### Palette

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

### Info

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

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

### Orientation

A common element of the `info` attribute for JPG and TIFF images is the EXIF orientation tag. This is an instruction for how the image data should be oriented. For example, it may instruct an image to be rotated by 90 degrees, or to be mirrored. To apply this information to an image, `exif_transpose()` can be used.

### Filters

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

- **PIL.Image.NEAREST**
  Pick one nearest pixel from the input image. Ignore all other input pixels.
PIL.Image.BOX
Each pixel of source image contributes to one pixel of the destination image with identical weights. For upscaling is equivalent of NEAREST. This filter can only be used with the resize() and thumbnail() methods.
New in version 3.4.0.

PIL.Image.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.

PIL.Image.HAMMING
Produces a sharper image than BILINEAR, doesn’t have dislocations on local level like with BOX. This filter can only be used with the resize() and thumbnail() methods.
New in version 3.4.0.

PIL.Image.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.

PIL.Image.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. This filter can only be used with the resize() and thumbnail() methods.
New in version 1.1.3.

Filters comparison table

<table>
<thead>
<tr>
<th>Filter</th>
<th>Downscaling quality</th>
<th>Upscaling quality</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEAREST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BILINEAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAMMING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BICUBIC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LANCZOS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.2.4 Appendices

Note: Contributors please include appendices as needed or appropriate with your bug fixes, feature additions and tests.

Image file formats

The Python Imaging Library supports a wide variety of raster file formats. Over 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.
Fully supported formats

Contents

• Image file formats
  – Fully supported formats
    • BMP
    • DIB
    • EPS
    • GIF
      • Reading sequences
      • Saving
      • Reading local images
    • ICNS
    • ICO
    • IM
    • JPEG
    • JPEG 2000
    • MSP
    • PCX
    • PNG
      • APNG sequences
      • Saving
    • PPM
    • SG1
    • SPIDER
      • Writing files in SPIDER format
    • TGA
    • TIFF
      • Reading Multi-frame TIFF Images
      • Saving Tiff Images
    • WebP
      • Saving sequences
    • XBM
  – Read-only formats
    • BLP
<table>
<thead>
<tr>
<th>Formats</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUR</td>
<td>Write-only</td>
</tr>
<tr>
<td>DCX</td>
<td>Write-only</td>
</tr>
<tr>
<td>DDS</td>
<td>Write-only</td>
</tr>
<tr>
<td>FLI, FLC</td>
<td>Write-only</td>
</tr>
<tr>
<td>FPX</td>
<td>Write-only</td>
</tr>
<tr>
<td>FTEX</td>
<td>Write-only</td>
</tr>
<tr>
<td>GBR</td>
<td>Write-only</td>
</tr>
<tr>
<td>GD</td>
<td>Write-only</td>
</tr>
<tr>
<td>IMT</td>
<td>Write-only</td>
</tr>
<tr>
<td>IPTC/NAA</td>
<td>Identify-only</td>
</tr>
<tr>
<td>MCIDAS</td>
<td>Identify-only</td>
</tr>
<tr>
<td>MIC</td>
<td>Identify-only</td>
</tr>
<tr>
<td>MPO</td>
<td>Identify-only</td>
</tr>
<tr>
<td>PCD</td>
<td>Identify-only</td>
</tr>
<tr>
<td>PIXAR</td>
<td>Identify-only</td>
</tr>
<tr>
<td>PSD</td>
<td>Identify-only</td>
</tr>
<tr>
<td>WAL</td>
<td>Identify-only</td>
</tr>
<tr>
<td>WMF</td>
<td>Identify-only</td>
</tr>
<tr>
<td>XPM</td>
<td>Identify-only</td>
</tr>
<tr>
<td>BUFR</td>
<td>Identify-only</td>
</tr>
<tr>
<td>FITS</td>
<td>Identify-only</td>
</tr>
<tr>
<td>GRIB</td>
<td>Identify-only</td>
</tr>
<tr>
<td>HDF5</td>
<td>Identify-only</td>
</tr>
<tr>
<td>MPEG</td>
<td>Identify-only</td>
</tr>
</tbody>
</table>

Write-only formats include CUR, DCX, DDS, FLI, FLC, FPX, FTEX, GBR, GD, IMT, IPTC/NAA, MCIDAS, MIC, MPO, PCD, PIXAR, PSD, WAL, WMF, and XPM.

Identify-only formats include BUFR, FITS, GRIB, HDF5, and MPEG.
BMP

Pillow 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.

DIB

Pillow reads and writes DIB files. DIB files are similar to BMP files, so see above for more information.

New in version 6.0.0.

EPS

Pillow 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. The EPS driver can read EPS images in L, LAB, RGB and CMYK mode, but Ghostscript may convert the images to RGB mode rather than leaving them in the original color space. The EPS driver can write images in L, RGB and CMYK modes.

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:

  ```python
  im = Image.open(...) 
  im.size  #(100,100) 
  im.load(scale=2) 
  im.size  #(200,200)
  ```

GIF

Pillow reads GIF87a and GIF89a versions of the GIF file format. The library writes run-length encoded files in GIF87a by default, unless GIF89a features are used or GIF89a is already in use.

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).
- **transparency**: Transparency color index. This key is omitted if the image is not transparent.
- **version**: Version (either GIF87a or GIF89a).
- **duration**: May not be present. The time to display the current frame of the GIF, in milliseconds.
- **loop**: May not be present. The number of times the GIF should loop. 0 means that it will loop forever.
- **comment**: May not be present. A comment about the image.
- **extension**: May not be present. Contains application specific information.
**Reading sequences**

The GIF loader supports the `seek()` and `tell()` methods. You can combine these methods to seek to the next frame (`im.seek(im.tell() + 1)`).

`im.seek()` raises an `EOFError` if you try to seek after the last frame.

**Saving**

When calling `save()` to write a GIF file, the following options are available:

```python
im.save(out, save_all=True, append_images=[im1, im2, ...])
```

- **save_all** If present and true, all frames of the image will be saved. If not, then only the first frame of a multiframe image will be saved.
- **append_images** A list of images to append as additional frames. Each of the images in the list can be single or multiframe images. This is currently supported for GIF, PDF, TIFF, and WebP.
  
  It is also supported for ICNS. If images are passed in of relevant sizes, they will be used instead of scaling down the main image.
- **include_color_table** Whether or not to include local color table.
- **interlace** Whether or not the image is interlaced. By default, it is, unless the image is less than 16 pixels in width or height.
- **disposal** Indicates the way in which the graphic is to be treated after being displayed.
  
  - 0 - No disposal specified.
  - 1 - Do not dispose.
  - 2 - Restore to background color.
  - 3 - Restore to previous content.
  
  Pass a single integer for a constant disposal, or a list or tuple to set the disposal for each frame separately.
- **palette** Use the specified palette for the saved image. The palette should be a bytes or bytearray object containing the palette entries in RGBRGB... form. It should be no more than 768 bytes. Alternately, the palette can be passed in as an `PIL.ImagePalette.ImagePalette` object.
- **optimize** If present and true, attempt to compress the palette by eliminating unused colors. This is only useful if the palette can be compressed to the next smaller power of 2 elements.

Note that if the image you are saving comes from an existing GIF, it may have the following properties in its `info` dictionary. For these options, if you do not pass them in, they will default to their `info` values.

- **transparency** Transparency color index.
- **duration** The display duration of each frame of the multiframe gif, in milliseconds. Pass a single integer for a constant duration, or a list or tuple to set the duration for each frame separately.
- **loop** Integer number of times the GIF should loop. 0 means that it will loop forever. By default, the image will not loop.
- **comment** A comment about the image.
**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:

```python
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)]
```

**ICNS**

Pillow reads and (macOS only) writes macOS `.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)`).

The `save()` method can take the following keyword arguments:

- **append_images**: A list of images to replace the scaled down versions of the image. The order of the images does not matter, as their use is determined by the size of each image.
  
  New in version 5.1.0.

**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), (256, 256)]`. Any sizes bigger than the original size or 256 will be ignored.

**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 Pillow formats.
JPEG

Pillow 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.

By default Pillow doesn’t allow loading of truncated JPEG files, set `ImageFile.LOAD_TRUNCATED_IMAGES` to override this.

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.
- **comment**  A comment about the image.

  New in version 7.1.0.

The `save()` method supports the following options:

- **quality**  The image quality, on a scale from 0 (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 and true, indicates that the encoder should make an extra pass over the image in order to select optimal encoder settings.
- **progressive**  If present and true, 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 and true, 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:

  ```python
  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, 4:2:0: Specific sampling values

-1: equivalent to keep
0: equivalent to 4:4:4
1: equivalent to 4:2:2
2: equivalent to 4:2:0

qttables 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.

Pillow 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. Pillow supports JPEG 2000 raw codestreams (.j2k files), as well as boxed JPEG 2000 files (.j2p or .jpx files). Pillow 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 Pillow 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 Pillow (if you fail to do this, you will get errors about not being able to load the _imaging DLL).

MSP

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

PCX

Pillow reads and writes PCX files containing 1, L, P, or RGB data.

PNG

Pillow identifies, reads, and writes PNG files containing 1, L, LA, I, P, RGB or RGBA data. Interlaced files are supported as of v1.1.7.

As of Pillow 6.0, EXIF data can be read from PNG images. However, unlike other image formats, EXIF data is not guaranteed to be present in info until load() has been called.

By default Pillow doesn’t allow loading of truncated PNG files, set ImageFile.LOAD_TRUNCATED_IMAGES to override this.

The open() function sets the following info properties, when appropriate:

chromaticity The chromaticity points, as an 8 tuple of floats. (White Point X,White Point Y,Red X,Red Y,Green X,Green Y,Blue X,Blue Y)

gamma Gamma, given as a floating point number.

srgb The sRGB rendering intent as an integer.

• 0 Perceptual
• 1 Relative Colorimetric
• 2 Saturation
• 3 Absolute Colorimetric

**transparency** For P images: Either the palette index for full transparent pixels, or a byte string with alpha values for each palette entry.

For L, I and RGB images, the color that represents full transparent pixels in this image.

This key is omitted if the image is not a transparent palette image.

open also sets `Image.text` to a dictionary 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 and true, 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, I, and RGB images, this option controls what color from the image to mark as transparent.

For P images, this can be a either the palette index, or a byte string with alpha values for each palette entry.

**dpi** A tuple of two numbers corresponding to the desired dpi in each direction.

**pnginfo** A `PIL.PngImagePlugin.PngInfo` instance containing text tags.

**compress_level** ZLIB compression level, a number between 0 and 9: 1 gives best speed, 9 gives best compression, 0 gives no compression at all. Default is 6. When optimize option is True compress_level has no effect (it is set to 9 regardless of a value passed).

**icc_profile** The ICC Profile to include in the saved file.

**exif** The exif data to include in the saved file.

New in version 6.0.0.

**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 installation documentation for details.

---

## APNG sequences

The PNG loader includes limited support for reading and writing Animated Portable Network Graphics (APNG) files. When an APNG file is loaded, `get_format_mimetype()` will return "image/apng". The value of the `is_animated` property will be True when the `n_frames` property is greater than 1. For APNG files, the `n_frames` property depends on both the animation frame count as well as the presence or absence of a default image. See the `default_image` property documentation below for more details. The `seek()` and `tell()` methods are supported.

`im.seek()` raises an `EOFError` if you try to seek after the last frame.

These `info` properties will be set for APNG frames, where applicable:

**default_image** Specifies whether or not this APNG file contains a separate default image, which is not a part of the actual APNG animation.
When an APNG file contains a default image, the initially loaded image (i.e. the result of `seek(0)`) will be the default image. To account for the presence of the default image, the `n_frames` property will be set to `frame_count + 1`, where `frame_count` is the actual APNG animation frame count. To load the first APNG animation frame, `seek(1)` must be called.

- **True** - The APNG contains default image, which is not an animation frame.
- **False** - The APNG does not contain a default image. The `n_frames` property will be set to the actual APNG animation frame count. The initially loaded image (i.e. `seek(0)`) will be the first APNG animation frame.

**loop** The number of times to loop this APNG, 0 indicates infinite looping.

**duration** The time to display this APNG frame (in milliseconds).

**Note:** The APNG loader returns images the same size as the APNG file’s logical screen size. The returned image contains the pixel data for a given frame, after applying any APNG frame disposal and frame blend operations (i.e. it contains what a web browser would render for this frame - the composite of all previous frames and this frame).

Any APNG file containing sequence errors is treated as an invalid image. The APNG loader will not attempt to repair and reorder files containing sequence errors.

### Saving

When calling `save()`, by default only a single frame PNG file will be saved. To save an APNG file (including a single frame APNG), the `save_all` parameter must be set to `True`. The following parameters can also be set:

- **default_image** Boolean value, specifying whether or not the base image is a default image. If `True`, the base image will be used as the default image, and the first image from the `append_images` sequence will be the first APNG animation frame. If `False`, the base image will be used as the first APNG animation frame. Defaults to `False`.

- **append_images** A list or tuple of images to append as additional frames. Each of the images in the list can be single or multiframe images. The size of each frame should match the size of the base image. Also note that if a frame’s mode does not match that of the base image, the frame will be converted to the base image mode.

- **loop** Integer number of times to loop this APNG, 0 indicates infinite looping. Defaults to 0.

- **duration** Integer (or list or tuple of integers) length of time to display this APNG frame (in milliseconds). Defaults to 0.

- **disposal** An integer (or list or tuple of integers) specifying the APNG disposal operation to be used for this frame before rendering the next frame. Defaults to 0.
  - 0 (`APNG_DISPOSE_OP_NONE`, default) - No disposal is done on this frame before rendering the next frame.
  - 1 (`PIL.PngImagePlugin.APNG_DISPOSE_OP_BACKGROUND`) - This frame’s modified region is cleared to fully transparent black before rendering the next frame.
  - 2 (`APNG_DISPOSE_OP_PREVIOUS`) - This frame’s modified region is reverted to the previous frame’s contents before rendering the next frame.

- **blend** An integer (or list or tuple of integers) specifying the APNG blend operation to be used for this frame before rendering the next frame. Defaults to 0.
  - 0 (`APNG_BLEND_OP_SOURCE`) - All color components of this frame, including alpha, overwrite the previous output image contents.
• I (APNG_BLEND_OP_OVER) - This frame should be alpha composited with the previous output image contents.

Note: The duration, disposal and blend parameters can be set to lists or tuples to specify values for each individual frame in the animation. The length of the list or tuple must be identical to the total number of actual frames in the APNG animation. If the APNG contains a default image (i.e. default_image is set to True), these list or tuple parameters should not include an entry for the default image.

PPM

Pillow reads and writes PBM, PGM, PPM and PNM files containing 1, L or RGB data.

SGI

Pillow reads and writes uncompressed L, RGB, and RGBA files.

SPIDER

Pillow reads and writes SPIDER image files of 32-bit floating point data (“F;32F”).

Pillow 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.
- n_frames  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):

```python
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:

```python
im.save('newimage.spi', format='SPIDER')
```

For more information about the SPIDER image processing package, see the SPIDER homepage at Wadsworth Center.
TGA

Pillow reads and writes TGA images containing L, LA, P, RGB, and RGBA data. Pillow can read and write both uncompressed and run-length encoded TGAs.

TIFF

Pillow reads and writes TIFF files. It can read both striped and tiled images, pixel and plane interleaved multi-band images. If you have libtiff and its headers installed, Pillow can read and write many kinds of compressed TIFF files. If not, Pillow will only read and write uncompressed files.

**Note:** Beginning in version 5.0.0, Pillow requires libtiff to read or write compressed files. Prior to that release, Pillow had buggy support for reading Packbits, LZW and JPEG compressed TIFFs without using libtiff.

The `open()` method sets the following `info` properties:

- **compression**  Compression mode.
  
  New in version 2.0.0.

- **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.

- **resolution**  Image resolution as an `(xres, yres)` tuple, where applicable. This is a measurement in whichever unit is specified by the file.
  
  New in version 1.1.5.

The `tag_v2` attribute contains a dictionary of TIFF metadata. The keys are numerical indexes from `TiffTags.TAGS_V2`. Values are strings or numbers for single items, multiple values are returned in a tuple of values. Rational numbers are returned as an `IFDRational` object.

  New in version 3.0.0.

For compatibility with legacy code, the `tag` attribute contains a dictionary of decoded TIFF fields as returned prior to version 3.0.0. Values are returned as either strings or tuples of numeric values. Rational numbers are returned as a tuple of `(numerator, denominator)`.

  Deprecated since version 3.0.0.

**Reading Multi-frame TIFF Images**

The TIFF loader supports the `seek()` and `tell()` methods, taking and returning frame numbers within the image file. You can combine these methods to seek to the next frame (`im.seek(im.tell() + 1)`). Frames are numbered from 0 to `im.num_frames - 1`, and can be accessed in any order.

`im.seek()` raises an `EOFError` if you try to seek after the last frame.
Saving Tiff Images

The `save()` method can take the following keyword arguments:

**save_all** If true, Pillow will save all frames of the image to a multiframe tiff document.

   New in version 3.4.0.

**append_images** A list of images to append as additional frames. Each of the images in the list can be single or multiframe images. Note however, that for correct results, all the appended images should have the same `encoderinfo` and `encoderconfig` properties.

   New in version 4.2.0.

**tiffinfo** A `ImageFileDirectory_v2` 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_v2` object and setting the type in `tagtype` with the appropriate numerical value from `TiffTags.TYPES`.

   New in version 2.3.0.

Metadata values that are of the rational type should be passed in using a `IFDRational` object.

   New in version 3.1.0.

For compatibility with legacy code, a `ImageFileDirectory_v1` object may be passed in this field. However, this is deprecated.

   New in version 5.4.0.

Previous versions only supported some tags when writing using libtiff. The supported list is found in `TiffTags.LIBTIFF_CORE`.

   New in version 6.1.0.

Added support for signed types (e.g. `TIFF_SIGNED_LONG`) and multiple values. Multiple values for a single tag must be to `ImageFileDirectory_v2` as a tuple and require a matching type in `tagtype`.

**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"

**quality** The image quality for JPEG compression, on a scale from 0 (worst) to 100 (best). The default is 75.

   New in version 6.1.0.

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** An integer. 1 for no unit, 2 for inches and 3 for centimeters.

**resolution** Either an integer or a float, used for both the x and y resolution.

**x_resolution** Either an integer or a float.

**y_resolution** Either an integer or a float.
dpi  A tuple of (x_resolution, y_resolution), with inches as the resolution unit. For consistency with other image formats, the x and y resolutions of the dpi will be rounded to the nearest integer.

**WebP**

Pillow reads and writes WebP files. The specifics of Pillow’s capabilities with this format are currently undocumented. The `save()` method supports the following options:

- **lossless** If present and true, instructs the WebP writer to use lossless compression.
- **quality** Integer, 1-100, Defaults to 80. For lossy, 0 gives the smallest size and 100 the largest. For lossless, this parameter is the amount of effort put into the compression: 0 is the fastest, but gives larger files compared to the slowest, but best, 100.
- **method** Quality/speed trade-off (0=fast, 6=slower-better). Defaults to 0.
- **icc_profile** 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.

**Saving sequences**

---

**Note:** Support for animated WebP files will only be enabled if the system WebP library is v0.5.0 or later. You can check webp animation support at runtime by calling `features.check("webp_anim")`.

When calling `save()` to write a WebP file, the following options are available when the `save_all` argument is present and true.

- **append_images** A list of images to append as additional frames. Each of the images in the list can be single or multiframe images.
- **duration** The display duration of each frame, in milliseconds. Pass a single integer for a constant duration, or a list or tuple to set the duration for each frame separately.
- **loop** Number of times to repeat the animation. Defaults to 0 (infinite).
- **background** Background color of the canvas, as an RGBA tuple with values in the range of (0-255).
- **minimize_size** If true, minimize the output size (slow). Implicitly disables key-frame insertion.
- **kmin, kmax** Minimum and maximum distance between consecutive key frames in the output. The library may insert some key frames as needed to satisfy this criteria. Note that these conditions should hold: kmax > kmin and kmin >= kmax / 2 + 1. Also, if kmax <= 0, then key-frame insertion is disabled; and if kmax == 1, then all frames will be key-frames (kmin value does not matter for these special cases).
- **allow_mixed** If true, use mixed compression mode; the encoder heuristically chooses between lossy and lossless for each frame.
XBM

Pillow reads and writes X bitmap files (mode 1).

Read-only formats

BLP

BLP is the Blizzard Mipmap Format, a texture format used in World of Warcraft. Pillow supports reading JPEG Compressed or raw BLP1 images, and all types of BLP2 images.

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.

DDS

DDS is a popular container texture format used in video games and natively supported by DirectX. Currently, uncompressed RGB data and DXT1, DXT3, and DXT5 pixel formats are supported, and only in RGBA mode.
New in version 3.4.0: DXT3

FLI, FLC

Pillow reads Autodesk FLI and FLC animations.
The open() method sets the following info properties:

duration The delay (in milliseconds) between each frame.

FPX

Pillow 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.
FTEX

New in version 3.2.0.
The FTEX decoder reads textures used for 3D objects in Independence War 2: Edge Of Chaos. The plugin reads a single texture per file, in the compressed and uncompressed formats.

GBR

The GBR decoder reads GIMP brush files, version 1 and 2.
The `open()` method sets the following info properties:
- `comment` The brush name.
- `spacing` The spacing between the brushes, in pixels. Version 2 only.

GD

Pillow reads uncompressed GD2 files. Note that 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.

IMT

Pillow reads Image Tools images containing L data.

IPTC/NAA

Pillow provides limited read support for IPTC/NAA newsphoto files.

MCIDAS

Pillow identifies and reads 8-bit McIdas area files.

MIC

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.
Note that there may be an embedded gamma of 2.2 in MIC files.
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.

PCD

Pillow reads PhotoCD files containing RGB data. This only reads the 768x512 resolution image from the file. Higher resolutions are encoded in a proprietary encoding.

PIXAR

Pillow provides limited support for PIXAR raster files. The library can identify and read “dumped” RGB files. The format code is PIXAR.

PSD

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

WAL

New in version 1.1.4.

Pillow 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.

WMF

Pillow can identify WMF files.

On Windows, it can read WMF files. By default, it will load the image at 72 dpi. To load it at another resolution:

```python
from PIL import Image
with Image.open("drawing.wmf") as im:
    im.load(dpi=144)
```

To add other read or write support, use `PIL.WmfImagePlugin.register_handler()` to register a WMF handler.

```python
from PIL import Image
from PIL import WmfImagePlugin

class WmfHandler:
    def open(self, im):
        ...
    def load(self, im):
```

(continues on next page)
def save(self, im, fp, filename):
    ...

wmf_handler = WmfHandler()

WmfImagePlugin.register_handler(wmf_handler)
im = Image.open("sample.wmf")

---

XPM

Pillow 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.

Write-only formats

PALM

Pillow provides write-only support for PALM pixmap files.
The format code is Palm, the extension is .palm.

PDF

Pillow can write PDF (Acrobat) images. Such images are written as binary PDF 1.4 files, using either JPEG or HEX encoding depending on the image mode (and whether JPEG support is available or not).
The save() method can take the following keyword arguments:

save_all  If a multiframe image is used, by default, only the first image will be saved. To save all frames, each frame to a separate page of the PDF, the save_all parameter must be present and set to True.
    New in version 3.0.0.
append_images  A list of images to append as additional pages. Each of the images in the list can be single or multiframe images.
    New in version 4.2.0.
append  Set to True to append pages to an existing PDF file. If the file doesn’t exist, an OSError will be raised.
    New in version 5.1.0.
resolution  Image resolution in DPI. This, together with the number of pixels in the image, will determine the physical dimensions of the page that will be saved in the PDF.
title  The document’s title. If not appending to an existing PDF file, this will default to the filename.
    New in version 5.1.0.
Pillow can read XV thumbnail files.

**Identify-only formats**

**BUFR**

New in version 1.1.3.

Pillow 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.

Pillow 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.

Pillow 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 Pillow.

To add read or write support to your application, use PIL.GribStubImagePlugin.register_handler().

HDF5

New in version 1.1.5.

Pillow provides a stub driver for HDF5 files.

To add read or write support to your application, use PIL.Hdf5StubImagePlugin.register_handler().

MPEG

Pillow identifies MPEG files.

Writing Your Own Image Plugin

Pillow uses a plugin model which allows you to add your own decoders to the library, without any changes to the library itself. Such plugins 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 image plugin manually.

Pillow decodes files in two stages:

1. It loops over the available image plugins in the loaded order, and calls the plugin’s _accept function with the first 16 bytes of the file. If the _accept function returns true, the plugin’s _open method is called to set up the image metadata and image tiles. The _open method is not for decoding the actual image data.

2. When the image data is requested, the ImageFile.load method is called, which sets up a decoder for each tile and feeds the data to it.

An image plugin should contain a format handler derived from 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, which contain a decoder name, extents of the tile, and any decoder-specific data. The format handler class must be explicitly registered, via a call to the Image module.

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

The following plugin 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:

```python
from PIL import Image, ImageFile

def _accept(prefix):
    return prefix[:4] == b"SPAM"

class SpamImageFile(ImageFile.ImageFile):
    format = "SPAM"
    format_description = "Spam raster image"

    def __open__(self):
        header = self.fp.read(128).split()

        # 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(SpamImageFile.format, SpamImageFile, _accept)

Image.register_extensions(SpamImageFile.format, [".spam", ".spa", ".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 plugin, the calling code considers exceptions like `SyntaxError`, `KeyError`, `IndexError`, `EOFError` and `struct.error` as a failure to identify the file.

Note that the image plugin 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.

Once the plugin has been imported, it can be used:
from PIL import Image
import SpamImagePlugin
with Image.open("hopper.spam") as im:
    pass

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:

\[(\text{decoder}, \text{region}, \text{offset}, \text{parameters})\]

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.

Decoders

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.frombytes() function, use the following syntax:

```python
image = Image.frombytes( mode, size, data, "raw", raw mode, stride, orientation )
```

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

\[(\text{raw mode}, \text{stride}, \text{orientation})\]

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:

<table>
<thead>
<tr>
<th>mode</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-bit bilevel, stored with the leftmost pixel in the most significant bit. 0 means black, 1 means white.</td>
</tr>
<tr>
<td>1;I</td>
<td>1-bit inverted bilevel, stored with the leftmost pixel in the most significant bit. 0 means white, 1 means black.</td>
</tr>
<tr>
<td>1;R</td>
<td>1-bit reversed bilevel, stored with the leftmost pixel in the least significant bit. 0 means black, 1 means white.</td>
</tr>
<tr>
<td>L</td>
<td>8-bit greyscale. 0 means black, 255 means white.</td>
</tr>
<tr>
<td>L;I</td>
<td>8-bit inverted greyscale. 0 means white, 255 means black.</td>
</tr>
<tr>
<td>P</td>
<td>8-bit palette-mapped image.</td>
</tr>
<tr>
<td>RGB</td>
<td>24-bit true colour, stored as (red, green, blue).</td>
</tr>
<tr>
<td>BGR</td>
<td>24-bit true colour, stored as (blue, green, red).</td>
</tr>
<tr>
<td>RGBX</td>
<td>24-bit true colour, stored as (red, green, blue, pad). The pad pixels may vary.</td>
</tr>
<tr>
<td>RGB;L</td>
<td>24-bit true colour, line interleaved (first all red pixels, then all green pixels, finally all blue pixels).</td>
</tr>
</tbody>
</table>

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 plugin implementations provided with the library.

Decoding floating point data

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

You can use the raw decoder to read images where data is packed in any standard machine data type, using one of the following raw modes:
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 `PIL.Image.frombytes()` function, use the following syntax:

```python
image = Image.frombytes( mode, size, data, "bit", bits, pad, fill, sign, orientation )
```

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

```python
(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.

Writing Your Own File Decoder in C

There are 3 stages in a file decoder’s lifetime:

1. Setup: Pillow looks for a function in the decoder registry, falling back to a function named [decodername]_decoder on the internal core image object. That function is called with the args tuple from the tile setup in the _open method.

2. Decoding: The decoder’s decode function is repeatedly called with chunks of image data.

3. Cleanup: If the decoder has registered a cleanup function, it will be called at the end of the decoding process, even if there was an exception raised.

Setup

The current conventions are that the decoder setup function is named PyImaging_[Decodername]DecoderNew and defined in decode.c. The python binding for it is named [decodername]_decoder and is setup from within the _imaging.c file in the codecs section of the function array.

The setup function needs to call PyImaging_DecoderNew and at the very least, set the decode function pointer. The fields of interest in this object are:

- **decode**  Function pointer to the decode function, which has access to im, state, and the buffer of data to be added to the image.

- **cleanup**  Function pointer to the cleanup function, has access to state.

- **im**  The target image, will be set by Pillow.

- **state**  An ImagingCodecStateInstance, will be set by Pillow. The context member is an opaque struct that can be used by the decoder to store any format specific state or options.

- **pulls_fd**  EXPERIMENTAL – WARNING, interface may change. If set to 1, state->fd will be a pointer to the Python file like object. The decoder may use the functions in codec_fd.c to read directly from the file like object rather than have the data pushed through a buffer. Note that this implementation may be refactored until this warning is removed.

  New in version 3.3.0.

Decoding

The decode function is called with the target (core) image, the decoder state structure, and a buffer of data to be decoded.

**Experimental** – If pulls_fd is set, then the decode function is called once, with an empty buffer. It is the decoder’s responsibility to decode the entire tile in that one call. The rest of this section only applies if pulls_fd is not set.

It is the decoder’s responsibility to pull as much data as possible out of the buffer and return the number of bytes consumed. The next call to the decoder will include the previous unconsumed tail. The decoder function will be called multiple times as the data is read from the file like object.

If an error occurs, set state->errcode and return -1.

Return -1 on success, without setting the errcode.
Cleanup

The cleanup function is called after the decoder returns a negative value, or if there is a read error from the file. This function should free any allocated memory and release any resources from external libraries.

Writing Your Own File Decoder in Python

Python file decoders should derive from `PIL.ImageFile.PyDecoder` and should at least override the decode method. File decoders should be registered using `PIL.Image.register_decoder()`. As in the C implementation of the file decoders, there are three stages in the lifetime of a Python-based file decoder:

1. Setup: Pillow looks for the decoder in the registry, then instantiates the class.
2. Decoding: The decoder instance’s `decode` method is repeatedly called with a buffer of data to be interpreted.
3. Cleanup: The decoder instance’s `cleanup` method is called.

1.3 Reference

1.3.1 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.

Examples

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

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).

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

Create thumbnails

The following script creates nice thumbnails of all JPEG images in the current directory preserving aspect ratios with 128x128 max resolution.

```python
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")
```
Functions

PIL.Image.open(fp, mode='r', formats=None)

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(). See File Handling in Pillow.

Parameters

- **fp** – A filename (string), pathlib.Path object or a file object. The file object must implement file.read, file.seek, and file.tell methods, and be opened in binary mode.
- **mode** – The mode. If given, this argument must be “r”.
- **formats** – A list or tuple of formats to attempt to load the file in. This can be used to restrict the set of formats checked. Pass None to try all supported formats. You can print the set of available formats by running python -m PIL or using the PIL.features.pilinfo() function.

Returns

An Image object.

Raises

- **FileNotFoundException** – If the file cannot be found.
- **PIL.UnidentifiedImageError** – If the image cannot be opened and identified.
- **ValueError** – If the mode is not “r”, or if a StringIO instance is used for fp.
- **TypeError** – If formats is not None, a list or a tuple.

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 number of pixels in an image is over a certain limit, PIL.Image.MAX_IMAGE_PIXELS.

This threshold can be changed by setting PIL.Image.MAX_IMAGE_PIXELS. It can be disabled by setting Image.MAX_IMAGE_PIXELS = None.

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.

If the number of pixels is greater than twice PIL.Image.MAX_IMAGE_PIXELS, then a DecompressionBombError will be raised instead.
Image processing

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

   Parameters
   • im1 – The first image. Must have mode RGBA.
   • im2 – The second image. Must have mode RGBA, and the same 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:

   \[ \text{out} = \text{image1} \times (1.0 - \alpha) + \text{image2} \times \alpha \]

   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.

If you have an image in NumPy:

```python
from PIL import Image
import numpy as np
im = Image.open('hopper.jpg')
a = np.asarray(im)
```

Then this can be used to convert it to a Pillow image:

```python
im = Image.fromarray(a)
```

**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.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 `frombytes()`. 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:

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

**Returns** An Image object.

New in version 1.1.4.

**Generating images**

```
PIL.Image.effect_mandelbrot(size, extent, quality)
```

Generate a Mandelbrot set covering the given extent.

**Parameters**

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **extent** – The extent to cover, as a 4-tuple: (x0, y0, x1, y2).
- **quality** – Quality.

```
PIL.Image.effect_noise(size, sigma)
```

Generate Gaussian noise centered around 128.

**Parameters**

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **sigma** – Standard deviation of noise.
PIL.Image.linear_gradient (mode)
Generate 256x256 linear gradient from black to white, top to bottom.

Parameters mode – Input mode.

PIL.Image.radial_gradient (mode)
Generate 256x256 radial gradient from black to white, centre to edge.

Parameters mode – Input mode.

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_save_all (id, driver)
Registers an image function to save all the frames of a multiframe format. 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.
PIL.Image.register_extensions(id, extensions)

Registers image extensions. This function should not be used in application code.

Parameters

- id – An image format identifier.
- extensions – A list of extensions used for this format.

PIL.Image.registered_extensions()

Returns a dictionary containing all file extensions belonging to registered plugins.

PIL.Image.register_decoder(name, decoder)

Registers an image decoder. This function should not be used in application code.

Parameters

- name – The name of the decoder
- decoder – A callable(mode, args) that returns an ImageFile.PyDecoder object

New in version 4.1.0.

PIL.Image.register_encoder(name, encoder)

Registers an image encoder. This function should not be used in application code.

Parameters

- name – The name of the encoder
- encoder – A callable(mode, args) that returns an ImageFile.PyEncoder object

New in version 4.1.0.

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.alpha_composite(im, dest=0, 0, source=0, 0)

‘In-place’ analog of Image.alpha_composite. Composites an image onto this image.

Parameters

- im – image to composite over this one
- dest – Optional 2 tuple (left, top) specifying the upper left corner in this (destination) image.
- source – Optional 2 (left, top) tuple for the upper left corner in the overlay source image, or 4 tuple (left, top, right, bottom) for the bounds of the source rectangle

Performance Note: Not currently implemented in-place in the core layer.
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 greyscale (mode “L”), the library uses the ITU-R 601-2 luma transform:

\[L = R \times \frac{299}{1000} + G \times \frac{587}{1000} + B \times \frac{114}{1000}\]

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 values larger than 128 are set to 255 (white), all other values to 0 (black). To use other thresholds, use the point() method.

When converting from “RGBA” to “P” without a matrix argument, this passes the operation to quantize(), and dither and palette are ignored.

Parameters

- **mode** – The requested mode. See: Modes.
- **matrix** – An optional conversion matrix. If given, this should be 4- or 12-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). Note that this is not used when matrix is supplied.
- **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:

```plaintext
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. See Coordinate System.

Note: Prior to Pillow 3.4.0, this was a lazy operation.

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

Return type **Image**

56 Chapter 1. Overview
Returns An `Image` object.

This crops the input image with the provided coordinates:

```python
from PIL import Image

im = Image.open("hopper.jpg")

# The crop method from the Image module takes four coordinates as input.
# The right can also be represented as (left+width)
# and lower can be represented as (upper+height).
(left, upper, right, lower) = (20, 20, 100, 100)

# Here the image "im" is cropped and assigned to new variable im_crop
im_crop = im.crop((left, upper, right, lower))
```

Image.\texttt{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.

If any changes are made, returns a tuple with the chosen mode and box with coordinates of the original image within the altered one.

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

Note: This method is not implemented for most images. It is currently implemented only for JPEG and MPO images.

\textbf{Parameters}

- \texttt{mode} – The requested mode.
- \texttt{size} – The requested size.

Image.\texttt{effect_spread}(distance)
Randomly spread pixels in an image.

\textbf{Parameters} \texttt{distance} – Distance to spread pixels.

Image.\texttt{entropy}(mask=None, extrema=None)
Calculates and returns the entropy for the image.

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

If a mask is provided, the method employs the 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”).

\textbf{Parameters}

- \texttt{mask} – An optional mask.
- \texttt{extrema} – An optional tuple of manually-specified extrema.

\textbf{Returns} A float value representing the image entropy

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

\textbf{Parameters} \texttt{filter} – Filter kernel.

\textbf{Returns} An `Image` object.
Image.frombytes(data, decoder_name='raw', *args)

Loads this image with pixel data from a bytes object.

This method is similar to the frombytes() function, but loads data into this image instead of creating a new image object.

This blurs the input image using a filter from the ImageFilter module:

```python
from PIL import Image, ImageFilter
im = Image.open("hopper.jpg")

# Blur the input image using the filter ImageFilter.BLUR
im_blurred = im.filter(filter=ImageFilter.BLUR)
```

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

This helps to get the bands of the input image:

```python
from PIL import Image
im = Image.open("hopper.jpg")
print(im.getbands())  # Returns ('R', 'G', 'B')
```

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. See Coordinate System. If the image is completely empty, this method returns None.

This helps to get the bounding box coordinates of the input image:

```python
from PIL import Image
im = Image.open("hopper.jpg")
print(im.getbbox())  # Returns four coordinates in the format (left, upper, right, lower)
```

Image.getchannel(channel)

Returns an image containing a single channel of the source image.

**Parameters** channel – What channel to return. Could be index (0 for “R” channel of “RGB”) or channel name (“A” for alpha channel of “RGBA”).

**Returns** An image in “L” mode.

New in version 4.3.0.

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.getexif()

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). See Coordinate System.

Returns

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

Image.getprojection()

Get projection to x and y axes

Returns

Two sequences, indicating where there are non-zero pixels along the X-axis and the Y-axis, respectively.

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.
• extrema – An optional tuple of manually-specified extrema.

Returns

A list containing pixel counts.

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)). See Coordinate System. 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 an 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 will mix the two images together, including their alpha channels if they have them.

See alpha_composite() if you want to combine images with respect to their alpha channels.

**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 65536 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.

  It may also be an ImagePointHandler object:

  ```python
  class Example(Image.ImagePointHandler):
      def point(self, data):
          # Return result
  ```

- `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”, “PA”, “L” or “LA” 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).

• **rawmode** – The raw mode of the palette.

`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. In addition to this, RGB and RGBA tuples are accepted for P 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). See Coordinate System.

• **value** – The pixel value.

`Image.quantize(colors=256, method=None, kmeans=0, palette=None, dither=1)`

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

**Parameters**

• **colors** – The desired number of colors, <= 256

• **method** – *MEDIANCUT* (median cut), *MAXCOVERAGE* (maximum coverage), *FASTOCTREE* (fast octree), *LIBIMAGEQUANT* (libimagequant; check support using `PIL.features.check_feature()` with feature="libimagequant").

• **kmeans** – Integer

• **palette** – Quantize to the palette of given `PIL.Image.Image`

• **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). Default: 1 (legacy setting)

**Returns** A new image

`Image.reduce(factor, box=None)`

Returns a copy of the image reduced `factor` times. If the size of the image is not dividable by `factor`, the resulting size will be rounded up.

**Parameters**

• **factor** – A greater than 0 integer or tuple of two integers for width and height separately.
• **box** – An optional 4-tuple of ints providing the source image region to be reduced. The values must be within (0, 0, width, height) rectangle. If omitted or None, the entire source is used.

**Image.remap_palette(dest_map, source_palette=None)**

Rewrites the image to reorder the palette.

**Parameters**

- **dest_map** – A list of indexes into the original palette. e.g. [1, 0] would swap a two item palette, and list(range(256)) is the identity transform.

- **source_palette** – Bytes or None.

**Returns** An Image object.

**Image.resize(size, resample=3, box=None, reducing_gap=None)**

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, PIL.Image.BOX, PIL.Image.BILINEAR, PIL.Image.HAMMING, PIL.Image.BICUBIC or PIL.Image.LANCZOS. Default filter is PIL.Image.BICUBIC. If the image has mode “1” or “P”, it is always set to PIL.Image.NEAREST. See: Filters.

- **box** – An optional 4-tuple of floats providing the source image region to be scaled. The values must be within (0, 0, width, height) rectangle. If omitted or None, the entire source is used.

- **reducing_gap** – Apply optimization by resizing the image in two steps. First, reducing the image by integer times using reduce(). Second, resizing using regular resampling. The last step changes size no less than by reducing_gap times. reducing_gap may be None (no first step is performed) or should be greater than 1.0. The bigger reducing_gap, the closer the result to the fair resampling. The smaller reducing_gap, the faster resizing. With reducing_gap greater or equal to 3.0, the result is indistinguishable from fair resampling in most cases. The default value is None (no optimization).

**Returns** An Image object.

This resizes the given image from (width, height) to (width/2, height/2):

```python
from PIL import Image

im = Image.open("hopper.jpg")

# Provide the target width and height of the image
(width, height) = (im.width // 2, im.height // 2)
im_resized = im.resize((width, height))
```

**Image.rotate(angle, resample=0, expand=0, center=None, translate=None, fillcolor=None)**

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 to `PIL.Image.NEAREST`. See Filters.

• **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. Note that the expand flag assumes rotation around the center and no translation.

• **center** – Optional center of rotation (a 2-tuple). Origin is the upper left corner. Default is the center of the image.

• **translate** – An optional post-rotate translation (a 2-tuple).

• **fillcolor** – An optional color for area outside the rotated image.

**Returns**  An `Image` object.

This rotates the input image by \( \theta \) degrees counter clockwise:

```python
from PIL import Image

im = Image.open("hopper.jpg")

# Rotate the image by 60 degrees counter clockwise
theta = 60
# Angle is in degrees counter clockwise
im_rotated = im.rotate(angle=theta)

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** – A filename (string), pathlib.Path object 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.

• **params** – Extra parameters to the image writer.

**Returns**  None

**Raises**

• **ValueError** – 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.

See tell().

If defined, n_frames refers to the number of available frames.

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.

This method calls PIL.ImageShow.show() internally. You can use PIL.ImageShow.register() to override its default behaviour.

The image is first saved to a temporary file. By default, it will be in PNG format.

On Unix, the image is then opened using the display, eog or xv utility, depending on which one can be found.

On macOS, the image is opened with the native Preview application.

On Windows, the image is opened with the standard PNG display utility.

Parameters title – Optional title to use for the image window, where possible.

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).

If you need only one band, getchannel() method can be more convenient and faster.

Returns A tuple containing bands.

Image.tell()

Returns the current frame number. See seek().

If defined, n_frames refers to the number of available frames.

Returns Frame number, starting with 0.

Image.thumbnail(size, resample=3, reducing_gap=2.0)

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.BOX, PIL.Image.BILINEAR, PIL.Image.HAMMING, 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). See: Filters.

• reducing_gap – Apply optimization by resizing the image in two steps. First, reducing the image by integer times using reduce() or draft() for JPEG images. Second, resizing using regular resampling. The last step changes size no less than by reducing_gap times. reducing_gap may be None (no first step is performed) or should be greater than
1.0. The bigger `reducing_gap`, the closer the result to the fair resampling. The smaller `reducing_gap`, the faster resizing. With `reducing_gap` greater or equal to 3.0, the result is indistinguishable from fair resampling in most cases. The default value is 2.0 (very close to fair resampling while still being faster in many cases).

**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.

**Warning:** This method returns the raw image data from the internal storage. For compressed image data (e.g. PNG, JPEG) use *save()* with a BytesIO parameter for in-memory data.

**Parameters**

- encoder_name – What encoder to use. The default is to use the standard “raw” encoder.

- args – Extra arguments to the encoder.

**Returns** A bytes object.

Image.transform(*size, method, data=None, resample=0, fill=1, fillcolor=None*)

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).

It may also be an ImageTransformHandler object:

```
class Example(Image.ImageTransformHandler):
    def transform(self, size, data, resample, fill=1):
        # Return result
```

It may also be an object with a method getData method that returns a tuple supplying new method and data values:
```python
class Example:
    def getdata(self):
        method = Image.EXTENT
        data = (0, 0, 100, 100)
        return method, data
```

- **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. See: Filters.
- **fill** – If method is an ImageTransformHandler object, this is one of the arguments passed to it. Otherwise, it is unused.
- **fillcolor** – Optional fill color for the area outside the transform in the output image.

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, PIL.Image.TRANSPOSE or PIL.Image.TRANSVERSE.

Returns Returns a flipped or rotated copy of this image.

This flips the input image by using the FLIP_LEFT_RIGHT method.

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

# Flip the image from left to right
im_flipped = im.transpose(method=Image.FLIP_LEFT_RIGHT)

# To flip the image from top to bottom,
# use the method "Image.FLIP_TOP_BOTTOM"
```

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.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.

If the file associated with the image was opened by Pillow, then this method will close it. The exception to this is if the image has multiple frames, in which case the file will be left open for seek operations. See File Handling in Pillow for more information.

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. See *File Handling in Pillow* for more information.

**Image Attributes**

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

*Image.*`filename`: *str*

The filename or path of the source file. Only images created with the factory function `open` have a filename attribute. If the input is a file like object, the filename attribute is set to an empty string.

*Image.*`format`: *Optional*[str]

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`.

*Image.*`mode`: *str*

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.

*Image.*`size`: *tuple*[int]

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

*Image.*`width`: *int*

Image width, in pixels.

*Image.*`height`: *int*

Image height, in pixels.

*Image.*`palette`: *Optional*[PIL.*ImagePalette.*ImagePalette]

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

*Image.*`info`: *dict*

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.

*Image.*`is_animated`: *bool*

True if this image has more than one frame, or `False` otherwise.

This attribute is only defined by image plugins that support animated images. Plugins may leave this attribute undefined if they don’t support loading animated images, even if the given format supports animated images.

Given that this attribute is not present for all images use `getattr(image, "is_animated", False)` to check if Pillow is aware of multiple frames in an image regardless of its format.

**See also:**

*n_frames, seek() and tell()*

*Image.*`n_frames`: *int*

The number of frames in this image.

This attribute is only defined by image plugins that support animated images. Plugins may leave this attribute undefined if they don’t support loading animated images, even if the given format supports animated images.
Given that this attribute is not present for all images use `getattr(image, "n_frames", 1)` to check the number of frames that Pillow is aware of in an image regardless of its format.

See also:

`is_animated`, `seek()` and `tell()`

### Classes

**class PIL.Image.Exif**

Bases: `collections.abc.MutableMapping`

```
endian = '<'
get_ifd(tag)
load(data)
tobytes(offset=8)
```

**class PIL.Image.ImagePointHandler**

Used as a mixin by point transforms (for use with `point()`)

**class PIL.Image.ImageTransformHandler**

Used as a mixin by geometry transforms (for use with `transform()`)

### Constants

**PIL.Image.NONE**

**PIL.Image.MAX_IMAGE_PIXELS**

Set to 89,478,485, approximately 0.25GB for a 24-bit (3 bpp) image. See `open()` for more information about how this is used.

### Transpose methods

Used to specify the `Image.transpose()` method to use.

**PIL.Image.FLIP_LEFT_RIGHT**

**PIL.Image.FLIP_TOP_BOTTOM**

**PIL.Image.ROTATE_90**

**PIL.Image.ROTATE_180**

**PIL.Image.ROTATE_270**

**PIL.Image.TRANSPOSE**

**PIL.Image.TRANSVERSE**
**Transform methods**

Used to specify the `Image.transform()` method to use.

- **PIL.Image.AFFINE**
  Affine transform

- **PIL.Image.EXTENT**
  Cut out a rectangular subregion

- **PIL.Image.PERSPECTIVE**
  Perspective transform

- **PIL.Image.QUAD**
  Map a quadrilateral to a rectangle

- **PIL.Image.MESH**
  Map a number of source quadrilaterals in one operation

**Resampling filters**

See *Filters* for details.

- **PIL.Image.NEAREST**
- **PIL.Image.BOX**
- **PIL.Image.BILINEAR**
- **PIL.Image.HAMMING**
- **PIL.Image.BICUBIC**
- **PIL.Image.LANCZOS**

Some filters are also available under the following names for backwards compatibility:

- **PIL.Image.NONE = NEAREST**
- **PIL.Image.LINEAR = BILINEAR**
- **PIL.Image.CUBIC = BICUBIC**
- **PIL.Image.ANTIALIAS = LANCZOS**

**Dither modes**

Used to specify the dithering method to use for the `convert()` and `quantize()` methods.

- **PIL.Image.NONE**
  No dither

- **PIL.Image.FLOYDSTEINBERG**
  Floyd-Steinberg dither
**Palettes**

Used to specify the palette to use for the `convert()` method.

- `PIL.Image.WEB`
- `PIL.Image.ADAPTIVE`

**Quantization methods**

Used to specify the quantization method to use for the `quantize()` method.

- `PIL.Image.MEDIANCUT`
  - Median cut
- `PIL.Image.MAXCOVERAGE`
  - Maximum coverage
- `PIL.Image.FASTOCTREE`
  - Fast octree
- `PIL.Image.LIBIMAGEQUANT`
  - libimagequant

Check support using `PIL.features.check_feature()` with feature="libimagequant".

### 1.3.2 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”).

**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.

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

Return type `Image`

`PIL.ImageChops.add_modulo(image1, image2)`

Add two images, without clipping the result.

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

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

    Return type  Image

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

    Both of the images must have mode “1”. If you would like to perform a logical AND on an image with a mode
    other than “1”, try multiply() instead, using a black-and-white mask as the second image.

    out = ((image1 and image2) % MAX)

    Return type  Image
PIL.ImageChops.logical_or(image1, image2)
Logical OR between two images.
Both of the images must have mode “1”.

\[
\text{out} = ((\text{image1 \ or \ image2}) \mod \text{MAX})
\]

\textbf{Return type} Image

PIL.ImageChops.logical_xor(image1, image2)
Logical XOR between two images.
Both of the images must have mode “1”.

\[
\text{out} = ((\text{bool(image1)} \neq \text{bool(image2)}) \mod \text{MAX})
\]

\textbf{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.

\[
\text{out} = \text{image1} \times \text{image2} / \text{MAX}
\]

\textbf{Return type} Image

PIL.ImageChops.soft_light(image1, image2)
Superimposes two images on top of each other using the Soft Light algorithm

\textbf{Return type} Image

PIL.ImageChops.hard_light(image1, image2)
Superimposes two images on top of each other using the Hard Light algorithm

\textbf{Return type} Image

PIL.ImageChops.overlay(image1, image2)
Superimposes two images on top of each other using the Overlay algorithm

\textbf{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 \texttt{yoffset} is omitted, it is assumed to be equal to \texttt{xoffset}.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{xoffset} – The horizontal distance.
  \item \texttt{yoffset} – The vertical distance. If omitted, both distances are set to the same value.
\end{itemize}

\textbf{Return type} Image

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

\[
\text{out} = \text{MAX} - ((\text{MAX} - \text{image1}) \times (\text{MAX} - \text{image2}) / \text{MAX})
\]
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)

1.3.3 ImageCms Module

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

class PIL.ImageCms.ImageCmsTransform(input, output, input_mode, output_mode, intent=0,
proof=None, proof_intent=3, flags=0)
Transform. This can be used with the procedural API, or with the standard point() method.

Will return the output profile in the output.info['icc_profile'].

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

Functions

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

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

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

If im.mode, transform.inMode or transform.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 calculation 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** – An *Image* object, and im.mode must be the same as the *inMode* supported by the transform.

• **transform** – A valid CmsTransform class object

• **inPlace** – Bool. 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 *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
ImageCms.INTENT_PERCEPTUAL = 0 (DEFAULT)
ImageCms.INTENT_RELATIVE_COLORIMETRIC = 1
ImageCms.INTENT_SATURATION = 2
ImageCms.INTENT_ABSOLUTE_COLORIMETRIC = 3

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

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

  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

1.3. Reference 75
• **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

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

  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

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

  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

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

  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

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

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** PyCMS Error –

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

If colorSpace not in ["LAB", "XYZ", "sRGB"], a PyCMS Error is raised.

If using LAB and colorTemp is not a positive integer, a PyCMS Error is raised.

If an error occurs while creating the profile, a PyCMS Error 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** PyCMS Error –

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 PyCMS Error is raised.

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

Use this function to determine the default (and usually best optimized) 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.
ImageCms.INTENT_PERCEPTUAL = 0  (DEFAULT)  ImageCms.INTENT_RELATIVE_COLORIMETRIC = 1  ImageCms.INTENT_SATURATION = 2  ImageCms.INTENT_ABSOLUTE_COLORIMETRIC = 3

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 valid 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 `intent` 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
  
  ImageCms.INTENT_PERCEPTUAL = 0 (DEFAULT)
  ImageCms.INTENT_RELATIVE_COLORIMETRIC = 1
  ImageCms.INTENT_SATURATION = 2
  ImageCms.INTENT.Absolute_COLORIMETRIC = 3

  see the pyCMS documentation for details on rendering intents and what they do.
- **direction** – Integer specifying 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=False, 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 is 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 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
  
  ImageCms.INTENT_PERCEPTUAL = 0 (DEFAULT)
  ImageCms.INTENT_RELATIVE_COLORIMETRIC = 1
  ImageCms.INTENT_SATURATION = 2
  ImageCms.INTENT.Absolute_COLORIMETRIC = 3

1.3. Reference
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. 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 Image object, depending on the value of inPlace

**Raises** PyCMSError –

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

**CmsProfile**

The ICC color profiles are wrapped in an instance of the class CmsProfile. The specification ICC.1:2010 contains more information about the meaning of the values in ICC profiles.

For convenience, all XYZ-values are also given as xyY-values (so they can be easily displayed in a chromaticity diagram, for example).

**class** PIL.ImageCms.CmsProfile

**creation_date**: Optional[datetime.datetime]

Date and time this profile was first created (see 7.2.1 of ICC.1:2010).

**version**: float

The version number of the ICC standard that this profile follows (e.g. 2.0).

**icc_version**: int

Same as version, but in encoded format (see 7.2.4 of ICC.1:2010).

**device_class**: str

4-character string identifying the profile class. One of scnr, mntr, prtr, link, spac, abst, nmcl (see 7.2.5 of ICC.1:2010 for details).

**xcolor_space**: str

4-character string (padded with whitespace) identifying the color space, e.g. XYZ, RGB or CMYK (see 7.2.6 of ICC.1:2010 for details).

**connection_space**: str

4-character string (padded with whitespace) identifying the color space on the B-side of the transform (see 7.2.7 of ICC.1:2010 for details).

**header_flags**: int

The encoded header flags of the profile (see 7.2.11 of ICC.1:2010 for details).

**header_manufacturer**: str

4-character string (padded with whitespace) identifying the device manufacturer, which shall match the signature contained in the appropriate section of the ICC signature registry found at www.color.org (see 7.2.12 of ICC.1:2010).

**header_model**: str

4-character string (padded with whitespace) identifying the device model, which shall match the signature
contained in the appropriate section of the ICC signature registry found at www.color.org (see 7.2.13 of ICC.1:2010).

**attributes**: `int`
Flags used to identify attributes unique to the particular device setup for which the profile is applicable (see 7.2.14 of ICC.1:2010 for details).

**rendering_intent**: `int`
The rendering intent to use when combining this profile with another profile (usually overridden at runtime, but provided here for DeviceLink and embedded source profiles, see 7.2.15 of ICC.1:2010).

One of `ImageCms.INTENT_ABSOLUTE_COLORIMETRIC`, `ImageCms.INTENT_PERCEPTUAL`, `ImageCms.INTENT_RELATIVE_COLORIMETRIC` and `ImageCms.INTENT_SATURATION`.

**profile_id**: `bytes`
A sequence of 16 bytes identifying the profile (via a specially constructed MD5 sum), or 16 binary zeroes if the profile ID has not been calculated (see 7.2.18 of ICC.1:2010).

**copyright**: `Optional[str]`
The text copyright information for the profile (see 9.2.21 of ICC.1:2010).

**manufacturer**: `Optional[str]`
The (English) display string for the device manufacturer (see 9.2.22 of ICC.1:2010).

**model**: `Optional[str]`
The (English) display string for the device model of the device for which this profile is created (see 9.2.23 of ICC.1:2010).

**profile_description**: `Optional[str]`
The (English) display string for the profile description (see 9.2.41 of ICC.1:2010).

**target**: `Optional[str]`
The name of the registered characterization data set, or the measurement data for a characterization target (see 9.2.14 of ICC.1:2010).

**red_colorant**: `Optional[tuple[tuple[float]]]`
The first column in the matrix used in matrix/TRC transforms (see 9.2.44 of ICC.1:2010).

The value is in the format `((X, Y, Z), (x, y, Y))`, if available.

**green_colorant**: `Optional[tuple[tuple[float]]]`
The second column in the matrix used in matrix/TRC transforms (see 9.2.30 of ICC.1:2010).

The value is in the format `((X, Y, Z), (x, y, Y))`, if available.

**blue_colorant**: `Optional[tuple[tuple[float]]]`
The third column in the matrix used in matrix/TRC transforms (see 9.2.4 of ICC.1:2010).

The value is in the format `((X, Y, Z), (x, y, Y))`, if available.

**luminance**: `Optional[tuple[tuple[float]]]`
The absolute luminance of emissive devices in candelas per square metre as described by the Y channel (see 9.2.32 of ICC.1:2010).

The value is in the format `((X, Y, Z), (x, y, Y))`, if available.

**chromaticity**: `Optional[tuple[tuple[float]]]`
The data of the phosphor/colorant chromaticity set used (red, green and blue channels, see 9.2.16 of ICC.1:2010).

The value is in the format `((x, y, Y), (x, y, Y), (x, y, Y))`, if available.

**chromatic_adaption**: `tuple[tuple[float]]`
The chromatic adaption matrix converts a color measured using the actual illumination conditions and
relative to the actual adopted white, to an color relative to the PCS adopted white, with complete adap-
tation from the actual adopted white chromaticity to the PCS adopted white chromaticity (see 9.2.15 of
ICC.1:2010).

Two 3-tuples of floats are returned in a 2-tuple, one in (X, Y, Z) space and one in (x, y, Y) space.

colorant_table: list[str]
This tag identifies the colorants used in the profile by a unique name and set of PCSXYZ or PCSLAB
values (see 9.2.19 of ICC.1:2010).

colorant_table_out: list[str]
This tag identifies the colorants used in the profile by a unique name and set of PCSLAB values (for
DeviceLink profiles only, see 9.2.19 of ICC.1:2010).

colorimetric_intent: Optional[str]
4-character string (padded with whitespace) identifying the image state of PCS colorimetry produced using
the colorimetric intent transforms (see 9.2.20 of ICC.1:2010 for details).

perceptual_rendering_intent_gamut: Optional[str]
4-character string (padded with whitespace) identifying the (one) standard reference medium gamut (see
9.2.37 of ICC.1:2010 for details).

saturation_rendering_intent_gamut: Optional[str]
4-character string (padded with whitespace) identifying the (one) standard reference medium gamut (see
9.2.37 of ICC.1:2010 for details).

technology: Optional[str]
4-character string (padded with whitespace) identifying the device technology (see 9.2.47 of ICC.1:2010
for details).

media_black_point: Optional[tuple[tuple[float]]]
This tag specifies the media black point and is used for generating absolute colorimetry.

This tag was available in ICC 3.2, but it is removed from version 4.

The value is in the format ((X, Y, Z), (x, y, Y)), if available.

media_white_point_temperature: Optional[float]
Calculates the white point temperature (see the LCMS documentation for more information).

viewing_condition: Optional[str]
The (English) display string for the viewing conditions (see 9.2.48 of ICC.1:2010).

screening_description: Optional[str]
The (English) display string for the screening conditions.

This tag was available in ICC 3.2, but it is removed from version 4.

red_primary: Optional[tuple[tuple[float]]]
The XYZ-transformed of the RGB primary color red (1, 0, 0).

The value is in the format ((X, Y, Z), (x, y, Y)), if available.

green_primary: Optional[tuple[tuple[float]]]
The XYZ-transformed of the RGB primary color green (0, 1, 0).

The value is in the format ((X, Y, Z), (x, y, Y)), if available.

blue_primary: Optional[tuple[tuple[float]]]
The XYZ-transformed of the RGB primary color blue (0, 0, 1).

The value is in the format ((X, Y, Z), (x, y, Y)), if available.
is_matrix_shaper:  bool
    True if this profile is implemented as a matrix shaper (see documentation on LCMS).

clut:  dict[tuple[bool]]
    Returns a dictionary of all supported intents and directions for the CLUT model.
    The dictionary is indexed by intents (ImageCms.INTENTbsolute_COLORIMETRIC, ImageCms.INTENT_PERCEPTUAL, ImageCms.INTENT_RELATIVE_COLORIMETRIC and ImageCms.INTENT_SATURATION).
    The values are 3-tuples indexed by directions (ImageCms.DIRECTION_INPUT, ImageCms.DIRECTION_OUTPUT, ImageCms.DIRECTION_PROOF).
    The elements of the tuple are booleans. If the value is True, that intent is supported for that direction.

intent_supported:  dict[tuple[bool]]
    Returns a dictionary of all supported intents and directions.
    The dictionary is indexed by intents (ImageCms.INTENTabsolute_COLORIMETRIC, ImageCms.INTENT_PERCEPTUAL, ImageCms.INTENT_RELATIVE_COLORIMETRIC and ImageCms.INTENT_SATURATION).
    The values are 3-tuples indexed by directions (ImageCms.DIRECTION_INPUT, ImageCms.DIRECTION_OUTPUT, ImageCms.DIRECTION_PROOF).
    The elements of the tuple are booleans. If the value is True, that intent is supported for that direction.

There is one function defined on the class:

is_intent_supported(intent, direction)
    Returns if the intent is supported for the given direction.
    Note that you can also get this information for all intents and directions with intent_supported.

Parameters

- intent – One of ImageCms.INTENTabsolute_COLORIMETRIC, ImageCms.INTENT_PERCEPTUAL, ImageCms.INTENT_RELATIVE_COLORIMETRIC and ImageCms.INTENT_SATURATION.
- direction – One of ImageCms.DIRECTION_INPUT, ImageCms.DIRECTION_OUTPUT and ImageCms.DIRECTION_PROOF

Returns  Boolean if the intent and direction is supported.

1.3.4 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.new() and the ImageDraw module, among others.
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.
- Hue-Saturation-Value (HSV) functions, given as hsv(hue, saturation%, value%) where hue and saturation are the same as HSL, and value is between 0% and 100% (black=0%, normal=100%). For example, hsv(0, 100%, 100%) is pure red. This format is also known as Hue-Saturation-Brightness (HSB), and can be given as hsb(hue, saturation%, brightness%), where each of the values are used as they are in HSV.
- 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.

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.

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.

1.3.5 ImageDraw Module

The ImageDraw module provides 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 use with web pages.

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

Example: Draw a gray cross over an image

```python
from PIL import Image, ImageDraw

with Image.open("hopper.jpg") as im:
    draw = ImageDraw.Draw(im)
    draw.line((0, 0) + im.size, fill=128)
    draw.line((0, im.size[1], im.size[0], 0), fill=128)
```
(continues on next page)
# write to stdout
im.save(sys.stdout, "PNG")

## Concepts

### Coordinates

The graphics interface uses the same coordinate system as PIL itself, with (0, 0) in the upper left corner. Any pixels drawn outside of the image bounds will be discarded.

### Colors

To specify colors, you can use numbers or tuples just as you would use with PIL.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 Colored 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 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.

### Example: Draw Partial Opacity Text

```python
from PIL import Image, ImageDraw, ImageFont
# get an image
base = Image.open("Pillow/Tests/images/hopper.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()

Example: Draw Multiline Text

```python
from PIL import Image, ImageDraw, ImageFont

# create an image
out = Image.new("RGB", (150, 100), (255, 255, 255))

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

# draw multiline text
d.multiline_text((10,10), "Hello

World", font=fnt, fill=(0, 0, 0))
out.show()
```

Functions

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.

Methods

ImageDraw.getfont ()

Get the current default font.

Returns

An image font.

ImageDraw.arc(xy, start, end, fill=None, width=0)

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

Parameters

- **xy** – Two points to define the bounding box. Sequence of [(x0, y0), (x1, y1)] or [x0, y0, x1, y1], where x1 >= x0 and y1 >= y0.
• **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.
• **width** – The line width, in pixels.

New in version 5.3.0.

**ImageDraw.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.

**ImageDraw.chord**(xy, start, end, fill=None, outline=None, width=1)
Same as `arc()`, but connects the end points with a straight line.

**Parameters**
• **xy** – Two points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`, where `x1 >= x0` and `y1 >= y0`.
• **outline** – Color to use for the outline.
• **fill** – Color to use for the fill.
• **width** – The line width, in pixels.

New in version 5.3.0.

**ImageDraw.ellipse**(xy, fill=None, outline=None, width=1)
Draws an ellipse inside the given bounding box.

**Parameters**
• **xy** – Two points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`, where `x1 >= x0` and `y1 >= y0`.
• **outline** – Color to use for the outline.
• **fill** – Color to use for the fill.
• **width** – The line width, in pixels.

New in version 5.3.0.

**ImageDraw.line**(xy, fill=None, width=0, joint=None)
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.

New in version 1.1.5.

---

**Note:** This option was broken until version 1.1.6.
• **joint** – Joint type between a sequence of lines. It can be "curve", for rounded edges, or None.
  New in version 5.3.0.

**ImageDraw.pieslice** *(xy, start, end, fill=None, outline=None, width=1)*

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

**Parameters**

• **xy** – Two points to define the bounding box. Sequence of [(x0, y0), (x1, y1)] or [x0, y0, x1, y1], where x1 >= x0 and y1 >= y0.
• **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 fill.
• **outline** – Color to use for the outline.
• **width** – The line width, in pixels.
  New in version 5.3.0.

**ImageDraw.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.

**ImageDraw.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.

**ImageDraw.regular_polygon** *(bounding_circle, n_sides, rotation=0, fill=None, outline=None)*

Draws a regular polygon inscribed in bounding_circle, with n_sides, and rotation of rotation degrees.

**Parameters**

• **bounding_circle** – The bounding circle is a tuple defined by a point and radius. (e.g. bounding_circle=(x, y, r) or ((x, y), r)). The polygon is inscribed in this circle.
• **n_sides** – Number of sides (e.g. n_sides=3 for a triangle, 6 for a hexagon).
• **rotation** – Apply an arbitrary rotation to the polygon (e.g. rotation=90, applies a 90 degree rotation).
• **fill** – Color to use for the fill.
• **outline** – Color to use for the outline.

ImageDraw.*rectangle*(xy, fill=None, outline=None, width=1)

Draws a rectangle.

**Parameters**

• **xy** – Two 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.

• **width** – The line width, in pixels.

New in version 5.3.0.

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

**Warning:** This method is experimental.

Draw a shape.

ImageDraw.*text*(xy, text, fill=None, font=None, anchor=None, spacing=4, align='left', direction=None, features=None, language=None, stroke_width=0, stroke_fill=None)

Draws the string at the given position.

**Parameters**

• **xy** – Top left corner of the text.

• **text** – Text to be drawn. If it contains any newline characters, the text is passed on to *multiline_text()*.

• **fill** – Color to use for the text.

• **font** – An *ImageFont* instance.

• **spacing** – If the text is passed on to *multiline_text()* , the number of pixels between lines.

• **align** – If the text is passed on to *multiline_text()* , "left", "center" or "right".

• **direction** – Direction of the text. It can be "rtl" (right to left), "ltr" (left to right) or "ttb" (top to bottom). Requires libraqm.

New in version 4.2.0.

• **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example "dlig" or "ss01", but can be also used to turn off default font features, for example "-liga" to disable ligatures or "-kern" to disable kerning. To get all supported features, see OpenType docs. Requires libraqm.

New in version 4.2.0.

• **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code. Requires libraqm.
New in version 6.0.0.

- **stroke_width** – The width of the text stroke.

New in version 6.2.0.

- **stroke_fill** – Color to use for the text stroke. If not given, will default to the `fill` parameter.

New in version 6.2.0.

`ImageDraw.multiline_text(xy, text, fill=None, font=None, anchor=None, spacing=4, align='left', direction=None, features=None, language=None)`

Draws the string at the given position.

**Parameters**

- **xy** – Top left corner of the text.
- **text** – Text to be drawn.
- **fill** – Color to use for the text.
- **font** – An `ImageFont` instance.
- **spacing** – The number of pixels between lines.
- **align** – "left", "center" or "right".
- **direction** – Direction of the text. It can be "rtl" (right to left), "ltr" (left to right) or "ttb" (top to bottom). Requires libraqm.

New in version 4.2.0.

- **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example "dlig" or "ss01", but can be also used to turn off default font features, for example "-liga" to disable ligatures or "-kern" to disable kerning. To get all supported features, see OpenType docs. Requires libraqm.

New in version 4.2.0.

- **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code. Requires libraqm.

New in version 6.0.0.

`ImageDraw.textsize(text, font=None, spacing=4, direction=None, features=None, language=None, stroke_width=0)`

Return the size of the given string, in pixels.

**Parameters**

- **text** – Text to be measured. If it contains any newline characters, the text is passed on to `multiline_textsize()`.

- **font** – An `ImageFont` instance.

- **spacing** – If the text is passed on to `multiline_textsize()`, the number of pixels between lines.

- **direction** – Direction of the text. It can be "rtl" (right to left), "ltr" (left to right) or "ttb" (top to bottom). Requires libraqm.

New in version 4.2.0.
• **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example "dlig" or "ss01", but can be also used to turn off default font features, for example "-liga" to disable ligatures or "-kern" to disable kerning. To get all supported features, see OpenType docs. Requires libraqm.

  New in version 4.2.0.

• **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code. Requires libraqm.

  New in version 6.0.0.

• **stroke_width** – The width of the text stroke.

  New in version 6.2.0.

**ImageDraw.multiline_textsize**(text, font=None, spacing=4, direction=None, features=None, language=None, stroke_width=0)

Return the size of the given string, in pixels.

**Parameters**

• **text** – Text to be measured.

• **font** – An ImageFont instance.

• **spacing** – The number of pixels between lines.

• **direction** – Direction of the text. It can be "rtl" (right to left), "ltr" (left to right) or "ttb" (top to bottom). Requires libraqm.

  New in version 4.2.0.

• **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example "dlig" or "ss01", but can be also used to turn off default font features, for example "-liga" to disable ligatures or "-kern" to disable kerning. To get all supported features, see OpenType docs. Requires libraqm.

  New in version 4.2.0.

• **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code. Requires libraqm.

  New in version 6.0.0.

• **stroke_width** – The width of the text stroke.

  New in version 6.2.0.

**PIL.ImageDraw.getdraw**(im=None, hints=None)

**Warning:** This method is experimental.

A more advanced 2D drawing interface for PIL images, based on the WCK interface.

**Parameters**
• **im** – The image to draw in.
• **hints** – An optional list of hints.

**Returns** A (drawing context, drawing resource factory) tuple.

```python
PIL.ImageDraw.floodfill(image, xy, value, border=None, thresh=0)
```

**Warning:** This method is experimental.

Fills a bounded region with a given color.

**Parameters**

• **image** – Target image.
• **xy** – Seed position (a 2-item coordinate tuple).
• **value** – Fill color.
• **border** – Optional border value. If given, the region consists of pixels with a color different from the border color. If not given, the region consists of pixels having the same color as the seed pixel.
• **thresh** – Optional threshold value which specifies a maximum tolerable difference of a pixel value from the ‘background’ in order for it to be replaced. Useful for filling regions of non-homogeneous, but similar, colors.

### 1.3.6 ImageEnhance Module

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

**Example: Vary the sharpness of an image**

```python
from PIL import ImageEnhance
enhancer = ImageEnhance.Sharpness(image)

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

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

**Classes**

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

```python
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.

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.

1.3.7 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.

Example: Parse an image

```
from PIL import ImageFile

fp = open("hopper.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")
```
Classes

```python
class PIL.ImageFile.Parser
    Incremental image parser. This class implements the standard feed/close consumer interface.

    close()
    (Consumer) Close the stream.

        Returns  An image object.

        Raises OSError – 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 OSError – 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.

class PIL.ImageFile.PyDecoder
    Python implementation of a format decoder. Override this class and add the decoding logic in the decode() method.

    See Writing Your Own File Decoder in Python

    cleanup()
    Override to perform decoder specific cleanup

        Returns  None

    decode(buffer)
    Override to perform the decoding process.

        Parameters buffer – A bytes object with the data to be decoded.

        Returns A tuple of (bytes consumed, errcode). If finished with decoding return <0 for the bytes consumed. Err codes are from ImageFile.ERRORS.

    init(args)
    Override to perform decoder specific initialization

        Parameters args – Array of args items from the tile entry

        Returns  None

    set_as_raw(data, rawmode=None)
    Convenience method to set the internal image from a stream of raw data

        Parameters

            • data – Bytes to be set

            • rawmode – The rawmode to be used for the decoder. If not specified, it will default to the mode of the image

        Returns  None

    setfd(fd)
    Called from ImageFile to set the python file-like object

        Parameters fd – A python file-like object
```
Returns None

setimage(im, extents=None)
Called from ImageFile to set the core output image for the decoder

Parameters
• im – A core image object
• extents – a 4 tuple of (x0, y0, x1, y1) defining the rectangle for this tile

Returns None

class PIL.ImageFile.ImageFile
Bases: PIL.Image.Image

Base class for image file format handlers.

tile
A list of tile descriptors, or None

get_format_mimetype()

verify()
Check file integrity

load()
Load image data based on tile list

load_prepare()

load_end()

class PIL.ImageFile.StubImageFile
Bases: PIL.ImageFile.ImageFile

Base class for stub image loaders.

A stub loader is an image loader that can identify files of a certain format, but relies on external code to load the file.

load()
Load image data based on tile list

Constants

PIL.ImageFile.LOAD_TRUNCATED_IMAGES = False
Whether or not to load truncated image files. User code may change this.

PIL.ImageFile.ERRORS
Dict of known error codes returned from PyDecoder.decode().
1.3.8 ImageFilter Module

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

Example: Filter an image

```python
from PIL import ImageFilter

im1 = im.filter(ImageFilter.BLUR)
im2 = im.filter(ImageFilter.MinFilter(3))
im3 = im.filter(ImageFilter.MinFilter)  # same as MinFilter(3)
```

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
- SHARPEN
- SMOOTH
- SMOOTH_MORE

class PIL.ImageFilter.Color3DLUT(size, table, channels=3, target_mode=None, **kwargs)

Three-dimensional color lookup table.

Transforms 3-channel pixels using the values of the channels as coordinates in the 3D lookup table and interpolating the nearest elements.

This method allows you to apply almost any color transformation in constant time by using pre-calculated decimated tables.

New in version 5.2.0.

Parameters

- **size** – Size of the table. One int or tuple of (int, int, int). Minimal size in any dimension is 2, maximum is 65.
- **table** – Flat lookup table. A list of `channels * size**3` float elements or a list of `size**3` channels-sized tuples with floats. Channels are changed first, then first dimension, then second, then third. Value 0.0 corresponds lowest value of output, 1.0 highest.
- **channels** – Number of channels in the table. Could be 3 or 4. Default is 3.
- **target_mode** – A mode for the result image. Should have not less than `channels` channels. Default is `None`, which means that mode wouldn’t be changed.
classmethod generate(size, callback, channels=3, target_mode=None)
Generates new LUT using provided callback.

Parameters

• size – Size of the table. Passed to the constructor.

• callback – Function with three parameters which correspond three color channels. Will be called $size^3$ times with values from 0.0 to 1.0 and should return a tuple with $channels$ elements.

• channels – The number of channels which should return callback.

• target_mode – Passed to the constructor of the resulting lookup table.

class PIL.ImageFilter.BoxBlur(radius)
Blurs the image by setting each pixel to the average value of the pixels in a square box extending radius pixels in each direction. Supports float radius of arbitrary size. Uses an optimized implementation which runs in linear time relative to the size of the image for any radius value.

Parameters radius – Size of the box in one direction. Radius 0 does not blur, returns an identical image. Radius 1 takes 1 pixel in each direction, i.e. 9 pixels in total.

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.

```python
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.
```

```python
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.
```

```python
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.
```

```python
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.
```

```python
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.
```

```python
class PIL.ImageFilter.Filter
An abstract mixin used for filtering images (for use with `filter()`).

Implementors must provide the following method:

```python
    filter(self, image)
    Applies a filter to a single-band image, or a single band of an image.

Returns A filtered copy of the image.
```

```python
class PIL.ImageFilter.MultibandFilter
An abstract mixin used for filtering multi-band images (for use with `filter()`).

Implementors must provide the following method:

```python
    filter(self, image)
    Applies a filter to a multi-band image.

Returns A filtered copy of the image.
```
1.3.9 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.ImageDraw.text()` method.

PIL uses its own font file format to store bitmap fonts. You can use the `pilfont` utility from pillow-scripts 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.

**Example**

```python
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)
```

**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** OSError – 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** OSError – If the file could not be read.

PIL.ImageFont.truetype(font=None, size=10, index=0, encoding='', layout_engine=None)

- Load a TrueType or OpenType font from a file or file-like object, and create a font object. This function loads a font object from the given file or file-like object, and creates a font object for a font of the given size.

  Pillow uses FreeType to open font files. If you are opening many fonts simultaneously on Windows, be aware that Windows limits the number of files that can be open in C at once to 512. If you approach that limit, an OSError may be thrown, reporting that FreeType “cannot open resource”.

  This function requires the _imagingft service.
Parameters

- **font** – A filename or file-like object containing a TrueType font. If the file is not found in this filename, the loader may also search in other directories, such as the `fonts/` directory on Windows or `/Library/Fonts/`, `/System/Library/Fonts/` and `~/Library/Fonts/` on macOS.

- **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). Possible encodings include (see the FreeType documentation for more information):
  - "unic" (Unicode)
  - "symb" (Microsoft Symbol)
  - "ADOB" (Adobe Standard)
  - "ADBE" (Adobe Expert)
  - "ADBC" (Adobe Custom)
  - "armn" (Apple Roman)
  - "jis" (Shift JIS)
  - "gb" (PRC)
  - "big5"
  - "wans" (Extended Wansung)
  - "joha" (Johab)
  - "lat1" (Latin-1)

This specifies the character set to use. It does not alter the encoding of any text provided in subsequent operations.

- **layout_engine** – Which layout engine to use, if available: `ImageFont.LAYOUT_BASIC` or `ImageFont.LAYOUT_RAQM`.

  You can check support for Raqm layout using `PIL.features.check_feature()` with `feature="raqm"`.

  New in version 4.2.0.

**Returns** A font object.

**Raises** `OSError` – 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.
Methods

```python
class PIL.ImageFont.ImageFont
    PIL font wrapper

    getmask (text, mode='', *args, **kwargs)
    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.

gensize (text, *args, **kwargs)
    Returns width and height (in pixels) of given text.

    Parameters text – Text to measure.

    Returns (width, height)
```

```python
class PIL.ImageFont.FreeTypeFont (font=None, size=10, index=0, encoding='', layout_engine=None)
    FreeType font wrapper (requires _imagingft service)

    font_variant (font=None, size=None, index=None, encoding=None, layout_engine=None)
    Create a copy of this FreeTypeFont object, using any specified arguments to override the settings.

    Parameters are identical to the parameters used to initialize this object.

    Returns
    A FreeTypeFont object.

get_variation_axes ()
    Returns
    A list of the axes in a variation font.

    Raises OSError – If the font is not a variation font.

get_variation_names ()
    Returns
    A list of the named styles in a variation font.

    Raises OSError – If the font is not a variation font.

gemask (text, mode='', direction=None, features=None, language=None, stroke_width=0)
    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.

- **direction** – Direction of the text. It can be ‘rtl’ (right to left), ‘ltr’ (left to right) or ‘ttb’ (top to bottom). Requires libraqm.

New in version 4.2.0.

- **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example ‘dlig’ or ‘ss01’, but can be also used to turn off default font features for example ‘-liga’ to disable ligatures or ‘-kern’ to disable kerning. To get all supported features, see https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist Requires libraqm.

New in version 4.2.0.

- **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code <https://www.w3.org/International/articles/language-tags/> Requires libraqm.

New in version 6.0.0.

- **stroke_width** – The width of the text stroke.

New in version 6.2.0.

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

```
getmask2 (text, mode='', fill=<built-in function fill>, direction=None, features=None, language=None, stroke_width=0, *args, **kwargs)
```

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.

- **direction** – Direction of the text. It can be ‘rtl’ (right to left), ‘ltr’ (left to right) or ‘ttb’ (top to bottom). Requires libraqm.

New in version 4.2.0.

- **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example ‘dlig’ or ‘ss01’, but can be also used to turn off default font features for example ‘-liga’ to disable ligatures or ‘-kern’ to disable kerning. To get all supported features, see https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist Requires libraqm.

New in version 4.2.0.

- **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code <https://www.w3.org/International/articles/language-tags/> Requires libraqm.

New in version 6.0.0.
- **stroke_width** – The width of the text stroke.
  
  New in version 6.2.0.

  Returns A tuple of an internal PIL storage memory instance as defined by the `PIL.Image.core` interface module, and the text offset, the gap between the starting coordinate and the first marking

  **getmetrics()**

  Returns A tuple of the font ascent (the distance from the baseline to the highest outline point) and descent (the distance from the baseline to the lowest outline point, a negative value)

  **getname()**

  Returns A tuple of the font family (e.g. Helvetica) and the font style (e.g. Bold)

  **getoffset(text)**

  Returns the offset of given text. This is the gap between the starting coordinate and the first marking. Note that this gap is included in the result of `getsize()`.

  Parameters
  
  **text** – Text to measure.

  Returns A tuple of the x and y offset

  **getsize(text, direction=None, features=None, language=None, stroke_width=0)**

  Returns width and height (in pixels) of given text if rendered in font with provided direction, features, and language.

  Parameters
  
  • **text** – Text to measure.
  
  • **direction** – Direction of the text. It can be ‘rtl’ (right to left), ‘ltr’ (left to right) or ‘ttb’ (top to bottom). Requires libraqm.
    
    New in version 4.2.0.
  
  • **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example ‘dlig’ or ‘ss01’, but can be also used to turn off default font features for example ‘-liga’ to disable ligatures or ‘-kern’ to disable kerning. To get all supported features, see [https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist](https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist) Requires libraqm.
    
    New in version 4.2.0.
  
  • **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a [BCP 47 language code](https://www.w3.org/International/articles/language-tags/) Requires libraqm.
    
    New in version 6.0.0.
  
  • **stroke_width** – The width of the text stroke.
    
    New in version 6.2.0.

  Returns (width, height)

  **getsize_multiline(text, direction=None, spacing=4, features=None, language=None, stroke_width=0)**

  Returns width and height (in pixels) of given text if rendered in font with provided direction, features, and language, while respecting newline characters.

  Parameters
• **text** – Text to measure.

• **direction** – Direction of the text. It can be ‘rtl’ (right to left), ‘ltr’ (left to right) or ‘ttb’ (top to bottom). Requires libraqm.

• **spacing** – The vertical gap between lines, defaulting to 4 pixels.

• **features** – A list of OpenType font features to be used during text layout. This is usually used to turn on optional font features that are not enabled by default, for example ‘dlig’ or ‘ss01’, but can be also used to turn off default font features for example ‘liga’ to disable ligatures or ‘kern’ to disable kerning. To get all supported features, see [https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist](https://docs.microsoft.com/en-us/typography/opentype/spec/featurelist) Requires libraqm.

• **language** – Language of the text. Different languages may use different glyph shapes or ligatures. This parameter tells the font which language the text is in, and to apply the correct substitutions as appropriate, if available. It should be a BCP 47 language code <https://www.w3.org/International/articles/language-tags/> Requires libraqm.

  New in version 6.0.0.

• **stroke_width** – The width of the text stroke.

  New in version 6.2.0.

  Returns (width, height)

  `set_variation_by_axes` *(axes)*

  Parameters **axes** – A list of values for each axis.

  Raises **OSError** – If the font is not a variation font.

  `set_variation_by_name` *(name)*

  Parameters **name** – The name of the style.

  Raises **OSError** – If the font is not a variation font.

  `class PIL.ImageFont.TransposedFont (font, orientation=None)`

  Wrapper for writing rotated or mirrored text

**Constants**

PIL.ImageFont.LAYOUT_BASIC

Use basic text layout for TrueType font. Advanced features such as text direction are not supported.

PIL.ImageFont.LAYOUT_RAQM

Use Raqm text layout for TrueType font. Advanced features are supported.

  Requires Raqm, you can check support using `PIL.features.check_feature()` with `feature="raqm"`. 

106 Chapter 1. Overview
1.3.10 ImageGrab Module

The ImageGrab module can be used to copy the contents of the screen or the clipboard to a PIL image memory. New in version 1.1.3.

```python
PIL.ImageGrab.grab(bbox=None, include_layered_windows=False, all_screens=False, xdisplay=None)
```

Take a snapshot of the screen. The pixels inside the bounding box are returned as an “RGBA” on macOS, or an “RGB” image otherwise. If the bounding box is omitted, the entire screen is copied.

New in version 1.1.3: (Windows), 3.0.0 (macOS), 7.1.0 (Linux (X11))

**Parameters**

- **bbox** – What region to copy. Default is the entire screen. Note that on Windows OS, the top-left point may be negative if all_screens=True is used.

- **include_layered_windows** – Includes layered windows. Windows OS only.
  New in version 6.1.0.

- **all_screens** – Capture all monitors. Windows OS only.
  New in version 6.2.0.

- **xdisplay** – X11 Display address. Pass None to grab the default system screen. Pass "" to grab the default X11 screen on Windows or macOS.

  You can check X11 support using PIL.features.check_feature() with feature="xcb".

  New in version 7.1.0.

**Returns** An image

```
PIL.ImageGrab.grabclipboard()
```

Take a snapshot of the clipboard image, if any. Only macOS and Windows are currently supported.

New in version 1.1.4: (Windows), 3.3.0 (macOS)

**Returns**

On Windows, 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.

On Mac, an image, or None if the clipboard does not contain image data.

1.3.11 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.
Example: Using the `ImageMath` module

```python
don PIL 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.

**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.

1.3.12 ImageMorph Module

The `ImageMorph` module provides morphology operations on images.

```python
class PIL.ImageMorph.LutBuilder (patterns=None, op_name=None)
Bases: object

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:

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

Example:

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

```python
def add_patterns(patterns)
build_default_lut()
build_lut()
```

```python
class PIL.ImageMorph.MorphOp(lut=None, op_name=None, patterns=None)
```

```python
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. See Coordinate System.

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. See Coordinate System.

save_lut(filename)
Save an operator to an mrl file

set_lut(lut)
Set the lut from an external source

1.3.13 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.

New in version 1.1.3.

```python
PIL.ImageOps.autocontrast(image, cutoff=0, ignore=None, mask=None)
```

Maximize (normalize) image contrast. This function calculates a histogram of the input image (or mask region), 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** – The percent to cut off from the histogram on the low and high ends. Either a tuple of (low, high), or a single number for both.

• **ignore** – The background pixel value (use None for no background).

• **mask** – Histogram used in contrast operation is computed using pixels within the mask. If no mask is given the entire image is used for histogram computation.

Returns An image.

PIL.ImageOps.colorize(image, black, white, mid=None, blackpoint=0, whitepoint=255, midpoint=127)

Colorize grayscale image. This function calculates a color wedge which maps all black pixels in the source image to the first color and all white pixels to the second color. If mid is specified, it uses three-color mapping. The black and white arguments should be RGB tuples or color names; optionally you can use three-color mapping by also specifying mid. Mapping positions for any of the colors can be specified (e.g. blackpoint), where these parameters are the integer value corresponding to where the corresponding color should be mapped. These parameters must have logical order, such that blackpoint <= midpoint <= whitepoint (if mid is specified).

Parameters

• **image** – The image to colorize.

• **black** – The color to use for black input pixels.

• **white** – The color to use for white input pixels.

• **mid** – The color to use for midtone input pixels.

• **blackpoint** – an int value [0, 255] for the black mapping.

• **whitepoint** – an int value [0, 255] for the white mapping.

• **midpoint** – an int value [0, 255] for the midtone mapping.

Returns An image.

PIL.ImageOps.pad(image, size, method=3, color=None, centering=0.5, 0.5)

Returns a sized and padded version of the image, expanded to fill the requested aspect ratio and size.

Parameters

• **image** – The image to size and crop.

• **size** – The requested output size in pixels, given as a (width, height) tuple.

• **method** – What resampling method to use. Default is PIL.Image.BICUBIC. See Filters.

• **color** – The background color of the padded image.

• **centering** – Control the position of the original image within the padded version.

    (0.5, 0.5) will keep the image centered (0, 0) will keep the image aligned to the top left
    (1, 1) will keep the image aligned to the bottom right

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:

\texttt{crop()}

Parameters

- \texttt{image} – The image to crop.
- \texttt{border} – The number of pixels to remove.

Returns

\texttt{PIL.ImageOps.scale(image, factor, resample=3)}

Returns a rescaled image by a specific factor given in parameter. A factor greater than 1 expands the image, between 0 and 1 contracts the image.

Parameters

- \texttt{image} – The image to rescale.
- \texttt{factor} – The expansion factor, as a float.
- \texttt{resample} – What resampling method to use. Default is \texttt{PIL.Image.BICUBIC}. See Filters.

Returns

\texttt{PIL.ImageOps.deform(image, deformer, resample=2)}

Deform the image.

Parameters

- \texttt{image} – The image to deform.
- \texttt{deformer} – A deformer object. Any object that implements a \texttt{getmesh} method can be used.
- \texttt{resample} – An optional resampling filter. Same values possible as in the \texttt{PIL.Image.transform} function.

Returns

\texttt{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

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

Returns

\texttt{PIL.ImageOps.expand(image, border=0, fill=0)}

Add border to the image

Parameters

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

Returns

An image.
PIL.ImageOps.fit(image, size, method=3, 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**

- image – The image to size and crop.
- size – The requested output size in pixels, given as a (width, height) tuple.
- method – What resampling method to use. Default is PIL.Image.BICUBIC. See Filters.
- 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). Cannot be greater than or equal to 0.5.
- 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.

PIL.ImageOps.exif_transpose(image)
If an image has an EXIF Orientation tag, return a new image that is transposed accordingly. Otherwise, return a copy of the image.

Parameters image – The image to transpose.

Returns An image.

1.3.14 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

Parameters

- mode – The mode to use for the Palette. See: Modes. Defaults to “RGB”
- palette – An optional palette. If given, it must be a bytearray, an array or a list of ints between 0-255 and of length size times the number of colors in mode. The list must be aligned by channel (All R values must be contiguous in the list before G and B values.) Defaults to 0 through 255 per channel.
- size – An optional palette size. If given, it cannot be equal to or greater than 256. Defaults to 0.

gtcolor(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.

### 1.3.15 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 \([x, y], \ldots\). If this argument is True, it returns a flat list \([x, y, \ldots]\) 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:

\[
\begin{align*}
    x_{\text{Out}} &= x_{\text{In}} \times a + y_{\text{In}} \times b + c \\
    y_{\text{Out}} &= x_{\text{In}} \times d + y_{\text{In}} \times e + f
\end{align*}
\]

1.3.16 ImageQt Module

The ImageQt module contains support for creating PyQt5 or PySide2 QImage objects from PIL images.

New in version 1.1.6.

class PIL.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 PyQt5/PySide2 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.

1.3.17 ImageSequence Module

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

Extracting frames from an animation

```python
from PIL import Image, ImageSequence

with Image.open("animation.fli") as im:
    index = 1
    for frame in ImageSequence.Iterator(im):
        frame.save(f"frame{index}.png")
        index += 1
```

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.
1.3.18 ImageShow Module

The ImageShow Module is used to display images. All default viewers convert the image to be shown to PNG format.

PIL.ImageShow.show(image, title=None, **options)
Display a given image.

Parameters
- image – An image object.
- title – Optional title. Not all viewers can display the title.
- **options – Additional viewer options.

Returns True if a suitable viewer was found, False otherwise.

class PIL.ImageShow.WindowsViewer
The default viewer on Windows is the default system application for PNG files.

class PIL.ImageShow.MacViewer
The default viewer on MacOS using Preview.app.

class PIL.ImageShow.UnixViewer
The following viewers may be registered on Unix-based systems, if the given command is found:

    class DisplayViewer
        The ImageMagick display command.
    
    class EogViewer
        The GNOME Image Viewer eog command.
    
    class XVViewer
        The X Viewer xv command. This viewer supports the title parameter.

PIL.ImageShow.register(viewer, order=1)
The register() function is used to register additional viewers.

Parameters
- viewer – The viewer to be registered.
- order – Zero or a negative integer to prepend this viewer to the list, a positive integer to append it.

class PIL.ImageShow.Viewer
Base class for viewers.

show(image, **options)
The main function for displaying an image. Converts the given image to the target format and displays it.

format = None
The format to convert the image into.

options = {}
Additional options used to convert the image.

get_format(image)
Return format name, or None to save as PGM/PPM.

get_command(file, **options)
Returns the command used to display the file. Not implemented in the base class.
save_image(image)
    Save to temporary file and return filename.

show_image(image, **options)
    Display the given image.

show_file(file, **options)
    Display the given file.

### 1.3.19 ImageStat Module

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

```python
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.

**Note:** This relies on the `histogram()` method, and simply returns the low and high bins used. This is correct for images with 8 bits per channel, but fails for other modes such as I or F. Instead, use `getextrema()` to return per-band extrema for the image. This is more correct and efficient because, for non-8-bit modes, the histogram method uses `getextrema()` to determine the bins used.

**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.
1.3.20 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. See Coordinate System. 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.
1.3.21 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:

```python
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.

```python
def 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.
```

```python
def 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 CDC.GetHandleAttrib() to get a suitable handle.
```

```python
def frombytes(buffer)
    Load display memory contents from byte data.

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

```python
def 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. See Coordinate System. If None is given instead of a tuple, all of the image is assumed.
```

```python
def 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.

```python
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.
```

### 1.3.22 ExifTags Module

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

**PIL.ExifTags.TAGS**: dict

The TAG dictionary maps 16-bit integer EXIF tag enumerations to descriptive string names. For instance:

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

**PIL.ExifTags.GPSTAGS**: dict

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

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

### 1.3.23 TiffTags Module

The `TiffTags` module exposes many of the standard TIFF metadata tag numbers, names, and type information.

**PIL.TiffTags.lookup**(tag)

**Parameters**

- `tag` – Integer tag number

**Returns**

Taginfo namedtuple, From the `TAGS_V2` info if possible, otherwise just populating the value and name from `TAGS`. If the tag is not recognized, “unknown” is returned for the name

New in version 3.1.0.

```python
class PIL.TiffTags.TagInfo

    __init__(self, value=None, name='unknown', type=None, length=0, enum=None)
```

1.3. Reference
• **value** – Integer Tag Number
• **name** – Tag Name
• **type** – Integer type from `PIL.TiffTags.TYPES`
• **length** – Array length: 0 == variable, 1 == single value, n = fixed
• **enum** – Dict of name:integer value options for an enumeration

```python
cvt_enum(self, value)
```

**Parameters**
- **value** – The enumerated value name

**Returns**
The integer corresponding to the name.

New in version 3.0.0.

**PIL.TiffTags.TAGS_V2: dict**
The `TAGS_V2` dictionary maps 16-bit integer tag numbers to `PIL.TiffTags.TagInfo` tuples for metadata fields defined in the TIFF spec.

New in version 3.0.0.

**PIL.TiffTags.TAGS: dict**
The `TAGS` dictionary maps 16-bit integer TIFF tag number to descriptive string names. For instance:

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

This dictionary contains a superset of the tags in `TAGS_V2`, common EXIF tags, and other well known metadata tags.

**PIL.TiffTags.TYPES: dict**
The `TYPES` dictionary maps the TIFF type short integer to a human readable type name.

**PIL.TiffTags.LIBTIFF_CORE: list**
A list of supported tag IDs when writing using LibTIFF.

### 1.3.24 JpegPresets Module

JPEG quality settings equivalent to the Photoshop settings. Can be used when saving JPEG files.

The following presets are available by default: `web_low`, `web_medium`, `web_high`, `web_very_high`, `web_maximum`, `low`, `medium`, `high`, `maximum`. More presets can be added to the `presets` dict if needed.

To apply the preset, specify:

```python
quality="preset_name"
```

To apply only the quantization table:

```python
qtables="preset_name"
```

To apply only the subsampling setting:

```python
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.: https://en.wikipedia.org/wiki/Chroma_subsampling)

Possible subsampling values are 0, 1 and 2 that correspond to 4:4:4, 4:2:2 and 4:2:0.

You can get the subsampling of a JPEG with the JpegImagePlugin.get_sampling() function.

In JPEG compressed data a JPEG marker is used instead of an EXIF tag. (ref.: https://www.exiv2.org/tags.html)

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.: https://en.wikipedia.org/wiki/Quantization_matrix#Quantization_matrices, https://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() function.


PIL.JpegPresets.preset: dict
A dictionary of all supported presets.

1.3.25 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 fp 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.

### 1.3.26 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.

---

**Example**

The following script loads an image, accesses one pixel from it, then changes it.

```python
from PIL import Image

with Image.open('hopper.jpg') as im:
    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)
```

Access using negative indexes is also possible.

```python
px[-1,-1] = (0,0,0)
print (px[-1,-1])
```
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).
• color – The pixel value according to its mode. e.g. tuple (r, g, b) for RGB mode

__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. In addition to this, RGB and RGBA tuples are accepted for P images.

Parameters

• xy – The pixel coordinate, given as (x, y).
• color – The pixel value according to its mode. e.g. tuple (r, g, b) for RGB mode

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.

1.3.27 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.
Example

The following script loads an image, accesses one pixel from it, then changes it.

```python
from PIL import Image
with Image.open('hopper.jpg') as im:
    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)
```

Access using negative indexes is also possible.

```python
px[-1,-1] = (0,0,0)
print(px[-1,-1])
```

PyAccess Class

```python
class PIL.PyAccess.PyAccess

    getpixel(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
    Parameters
    xy -- The pixel coordinate, given as (x, y). See Coordinate System.
    Returns
    a pixel value for single band images, a tuple of pixel values for multiband images.

    putpixel(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). See Coordinate System.
    color -- The pixel value.
```

1.3.28 features Module

The PIL.features module can be used to detect which Pillow features are available on your system.

```python
PIL.features.pilinfo(out=None, supported_formats=True)
Prints information about this installation of Pillow. This function can be called with `python -m PIL`.
```

Parameters

- `out` -- The output stream to print to. Defaults to sys.stdout if None.
- `supported_formats` -- If True, a list of all supported image file formats will be printed.

```python
PIL.features.check(feature)
```

Parameters

- `feature` -- A module, codec, or feature name.
Returns True if the module, codec, or feature is available, False or None otherwise.

PIL.features.version (feature)

Parameters feature – The module, codec, or feature to check for.

Returns The version number as a string, or None if unknown or not available.

PIL.features.get_supported ()

Returns A list of all supported modules, features, and codecs.

Modules

Support for the following modules can be checked:

• pil: The Pillow core module, required for all functionality.
• tkinter: Tkinter support. Version number not available.
• freetype2: FreeType font support via PIL.ImageFont.truetype().
• littlecms2: LittleCMS 2 support via PIL.ImageCms.
• webp: WebP image support.

PIL.features.check_module (feature)

Checks if a module is available.

Parameters feature – The module to check for.

Returns True if available, False otherwise.

Raises ValueError – If the module is not defined in this version of Pillow.

PIL.features.version_module (feature)

Parameters feature – The module to check for.

Returns The loaded version number as a string, or None if unknown or not available.

Raises ValueError – If the module is not defined in this version of Pillow.

PIL.features.get_supported_modules ()

Returns A list of all supported modules.

Codecs

Support for these is only checked during Pillow compilation. If the required library was uninstalled from the system, the pil core module may fail to load instead. Except for jpg, the version number is checked at run-time.

Support for the following codecs can be checked:

• jpg: (compile time) Libjpeg support, required for JPEG based image formats. Only compile time version number is available.
• jpg_2000: (compile time) OpenJPEG support, required for JPEG 2000 image formats.
• zlib: (compile time) Zlib support, required for zlib compressed formats, such as PNG.
• libtiff: (compile time) LibTIFF support, required for TIFF based image formats.

PIL.features.check_codec (feature)

Checks if a codec is available.
Parameters `feature` – The codec to check for.

Returns `True` if available, `False` otherwise.

Raises `ValueError` – If the codec is not defined in this version of Pillow.

PIL.features.version_codec(`feature`)  
Parameters `feature` – The codec to check for.

Returns The version number as a string, or `None` if not available. Checked at compile time for `jpg`, run-time otherwise.

Raises `ValueError` – If the codec is not defined in this version of Pillow.

PIL.features.get_supported_codecs()  
Returns A list of all supported codecs.

Features

Some of these are only checked during Pillow compilation. If the required library was uninstalled from the system, the relevant module may fail to load instead. Feature version numbers are available only where stated.

Support for the following features can be checked:

- `libjpeg_turbo`: (compile time) Whether Pillow was compiled against the libjpeg-turbo version of libjpeg. Compile-time version number is available.

- `transp_webp`: Support for transparency in WebP images.

- `webp_mux`: (compile time) Support for EXIF data in WebP images.

- `webp_anim`: (compile time) Support for animated WebP images.

- `raqm`: Raqm library, required for `ImageFont.LAYOUT_RAQM` in `PIL.ImageFont.truetype()`. Run-time version number is available for Raqm 0.7.0 or newer.

- `libimagequant`: (compile time) ImageQuant quantization support in `PIL.Image.Image.quantize()`. Run-time version number is available.

- `xcb`: (compile time) Support for X11 in `PIL.ImageGrab.grab()` via the XCB library.

PIL.features.check_feature(`feature`)  
Checks if a feature is available.

Parameters `feature` – The feature to check for.

Returns `True` if available, `False` if unavailable, `None` if unknown.

Raises `ValueError` – If the feature is not defined in this version of Pillow.

PIL.features.version_feature(`feature`)  
Parameters `feature` – The feature to check for.

Returns The version number as a string, or `None` if not available.

Raises `ValueError` – If the feature is not defined in this version of Pillow.

PIL.features.get_supported_features()  
Returns A list of all supported features.
1.3.29 PIL Package (autodoc of remaining modules)

Reference for modules whose documentation has not yet been ported or written can be found here.

**PIL Module**

```python
exception PIL.UnidentifiedImageError
    Bases: OSError
          Raised in PIL.Image.open() if an image cannot be opened and identified.
```

**BdfFontFile Module**

Parse X Bitmap Distribution Format (BDF)

```python
class PIL.BdfFontFile.BdfFontFile(fp)
    Bases: PIL.FontFile.FontFile
          Font file plugin for the X11 BDF format.
```

**ContainerIO Module**

```python
class PIL.ContainerIO.ContainerIO(file, offset, length)
    Bases: object
    A file object that provides read access to a part of an existing file (for example a TAR file).
```

```python
isatty()
read(n=0)
    Read data.
    Parameters n – Number of bytes to read. If omitted or zero, read until end of region.
    Returns An 8-bit string.
```

```python
readline()
    Read a line of text.
    Returns An 8-bit string.
```

```python
readlines()
    Read multiple lines of text.
    Returns A list of 8-bit strings.
```

```python
seek(offset, mode=0)
    Move file pointer.
    Parameters
    • offset – Offset in bytes.
    • mode – Starting position. Use 0 for beginning of region, 1 for current offset, and 2 for end of region. You cannot move the pointer outside the defined region.
```

```python
tell()
    Get current file pointer.
```
Returns Offset from start of region, in bytes.

FontFile Module

class PIL.FontFile.FontFile
    Bases: object
    Base class for raster font file handlers.
    bitmap = None
    compile()
        Create metrics and bitmap
    save(filename)
        Save font
PIL.FontFile.puti16(fp, values)
    Write network order (big-endian) 16-bit sequence

GdImageFile Module

Note: This format cannot be automatically recognized, so the class is not registered for use with PIL.Image.open(). To open a gd file, use the PIL.GdImageFile.open() function instead.

Warning: THE GD FORMAT IS NOT DESIGNED FOR DATA INTERCHANGE. This implementation is provided for convenience and demonstrational purposes only.

class PIL.GdImageFile.GdImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    Image plugin for the GD uncompressed format. Note that this format is not supported by the standard PIL.Image.open() function. To use this plugin, you have to import the PIL.GdImageFile module and use the PIL.GdImageFile.open() function.
    format = 'GD'
    format_description = 'GD uncompressed images'
PIL.GdImageFile.open(fp, mode='r')
    Load texture from a GD image file.
    Parameters
        • filename – GD file name, or an opened file handle.
        • mode – Optional mode. In this version, if the mode argument is given, it must be “r”.
    Returns An image instance.
    Raises OSError – If the image could not be read.
**GimpGradientFile Module**

Stuff to translate curve segments to palette values (derived from the corresponding code in GIMP, written by Federico Mena Quintero. See the GIMP distribution for more information.)

```python
PIL.GimpGradientFile.EPSILON = 1e-10
class PIL.GimpGradientFile.GimpGradientFile(fp)
    Bases: PIL.GimpGradientFile.GradientFile
    File handler for GIMP’s gradient format.
class PIL.GimpGradientFile.GradientFile
    Bases: object
    getpalette(entries=256)
    gradient = None
PIL.GimpGradientFile.SEVERTS = [function.linear, function.curved, function.sine, function.sphere_increasing, function.sphere_decreasing]
PIL.GimpGradientFile.linear(middle, pos)
PIL.GimpGradientFile.sphere_decreasing(middle, pos)
PIL.GimpGradientFile.sphere_increasing(middle, pos)
```

**GimpPaletteFile Module**

```python
class PIL.GimpPaletteFile.GimpPaletteFile(fp)
    Bases: object
    File handler for GIMP’s palette format.
    getpalette()
    rawmode = 'RGB'
```

**ImageDraw2 Module**

(Experimental) WCK-style drawing interface operations

See also:

```python
PIL.ImageDraw

class PIL.ImageDraw2.Pen(color, width=1, opacity=255)
    Bases: object
    Stores an outline color and width.
class PIL.ImageDraw2.Brush(color, opacity=255)
    Bases: object
    Stores a fill color

class PIL.ImageDraw2.Font(color, file, size=12)
    Bases: object
    Stores a TrueType font and color
```
class PIL.ImageDraw2.Draw(image, size=None, color=None)
Bases: object

(Experimental) WCK-style drawing interface

flush()

render(op, xy, pen, brush=None)

settransform(offset)
Sets a transformation offset.

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

See also:
PIL.ImageDraw.ImageDraw.arc()

chord(xy, start, end, *options)
Same as arc(), but connects the end points with a straight line.

See also:
PIL.ImageDraw.ImageDraw.chord()

ellipse(xy, *options)
Draws an ellipse inside the given bounding box.

See also:
PIL.ImageDraw.ImageDraw.ellipse()

line(xy, *options)
Draws a line between the coordinates in the xy list.

See also:
PIL.ImageDraw.ImageDraw.line()

pieslice(xy, start, end, *options)
Same as arc, but also draws straight lines between the end points and the center of the bounding box.

See also:
PIL.ImageDraw.ImageDraw.pieslice()

polygon(xy, *options)
Draws a polygon.

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

See also:
PIL.ImageDraw.ImageDraw.polygon()

rectangle(xy, *options)
Draws a rectangle.

See also:
PIL.ImageDraw.ImageDraw.rectangle()
**text** (xy, text, font)

Draws the string at the given position.

See also:

```
PIL.ImageDraw.ImageDraw.text()
```

**textsize** (text, font)

Return the size of the given string, in pixels.

See also:

```
PIL.ImageDraw.ImageDraw.textsize()
```

**ImageTransform Module**

class PIL.ImageTransform.AffineTransform(data)

Bases: PIL.ImageTransform.Transform

Define an affine image transform.

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

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

See `transform()`

**Parameters**

- **matrix** – A 6-tuple (a, b, c, d, e, f) containing the first two rows from an affine transform matrix.

method = 0

class PIL.ImageTransform.ExtentTransform(data)

Bases: PIL.ImageTransform.Transform

Define a transform to extract a subregion from an image.

Maps a rectangle (defined by two corners) from the image to a rectangle of the given size. The resulting image will contain data sampled from between the corners, such that (x0, y0) in the input image will end up at (0,0) in the output image, and (x1, y1) at size.

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

See `transform()`

**Parameters**

- **bbox** – A 4-tuple (x0, y0, x1, y1) which specifies two points in the input image’s coordinate system. See **Coordinate System**.

method = 1

class PIL.ImageTransform.MeshTransform(data)

Bases: PIL.ImageTransform.Transform

Define a mesh image transform. A mesh transform consists of one or more individual quad transforms.

See `transform()`

**Parameters**

- **data** – A list of (bbox, quad) tuples.

method = 4
class PIL.ImageTransform.QuadTransform(data)
    Bases: PIL.ImageTransform.Transform
    Define a quad image transform.
    Maps a quadrilateral (a region defined by four corners) from the image to a rectangle of the given size.
    See transform()
    Parameters xy – An 8-tuple (x0, y0, x1, y1, x2, y2, x3, y3) which contain the upper left, lower left,
        lower right, and upper right corner of the source quadrilateral.
    method = 3

class PIL.ImageTransform.Transform(data)
    Bases: PIL.Image.ImageTransformHandler
    getdata()
    transform(size, image, **options)

PaletteFile Module

class PIL.PaletteFile.PaletteFile(fp)
    Bases: object
    File handler for Teragon-style palette files.
    getpalette()
    rawmode = 'RGB'

PcfFontFile Module

class PIL.PcfFontFile.PcfFontFile(fp, charset_encoding='iso8859-1')
    Bases: PIL.FontFile.FontFile
    Font file plugin for the X11 PCF format.
    name = 'name'
    PIL.PcfFontFile.sz(s, o)

PngImagePlugin.iTXt Class

class PIL.PngImagePlugin.iTXt(text, lang=None, tkey=None)
    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
**PngImagePlugin.PngInfo Class**

```python
class PIL.PngImagePlugin.PngInfo
    Bases: object
```

PNG chunk container (for use with save(pnginfo=))

```python
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

```python
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

```python
add_text(key, value, zip=False)
    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

**TarIO Module**

```python
class PIL.TarIO.TarIO(tarfile, file)
    Bases: PIL.ContainerIO.ContainerIO
```

A file object that provides read access to a given member of a TAR file.

```python
close()
```

**WalImageFile Module**

This reader is based on the specification available from: [https://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml](https://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml) and has been tested with a few sample files found using google.

**Note:** This format cannot be automatically recognized, so the reader is not registered for use with `PIL.Image.open()`. To open a WAL file, use the `PIL.WalImageFile.open()` function instead.
PIL.WalImageFile.open(filename)
Load texture from a Quake2 WAL texture file.
By default, a Quake2 standard palette is attached to the texture. To override the palette, use the PIL.Image.
Image.putpalette() method.

Parameters filename – WAL file name, or an opened file handle.
Returns An image instance.

1.3.30 Plugin reference

BmpImagePlugin Module

class PIL.BmpImagePlugin.BmpImageFile(fp=None, filename=None)
Bases: PIL.ImageFile.ImageFile
Image plugin for the Windows Bitmap format (BMP)

BITFIELDS = 3
COMPRESSIONS = {'BITFIELDS': 3, 'JPEG': 4, 'PNG': 5, 'RAW': 0, 'RLE4': 2, 'RLE8': 1}
JPEG = 4
PNG = 5
RAW = 0
RLE4 = 2
RLE8 = 1
format = 'BMP'
format_description = 'Windows Bitmap'
k = 'PNG'
v = 5

class PIL.BmpImagePlugin.DibImageFile(fp=None, filename=None)
Bases: PIL.BmpImagePlugin.BmpImageFile
format = 'DIB'
format_description = 'Windows Bitmap'

BufrStubImagePlugin Module

class PIL.BufrStubImagePlugin.BufrStubImageFile(fp=None, filename=None)
Bases: PIL.ImageFile.StubImageFile
format = 'BUFR'
format_description = 'BUFR'
PIL.BufrStubImagePlugin.register_handler(handler)
Install application-specific BUFR image handler.

Parameters handler – Handler object.
CurImagePlugin Module

class PIL.CurImagePlugin.CurImageFile(fp=None, filename=None)
    Bases: PIL.BmpImagePlugin.BmpImageFile
    format = 'CUR'
    format_description = 'Windows Cursor'

DcxImagePlugin Module

class PIL.DcxImagePlugin.DcxImageFile(fp=None, filename=None)
    Bases: PIL.PcxImagePlugin.PcxImageFile
    format = 'DCX'
    format_description = 'Intel DCX'
    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.
    See tell().
    If defined, n_frames refers to the number of available frames.
    Parameters frame – Frame number, starting at 0.
    Raises EOFError – If the call attempts to seek beyond the end of the sequence.

tell()
    Returns the current frame number. See seek().
    If defined, n_frames refers to the number of available frames.
    Returns Frame number, starting with 0.

EpsImagePlugin Module

class PIL.EpsImagePlugin.EpsImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    EPS File Parser for the Python Imaging Library
    format = 'EPS'
    format_description = 'Encapsulated Postscript'
    load(scale=1)
        Load image data based on tile list
    load_seek(*args, **kwargs)
    mode_map = {1: 'L', 2: 'LAB', 3: 'RGB', 4: 'CMYK'}

PIL.EpsImagePlugin.Ghostscript(tile, size, fp, scale=1)
    Render an image using Ghostscript

class PIL.EpsImagePlugin.PSFile(fp)
    Bases: object
    Wrapper for bytesio object that treats either CR or LF as end of line.
readline()
seek(offset, whence=0)
PIL.EpsImagePlugin.has_ghostscript()

**FitsStubImagePlugin Module**

class PIL.FitsStubImagePlugin.FITSStubImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.StubImageFile
    format = 'FITS'
    format_description = 'FITS'
PIL.FitsStubImagePlugin.register_handler(handler)
    Install application-specific FITS image handler.
    Parameters
    handler – Handler object.

**FliImagePlugin Module**

class PIL.FliImagePlugin.FliImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'FLI'
    format_description = 'Autodesk FLI/FLC Animation'
    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.
    See tell().
    If defined, n_frames refers to the number of available frames.
    Parameters
    frame – Frame number, starting at 0.
    Raises
    EOFError – If the call attempts to seek beyond the end of the sequence.
    tell()
    Returns the current frame number. See seek().
    If defined, n_frames refers to the number of available frames.
    Returns
    Frame number, starting with 0.

**FpxImagePlugin Module**

class PIL.FpxImagePlugin.FpxImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'FPX'
    format_description = 'FlashPix'
    load()
    Load image data based on tile list
GbrImagePlugin Module

class PIL.GbrImagePlugin.GbrImageFile (fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    
    format = 'GBR'
    format_description = 'GIMP brush file'
    load()
    Load image data based on tile list

GifImagePlugin Module

class PIL.GifImagePlugin.GifImageFile (fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    
    data()
    format = 'GIF'
    format_description = 'Compuserve GIF'
    global_palette = None
    property is_animated
    load_end()
    
    property n_frames
    
    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.
    
    See tell().
    
    If defined, n_frames refers to the number of available frames.
    
    Parameters frame – Frame number, starting at 0.

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

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

    If defined, n_frames refers to the number of available frames.

    Returns Frame number, starting with 0.

    PIL.GifImagePlugin.get_interlace (im)

    PIL.GifImagePlugin.getdata (im, offset=0, 0, **params)
    Legacy Method
    Return a list of strings representing this image. The first string is a local image header, the rest contains encoded
    image data.

    Parameters
    • im – Image object
    • offset – Tuple of (x, y) pixels. Defaults to (0,0)
    • **params – E.g. duration or other encoder info parameters

1.3. Reference
Returns List of Bytes containing gif encoded frame data

PIL.GifImagePlugin.getheader(im, palette=None, info=None)
Legacy Method to get Gif data from image.

Warning:: May modify image data.

Parameters

- im – Image object
- palette – bytes object containing the source palette, or ....
- info – encoderinfo

Returns tuple of(list of header items, optimized palette)

GribStubImagePlugin Module

class PIL.GribStubImagePlugin.GribStubImageFile(fp=None, filename=None)
Bases: PIL.ImageFile.StubImageFile

format = 'GRIB'
format_description = 'GRIB'

PIL.GribStubImagePlugin.register_handler(handler)
Install application-specific GRIB image handler.

Parameters handler – Handler object.

Hdf5StubImagePlugin Module

class PIL.Hdf5StubImagePlugin.HDF5StubImageFile(fp=None, filename=None)
Bases: PIL.ImageFile.StubImageFile

format = 'HDF5'
format_description = 'HDF5'

PIL.Hdf5StubImagePlugin.register_handler(handler)
Install application-specific HDF5 image handler.

Parameters handler – Handler object.

IcnsImagePlugin Module

class PIL.IcnsImagePlugin.IcnsFile(fobj)
Bases: object

SIZES = { (16, 16, 1): [(b'icp4', <function read_png_or_jpeg2000>), (b'is32', <function ... 512, 1): [(b'ic09', <function read_png_or_jpeg2000>)], (512, 512, 2): [(b'ic10', <function read_png_or_jpeg2000>)]

bestsize()
dataforsize(size)
      Get an icon resource as {channel: array}. Note that the arrays are bottom-up like windows bitmaps and will likely need to be flipped or transposed in some way.

getimage(size=None)

itersizes()
class PIL.IcnsImagePlugin.IcnsImageFile(fp=None, filename=None)
   Bases: PIL.ImageFile.ImageFile

   PIL image support for Mac OS .icns files. Chooses the best resolution, but will possibly load a different size image if you mutate the size attribute before calling 'load'.

   The info dictionary has a key ‘sizes’ that is a list of sizes that the icns file has.

   format = 'ICNS'
   format_description = 'Mac OS icns resource'

   load()
      Load image data based on tile list

   property size

PIL.IcnsImagePlugin.nextheader(fobj)

PIL.IcnsImagePlugin.read_32(fobj, start_length, size)
   Read a 32bit RGB icon resource. Seems to be either uncompressed or an RLE packbits-like scheme.

PIL.IcnsImagePlugin.read_32t(fobj, start_length, size)

PIL.IcnsImagePlugin.read_mk(fobj, start_length, size)

PIL.IcnsImagePlugin.read_png_or_jpeg2000(fobj, start_length, size)

IcoImagePlugin Module

class PIL.IcoImagePlugin.IcoFile(buf)
   Bases: object

   frame(idx)
      Get an image from frame idx

   getentryindex(size, bpp=False)

   getimage(size, bpp=False)
      Get an image from the icon

   sizes()
      Get a list of all available icon sizes and color depths.

class PIL.IcoImagePlugin.IcoImageFile(fp=None, filename=None)
   Bases: PIL.ImageFile.ImageFile

   PIL read-only image support for Microsoft Windows .ico files.

   By default the largest resolution image in the file will be loaded. This can be changed by altering the ‘size’ attribute before calling ‘load’.

   The info dictionary has a key ‘sizes’ that is a list of the sizes available in the icon file.

   Handles classic, XP and Vista icon formats.

   When saving, PNG compression is used. Support for this was only added in Windows Vista.

   This plugin is a refactored version of Win32IconImagePlugin by Bryan Davis <<casadebender@gmail.com>>.
   https://code.google.com/archive/p/casadebender/wikis/Win32IconImagePlugin.wiki

   format = 'ICO'
   format_description = 'Windows Icon'
load()
    Load image data based on tile list

load_seek()

property size

ImImagePlugin Module

class PIL.ImImagePlugin.ImImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'IM'
    format_description = 'IFUNC Image Memory'
    property is_animated
    property n_frames
    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.
        See tell().
        If defined, n_frames refers to the number of available frames.

            Parameters frame – Frame number, starting at 0.

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

tell()
        Returns the current frame number. See seek().
        If defined, n_frames refers to the number of available frames.

            Returns: Frame number, starting with 0.

PIL.ImImagePlugin.number(s)

ImtImagePlugin Module

class PIL.ImtImagePlugin.ImtImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'IMT'
    format_description = 'IM Tools'
IptcImagePlugin Module

class PIL.IptcImagePlugin.IptcImageFile (fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile

    field()
    format = 'IPTC'
    format_description = 'IPTC/NAA'
    getint (key)

load()
    Load image data based on tile list

PIL.IptcImagePlugin.dump(c)
PIL.IptcImagePlugin.getiptcinfo (im)
    Get IPTC information from TIFF, JPEG, or IPTC file.

    Parameters
    im -- An image containing IPTC data.

    Returns
    A dictionary containing IPTC information, or None if no IPTC information block was
    found.

PIL.IptcImagePlugin.i(c)

JpegImagePlugin Module

PIL.JpegImagePlugin.APP (self, marker)
PIL.JpegImagePlugin.COM (self, marker)
PIL.JpegImagePlugin.DQT (self, marker)

class PIL.JpegImagePlugin.JpegImageFile (fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile

    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.

    If any changes are made, returns a tuple with the chosen mode and box with coordinates of the original
    image within the altered one.

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

    Note: This method is not implemented for most images. It is currently implemented only for JPEG and
    MPO images.

    Parameters
    • mode -- The requested mode.
    • size -- The requested size.

    format = 'JPEG'
    format_description = 'JPEG (ISO 10918)'

load_djpeg()
load_read(read_bytes)
    internal: read more image data For premature EOF and LOAD_TRUNCATED_IMAGES adds EOI marker
    so libjpeg can finish decoding

PIL.JpegImagePlugin.SOF(self, marker)
PIL.JpegImagePlugin.Skip(self, marker)
PIL.JpegImagePlugin.convert_dict_qtables(qtables)
PIL.JpegImagePlugin.get_sampling(im)
PIL.JpegImagePlugin.jpeg_factory(fp=None, filename=None)

Jpeg2KImagePlugin Module

class PIL.Jpeg2KImagePlugin.Jpeg2KImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'JPEG2000'
    format_description = 'JPEG 2000 (ISO 15444)'
    load()
        Load image data based on tile list

    property reduce
        Returns a copy of the image reduced factor times. If the size of the image is not dividable by factor,
        the resulting size will be rounded up.

        Parameters
            - factor – A greater than 0 integer or tuple of two integers for width and height separately.
            - box – An optional 4-tuple of ints providing the source image region to be reduced. The
              values must be within (0, 0, width, height) rectangle. If omitted or None, the
              entire source is used.

McIdasImagePlugin Module

class PIL.McIdasImagePlugin.McIdasImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'MCIDAS'
    format_description = 'McIdas area file'

MicImagePlugin Module

class PIL.MicImagePlugin.MicImageFile(fp=None, filename=None)
    Bases: PIL.TiffImagePlugin.TiffImageFile
    format = 'MIC'
    format_description = 'Microsoft Image Composer'
    seek(frame)
        Select a given frame as current image
    tell()
        Return the current frame number

144 Chapter 1. Overview
MpegImagePlugin Module

class PIL.MpegImagePlugin.BitStream(fp)
    Bases: object
    next()
    peek(bits)
    read(bits)
    skip(bits)

class PIL.MpegImagePlugin.MpegImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'MPEG'
    format_description = 'MPEG'

MspImagePlugin Module

class PIL.MspImagePlugin.MspDecoder(mode, *args)
    Bases: PIL.ImageFile.PyDecoder
    decode(buffer)
        Override to perform the decoding process.

        Parameters buffer – A bytes object with the data to be decoded.

        Returns A tuple of (bytes consumed, errcode). If finished with decoding return <0
        for the bytes consumed. Err codes are from ImageFile.ERRORS.

class PIL.MspImagePlugin.MspImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'MSP'
    format_description = 'Windows Paint'

PalmImagePlugin Module

PIL.PalmImagePlugin.build_prototype_image()

PcdImagePlugin Module

class PIL.PcdImagePlugin.PcdImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'PCD'
    format_description = 'Kodak PhotoCD'
    load_end()
PcxImagePlugin Module

```python
class PIL.PcxImagePlugin.PcxImageFile(fp=None, filename=None):
    
    format = 'PCX'
    format_description = 'Paintbrush'
```

PdfImagePlugin Module

PixarImagePlugin Module

```python
class PIL.PixarImagePlugin.PixarImageFile(fp=None, filename=None):
    
    format = 'PIXAR'
    format_description = 'PIXAR raster image'
```

PngImagePlugin Module

```python
class PIL.PngImagePlugin.ChunkStream(fp):
    
    call(cid, pos, length)
        
        Call the appropriate chunk handler
    
    close()
    
    crc(cid, data)
        
        Read and verify checksum
    
    crc_skip(cid, data)
        
        Read checksum. Used if the C module is not present
    
    push(cid, pos, length)
    
    read()
        
        Fetch a new chunk. Returns header information.
    
    verify(endchunk=b'IEND')
```

```python
class PIL.PngImagePlugin.PngImageFile(fp=None, filename=None):
    
    getexif()
    
    load_end()
        
        internal: finished reading image data
    
    load_prepare()
        
        internal: prepare to read PNG file
    
    load_read(read_bytes)
        
        internal: read more image data
    
    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.
```
See `tell()`. If defined, `n_frames` refers to the number of available frames.

**Parameters**

- `frame` – Frame number, starting at 0.
- Raises `EOFError` – If the call attempts to seek beyond the end of the sequence.

**tell()**

- Returns the current frame number. See `seek()`.
- If defined, `n_frames` refers to the number of available frames.

**Returns**

Frame number, starting with 0.

**verify()**

Verify PNG file

- `format = 'PNG'
- `format_description = 'Portable network graphics'
- `property text`

**class PIL.PngImagePlugin.PngStream(fp)**

Bases: `PIL.PngImagePlugin.ChunkStream`

- `check_text_memory(chunklen)`
- `chunk_IDAT(pos, length)`
- `chunk_IEND(pos, length)`
- `chunk_IHDR(pos, length)`
- `chunk_PLTE(pos, length)`
- `chunk_acTL(pos, length)`
- `chunk_cHRM(pos, length)`
- `chunk_eXIf(pos, length)`
- `chunk_fcTL(pos, length)`
- `chunk_fdAT(pos, length)`
- `chunk_gAMA(pos, length)`
- `chunk_iCCP(pos, length)`
- `chunk_iTXt(pos, length)`
- `chunk_pHYs(pos, length)`
- `chunk_sRGB(pos, length)`
- `chunk_tEXT(pos, length)`
- `chunk_tRNS(pos, length)`
- `chunk_zTXT(pos, length)`
- `rewind()`
- `save_rewind()`

**PIL.PngImagePlugin.getchunks(im, **params)**

Return a list of PNG chunks representing this image.
PIL.PngImagePlugin.is_cid(string, pos=0, endpos=922372036854775807)
  Matches zero or more characters at the beginning of the string.

PIL.PngImagePlugin.putchunk(fp, cid, *data)
  Write a PNG chunk (including CRC field)

PIL.PngImagePlugin.APNG_BLEND_OP_OVER = 1
  This frame should be alpha composited with the previous output image contents. See Saving APNG sequences.

PIL.PngImagePlugin.APNG_BLEND_OP_SOURCE = 0
  All color components of this frame, including alpha, overwrite the previous output image contents. See Saving APNG sequences.

PIL.PngImagePlugin.APNG_DISPOSE_OP_BACKGROUND = 1
  This frame’s modified region is cleared to fully transparent black before rendering the next frame. See Saving APNG sequences.

PIL.PngImagePlugin.APNG_DISPOSE_OP_NONE = 0
  No disposal is done on this frame before rendering the next frame. See Saving APNG sequences.

PIL.PngImagePlugin.APNG_DISPOSE_OP_PREVIOUS = 2
  This frame’s modified region is reverted to the previous frame’s contents before rendering the next frame. See Saving APNG sequences.

PIL.PngImagePlugin.MAX_TEXT_CHUNK = 1048576
  Maximum decompressed size for a iTXt or zTXt chunk. Eliminates decompression bombs where compressed chunks can expand 1000x. See Text in PNG File Format.

PIL.PngImagePlugin.MAX_TEXT_MEMORY = 67108864
  Set the maximum total text chunk size. See Text in PNG File Format.

PpmImagePlugin Module

class PIL.PpmImagePlugin.PpmImageFile(fp=None, filename=None)
  Bases: PIL.ImageFile.ImageFile
  format = 'PPM'
  format_description = 'Pbmplus image'

PsdImagePlugin Module

class PIL.PsdImagePlugin.PsdImageFile(fp=None, filename=None)
  Bases: PIL.ImageFile.ImageFile
  format = 'PSD'
  format_description = 'Adobe Photoshop'
  load_prepare()

  seek(layer)
    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.

    See tell().

    If defined, n_frames refers to the number of available frames.

    Parameters frame – Frame number, starting at 0.
Raises **EOFError** – If the call attempts to seek beyond the end of the sequence.

**tell()**

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

If defined, **n_frames** refers to the number of available frames.

Returns Frame number, starting with 0.

### SgiImagePlugin Module

class PIL.SgiImagePlugin.SGI16Decoder (*mode*, *args*)

Bases: PIL.ImageFile.PyDecoder

decode (*buffer*)

Override to perform the decoding process.

Parameters **buffer** – A bytes object with the data to be decoded.

Returns A tuple of (bytes consumed, errcode). If finished with decoding return <0 for the bytes consumed. Err codes are from ImageFile.ERRORS.

class PIL.SgiImagePlugin.SgiImageFile (fp=None, filename=None)

Bases: PIL.ImageFile.ImageFile

format = 'SGI'

format_description = 'SGI Image File Format'

### SpiderImagePlugin Module

class PIL.SpiderImagePlugin.SpiderImageFile (fp=None, filename=None)

Bases: PIL.ImageFile.ImageFile

convert2byte (*depth=255*)

format = 'SPIDER'

format_description = 'Spider 2D image'

property **is_animated**

property **n_frames**

**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.

See **tell()**.

If defined, **n_frames** refers to the number of available frames.

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

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

**tell()**

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

If defined, **n_frames** refers to the number of available frames.

Returns Frame number, starting with 0.
tkPhotoImage()

PIL.SpiderImagePlugin.isInt(f)

PIL.SpiderImagePlugin.isSpiderHeader(t)

PIL.SpiderImagePlugin.isSpiderImage(filename)

PIL.SpiderImagePlugin.loadImageSeries(filelist=None)

create a list of Image objects for use in a montage

PIL.SpiderImagePlugin.makeSpiderHeader(im)

**SunImagePlugin Module**

class PIL.SunImagePlugin.SunImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'SUN'
    format_description = 'Sun Raster File'

**TgaImagePlugin Module**

class PIL.TgaImagePlugin.TgaImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'TGA'
    format_description = 'Targa'

**TiffImagePlugin Module**

class PIL.TiffImagePlugin.AppendingTiffWriter(fn, new=False)
    Bases: object
    Tags = {273, 288, 324, 519, 520, 521}
    close()
    fieldSizes = [0, 1, 1, 2, 4, 8, 1, 1, 2, 4, 8, 4, 8]
    finalize()
    fixIFD()
    fixOffsets(count, isShort=False, isLong=False)
    goToEnd()
    newFrame()
    readLong()
    readShort()
    rewriteLastLong(value)
    rewriteLastShort(value)
    rewriteLastShortToLong(value)
    seek(offset, whence=0)
setEndian (endian)
setup()
skipIFDs()
tell()
write (data)
writeLong (value)
writeShort (value)

class PIL.TiffImagePlugin.IFDRational (value, denominator=1)
   Bases: numbers.Rational
   Implements a rational class where 0/0 is a legal value to match the in
   the wild use of exif rationals.
   e.g., DigitalZoomRatio - 0.00/0.00 indicates that no digital zoom was used

   property denominator

   limit_rational (max_denominator)
      Parameters max_denominator – Integer, the maximum denominator value
      Returns Tuple of (numerator, denominator)

   property numerator

PIL.TiffImagePlugin.ImageFileDirectory
   alias of PIL.TiffImagePlugin.ImageFileDirectory_v1

class PIL.TiffImagePlugin.ImageFileDirectory_v1 (*args, **kwargs)
   Bases: PIL.TiffImagePlugin.ImageFileDirectory_v2

   This class represents the legacy interface to a TIFF tag directory.
   Exposes a dictionary interface of the tags in the directory:

   ifd = ImageFileDirectory_v1()
   ifd[key] = 'Some Data'
   ifd.tagtype[key] = TiffTags.ASCII
   print (ifd[key])
   ('Some Data',)

   Also contains a dictionary of tag types as read from the tiff image file, tagtype.
   Values are returned as a tuple.
   Deprecation since version 3.0.0.

   classmethod from_v2 (original)
      Returns an ImageFileDirectory_v1 instance with the same data as is contained in the original
      ImageFileDirectory_v2 instance.
      Returns ImageFileDirectory_v1

   property tagdata

   property tags

   tagtype: dict
      Dictionary of tag types
to_v2()
    Returns an :class:`ImageFileDirectory_v2` instance with the same data as is contained in the original

    Returns :class:`ImageFileDirectory_v2`

class PIL.TiffImagePlugin.ImageFileDirectory_v2(ifh=b'I\x00\x00\x00\x00\x00',  
    prefix=None)

    Bases: collections.abc.MutableMapping

    This class represents a TIFF tag directory. To speed things up, we don’t decode tags unless they’re asked for.

    Exposes a dictionary interface of the tags in the directory:

    >>> ifd = ImageFileDirectory_v2()
    >>> ifd[key] = 'Some Data'
    >>> ifd.tagtype[key] = TiffTags.ASCII
    >>> print(ifd[key])
    'Some Data'

    Individual values are returned as the strings or numbers, sequences are returned as tuples of the values.

    The tiff metadata type of each item is stored in a dictionary of tag types in :attr:`tagtype`. The types are read from
    a tiff file, guessed from the type added, or added manually.

    Data Structures:
    * self.tagtype = {}
        * Key: numerical tiff tag number
        * Value: integer corresponding to the data type from :attr:`PIL.TiffTags.TYPES`

    New in version 3.0.0.

    property legacy_api
    load(fp)
    load_byte(data, legacy_api=True)
    load_double(data, legacy_api=True)
    load_float(data, legacy_api=True)
    load_long(data, legacy_api=True)
    load_rational(data, legacy_api=True)
    load_short(data, legacy_api=True)
    load_signed_byte(data, legacy_api=True)
    load_signed_long(data, legacy_api=True)
    load_signed_rational(data, legacy_api=True)
    load_signed_short(data, legacy_api=True)
    load_string(data, legacy_api=True)
    load_undefined(data, legacy_api=True)
    named()

    Returns dict of name|key: value

    Returns the complete tag dictionary, with named tags where possible.
property offset
property prefix
reset()
save(fp)
tagtype
    Dictionary of tag types
tobytes(offset=0)
write_byte(data)
write_double(*values)
write_float(*values)
write_long(*values)
write_rational(*values)
write_short(*values)
write_signed_byte(*values)
write_signed_long(*values)
write_signed_rational(*values)
write_signed_short(*values)
write_string(value)
write_undefined(value)
class PIL.TiffImagePlugin.TiffImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'TIFF'
    format_description = 'Adobe TIFF'
load()
    Load image data based on tile list
load_end()
property n_frames
seek(frame)
    Select a given frame as current image
tag
    Legacy tag entries
tag_v2
    Image file directory (tag dictionary)
tell()
    Return the current frame number
WebPImagePlugin Module

class PIL.WebPImagePlugin.WebPImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    
    format = 'WEBP'
    format_description = 'WebP image'
    load()
        Load image data based on tile list
    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.
        
        See tell().
        
        If defined, n_frames refers to the number of available frames.

        Parameters frame – Frame number, starting at 0.
        
        Raises EOFError – If the call attempts to seek beyond the end of the sequence.
    
    tell()
        Returns the current frame number. See seek().
        
        If defined, n_frames refers to the number of available frames.

        Returns Frame number, starting with 0.

WmfImagePlugin Module

class PIL.WmfImagePlugin.WmfStubImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.StubImageFile
    
    format = 'WMF'
    format_description = 'Windows Metafile'
    load(dpi=None)
        Load image data based on tile list
    PIL.WmfImagePlugin.register_handler(handler)
        Install application-specific WMF image handler.

        Parameters handler – Handler object.

XVThumbImagePlugin Module

class PIL.XVThumbImagePlugin.XVThumbImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    
    format = 'XVThumb'
    format_description = 'XV thumbnail image'
XbmImagePlugin Module

class PIL.XbmImagePlugin.XbmImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'XBM'
    format_description = 'X11 Bitmap'

XpmImagePlugin Module

class PIL.XpmImagePlugin.XpmImageFile(fp=None, filename=None)
    Bases: PIL.ImageFile.ImageFile
    format = 'XPM'
    format_description = 'X11 Pixel Map'
    load_read(bytes)

1.3.31 Internal Reference Docs

File Handling in Pillow

When opening a file as an image, Pillow requires a filename, pathlib.Path object, or a file-like object. Pillow uses the filename or Path to open a file, so for the rest of this article, they will all be treated as a file-like object.

The following are all equivalent:

```python
from PIL import Image
import io
import pathlib

with Image.open('test.jpg') as im:
    ...

with Image.open(pathlib.Path('test.jpg')) as im2:
    ...

with open('test.jpg', 'rb') as f:
    im3 = Image.open(f)
    ...

with open('test.jpg', 'rb') as f:
    im4 = Image.open(io.BytesIO(f.read()))
    ...
```

If a filename or a path-like object is passed to Pillow, then the resulting file object opened by Pillow may also be closed by Pillow after the `Image.Image.load()` method is called, provided the associated image does not have multiple frames.

Pillow cannot in general close and reopen a file, so any access to that file needs to be prior to the close.
Image Lifecycle

- `Image.open()` Filenames and `Path` objects are opened as a file. Metadata is read from the open file. The file is left open for further usage.

- `Image.Image.load()` When the pixel data from the image is required, `load()` is called. The current frame is read into memory. The image can now be used independently of the underlying image file.

  If a filename or a `Path` object was passed to `Image.open()`, then the file object was opened by Pillow and is considered to be used exclusively by Pillow. So if the image is a single-frame image, the file will be closed in this method after the frame is read. If the image is a multi-frame image, (e.g. multipage TIFF and animated GIF) the image file is left open so that `Image.Image.seek()` can load the appropriate frame.

- `Image.Image.close()` Closes the file and destroys the core image object. This is used in the Pillow context manager support. e.g.:

  ```python
  with Image.open('test.jpg') as img:
      ...  # image operations here.
  ```

The lifecycle of a single-frame image is relatively simple. The file must remain open until the `load()` or `close()` function is called or the context manager exits.

Multi-frame images are more complicated. The `load()` method is not a terminal method, so it should not close the underlying file. In general, Pillow does not know if there are going to be any requests for additional data until the caller has explicitly closed the image.

Complications

- `TiffImagePlugin` has some code to pass the underlying file descriptor into libtiff (if working on an actual file). Since libtiff closes the file descriptor internally, it is duplicated prior to passing it into libtiff.

- After a file has been closed, operations that require file access will fail:

  ```python
  with open('test.jpg', 'rb') as f:
      im5 = Image.open(f)
  im5.load()  # FAILS, closed file

  with Image.open('test.jpg') as im6:
      pass
  im6.load()  # FAILS, closed file
  ```

Proposed File Handling

- `Image.Image.load()` should close the image file, unless there are multiple frames.

- `Image.Image.seek()` should never close the image file.

- Users of the library should use a context manager or call `Image.Image.close()` on any image opened with a filename or `Path` object to ensure that the underlying file is closed.
Limits

This page is documentation to the various fundamental size limits in the Pillow implementation.

Internal Limits

- Image sizes cannot be negative. These are checked both in Storage.c and Image.py
- Image sizes may be 0. (Although not in 3.4)
- Maximum pixel dimensions are limited to INT32, or $2^{31}$ by the sizes in the image header.
- Individual allocations are limited to 2GB in Storage.c
- The 2GB allocation puts an upper limit to the xsize of the image of either $2^{31}$ for ‘L’ or $2^{29}$ for ‘RGB’
- Individual memory mapped segments are limited to 2GB in map.c based on the overflow checks. This requires that any memory mapped image is smaller than 2GB, as calculated by $y \times \text{stride}$ (so 2Gpx for ‘L’ images, and .5Gpx for ‘RGB’)
- Any call to internal python size functions for buffers or strings are currently returned as int32, not py_ssize_t. This limits the maximum buffer to 2GB for operations like frombytes and frombuffer.
- This also limits the size of buffers converted using a decoder. (decode.c:127)

Format Size Limits

- ICO: Max size is 256x256
- Webp: 16383x16383 (underlying library size limit: https://developers.google.com/speed/webp/docs/api)

Block Allocator

Previous Design

Historically there have been two image allocators in Pillow: ImagingAllocateBlock and ImagingAllocateArray. The first works for images smaller than 16MB of data and allocations one large chunk of memory of \( \text{im->linesize} \times \text{im->ysize} \) bytes. The second works for large images and makes one allocation for each scan line of size \( \text{im->linesize} \) bytes. This makes for a very sharp transition between one allocation and potentially thousands of small allocations, leading to unpredictable performance penalties around the transition.

New Design

ImagingAllocateArray now allocates space for images as a chain of blocks with a maximum size of 16MB. If there is a memory allocation error, it falls back to allocating a 4KB block, or at least one scan line. This is now the default for all internal allocations.

ImagingAllocateBlock is now only used for those cases when we are specifically requesting a single segment of memory for sharing with other code.
Memory Pools

There is now a memory pool to contain a supply of recently freed blocks, which can then be reused without going back to the OS for a fresh allocation. This caching of free blocks is currently disabled by default, but can be enabled and tweaked using three environment variables:

- `PILLOW_ALIGNMENT`, in bytes. Specifies the alignment of memory allocations. Valid values are powers of 2 between 1 and 128, inclusive. Defaults to 1.
- `PILLOW_BLOCK_SIZE`, in bytes, K, or M. Specifies the maximum block size for `ImagingAllocateArray`. Valid values are integers, with an optional k or m suffix. Defaults to 16M.
- `PILLOW_BLOCKS_MAX` Specifies the number of freed blocks to retain to fill future memory requests. Any freed blocks over this threshold will be returned to the OS immediately. Defaults to 0.

Internal Modules

_binary Module

Binary input/output support routines.

PIL._binary.i16be(c, o=0)
PIL._binary.i16le(c, o=0)
  Converts a 2-bytes (16 bits) string to an unsigned integer.
  **Parameters**
  - 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 unsigned integer.
  **Parameters**
  - 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)
PIL._binary.si16le(c, o=0)
  Converts a 2-bytes (16 bits) string to a signed integer.
  **Parameters**
  - c – string containing bytes to convert
  - o – offset of bytes to convert in string
PIL._binary.si32le(c, o=0)
Converts a 4-bytes (32 bits) string to a signed integer.

Parameters
- c – string containing bytes to convert
- o – offset of bytes to convert in string

_tkinter_finder Module

Find compiled module linking to Tcl / Tk libraries

__util Module

class PIL._util.deferred_error(ex)
    Bases: object
PIL._util.isDirectory(f)
PIL._util.isDirectory(f)

__version Module

PIL._version.__version__: str
    This is the master version number for Pillow, all other uses reference this module.

PIL.Image.core Module

An internal interface module previously known as _imaging, implemented in _imaging.c.

1.4 Porting

Porting existing PIL-based code to Pillow

Pillow is a functional drop-in replacement for the Python Imaging Library.

PIL is Python 2 only. Pillow dropped support for Python 2 in Pillow 7.0. So if you would like to run the latest version of Pillow, you will first and foremost need to port your code from Python 2 to 3.

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 PIL._imaging module has been moved to PIL.Image.core. 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.

### 1.5 About

#### 1.5.1 Goals

The fork author’s goal is to foster and support active development of PIL through:

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

#### 1.5.2 License

Like PIL, Pillow is licensed under the open source PIL Software License.

#### 1.5.3 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.

#### 1.5.4 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 (1.1.7 in 2009), the likelihood of a new PIL release decreases. However, we’ve yet to hear an official “PIL is dead” announcement.

### 1.6 Release Notes

Pillow is released quarterly on January 2nd, April 1st, July 1st and October 15th. Patch releases are created if the latest release contains severe bugs, or if security fixes are put together before a scheduled release.

Please use the latest version of Pillow. Functionality and security fixes should not be expected to be backported to earlier versions.

**Note:** Contributors please include release notes as needed or appropriate with your bug fixes, feature additions and tests.
1.6.1 8.0.0

Backwards Incompatible Changes

Python 3.5

Pillow has dropped support for Python 3.5, which reached end-of-life on 2020-09-13.

im.offset

im.offset() has been removed, call ImageChops.offset() instead.

Image.fromstring, im.fromstring and im.tostring

- Image.fromstring() has been removed, call Image.frombytes() instead.
- im.fromstring() has been removed, call frombytes() instead.
- im.tostring() has been removed, call tobytes() instead.

ImageCms.CmsProfile attributes

Some attributes in PIL.ImageCms.CmsProfile have been removed:

<table>
<thead>
<tr>
<th>Removed</th>
<th>Use instead</th>
</tr>
</thead>
<tbody>
<tr>
<td>color_space</td>
<td>Padded xcolor_space</td>
</tr>
<tr>
<td>pcs</td>
<td>Padded connection_space</td>
</tr>
<tr>
<td>product_copyright</td>
<td>Unicode copyright</td>
</tr>
<tr>
<td>product_desc</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_description</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_manufacturer</td>
<td>Unicode manufacturer</td>
</tr>
<tr>
<td>product_model</td>
<td>Unicode model</td>
</tr>
</tbody>
</table>

API Changes

Add MIME type to PsdImagePlugin

“image/vnd.adobe.photoshop” is now registered as the PsdImagePlugin.PsdImageFile MIME type.

API Additions

Image.open: add formats parameter

Added a new formats parameter to Image.open():

- A list or tuple of formats to attempt to load the file in. This can be used to restrict the set of formats checked.
  Pass None to try all supported formats. You can print the set of available formats by running python -m PIL or using the PIL.features.pilinfo() function.
ImageOps.autocontrast: add mask parameter

ImageOps.autocontrast() can now take a mask parameter:

- Histogram used in contrast operation is computed using pixels within the mask. If no mask is given the entire image is used for histogram computation.

ImageOps.autocontrast cutoffs

Previously, the cutoff parameter of ImageOps.autocontrast() could only be a single number, used as the percent to cut off from the histogram on the low and high ends.

Now, it can also be a tuple (low, high).

ImageDraw.regular_polygon

A new method ImageDraw.regular_polygon(), draws a regular polygon of n_sides, inscribed in a bounding_circle.

For example draw.regular_polygon(((100, 100), 50), 5) draws a pentagon centered at the point (100, 100) with a polygon radius of 50.

Security

TODO

Other Changes

TODO

TODO

1.6.2 7.2.0

API Changes

Replaced TiffImagePlugin DEBUG with logging

TiffImagePlugin.DEBUG = True has been a way to print various debugging information when interacting with TIFF images. This has now been removed in favour of Python’s logging module, already used in other places in the Pillow source code.
Corrected default offset when writing EXIF data

Previously, the default offset argument for `tobytes()` was 0, which did not include the magic header. It is now 8.

Moved to `ImageFileDirectory_v2` in `Image.Exif`

Moved from the legacy `PIL.TiffImagePlugin.ImageFileDirectory_v1` to `PIL.TiffImagePlugin.ImageFileDirectory_v2` in `PIL.Image.Exif`. This means that Exif RATIONALs and SIGNED_RATIONALs are now read as `PIL.TiffImagePlugin.IFDRational`, instead of as a tuple with a numerator and a denominator.

TIFF BYTE tags format

TIFF BYTE tags were previously read as a tuple containing a bytestring. They are now read as just a single bytestring.

Deprecations

Image.show command parameter

The `command` parameter was deprecated and will be removed in a future release. Use a subclass of `PIL.ImageShow.Viewer` instead.

Image._showxv

`Image._showxv` has been deprecated. Use `show()` instead. If custom behaviour is required, use `register()` to add a custom `Viewer` class.

ImageFile.raise_ioerror

IOError was merged into OSError in Python 3.3. So, `ImageFile.raise_ioerror` is now deprecated and will be removed in a future release. Use `ImageFile.raise_oserror` instead.

1.6.3  7.1.2

Fix another regression seeking PNG files

This fixes a regression introduced in 7.1.0 when adding support for APNG files.

When calling `seek(n)` on a regular PNG where `n > 0`, it failed to raise an EOFError as it should have done, resulting in:

```
AttributeError: 'NoneType' object has no attribute 'read'
```

Pillow 7.1.2 now raises the correct exception.
1.6.4 7.1.1

Fix regression seeking PNG files

This fixes a regression introduced in 7.1.0 when adding support for APNG files when calling `seek` and `tell`:

```python
>>> from PIL import Image
>>> with Image.open("Tests/images/hopper.png") as im:
    ...   im.seek(0)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/PIL/PngImagePlugin.py", line 739, in seek
      if not self._seek_check(frame):
    File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/PIL/ImageFile.py", line 306, in _seek_check
      return self.tell() != frame
    File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/PIL/PngImagePlugin.py", line 827, in tell
      return self.__frame
AttributeError: 'PngImageFile' object has no attribute '__frame'
```

1.6.5 7.1.0

API Changes

Allow saving of zero quality JPEG images

If no quality was specified when saving a JPEG, Pillow internally used a value of zero to indicate that the default quality should be used. However, this removed the ability to actually save a JPEG with zero quality. This has now been resolved.

```python
from PIL import Image
im = Image.open("hopper.jpg")
im.save("out.jpg", quality=0)
```

API Additions

New channel operations

Three new channel operations have been added: `soft_light()`, `hard_light()` and `overlay()`.
PILLOW_VERSION constant

PILLOW_VERSION has been re-added but is deprecated and will be removed in a future release. Use __version__ instead.

It was initially removed in Pillow 7.0.0, but brought back in 7.1.0 to give projects more time to upgrade.

Reading JPEG comments

When opening a JPEG image, the comment may now be read into info.

Support for different charset encodings in PcfFontFile

Previously PcfFontFile output only bitmap PIL fonts with ISO 8859-1 encoding, even though the PCF format supports Unicode, making it hard to work with Pillow with bitmap fonts in languages which use different character sets.

Now it’s possible to set a different charset encoding in PcfFontFile’s class constructor. By default, it generates a PIL font file with ISO 8859-1 as before. The generated PIL font file still contains up to 256 characters, but the character set is different depending on the selected encoding.

To use such a font with ImageDraw.text, call it with a bytes object with the same encoding as the font file.

X11 ImageGrab.grab()

Support has been added for ImageGrab.grab() on Linux using the X server with the XCB library.

An optional xdisplay parameter has been added to select the X server, with the default value of None using the default X server.

Passing a different value on Windows or macOS will force taking a snapshot using the selected X server; pass an empty string to use the default X server. XCB support is not included in pre-compiled wheels for Windows and macOS.

Security

This release includes security fixes.

- CVE-2020-10177 Fix multiple OOB reads in FLI decoding
- CVE-2020-10378 Fix bounds overflow in PCX decoding
- CVE-2020-10379 Fix two buffer overflows in TIFF decoding
- CVE-2020-10994 Fix bounds overflow in JPEG 2000 decoding
- CVE-2020-11538 Fix buffer overflow in SGI-RLE decoding
Other Changes

If present, only use alpha channel for bounding box

When the `getbbox()` method calculates the bounding box, for an RGB image it trims black pixels. Similarly, for an RGBA image it would trim black transparent pixels. This is now changed so that if an image has an alpha channel (RGBA, RGBa, PA, LA, La), any transparent pixels are trimmed.

Improved APNG support

Added support for reading and writing Animated Portable Network Graphics (APNG) images. The PNG plugin now supports using the `seek()` method and the `Iterator` class to read APNG frame sequences. The PNG plugin also now supports using the `append_images` argument to write APNG frame sequences. See APNG sequences for further details.

1.6.6 7.0.0

Backwards Incompatible Changes

Python 2.7

Pillow has dropped support for Python 2.7, which reached end-of-life on 2020-01-01.

PILLOW_VERSION constant

`PILLOW_VERSION` has been removed. Use `__version__` instead.
**PIL.*ImagePlugin.__version__ attributes**

The version constants of individual plugins have been removed. Use `PIL.__version__` instead.

<table>
<thead>
<tr>
<th>Removed</th>
<th>Removed</th>
<th>Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BmpImagePlugin.<strong>version</strong></td>
<td>Jpeg2KImagePlugin.<strong>version</strong></td>
<td>PngImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>CurImagePlugin.<strong>version</strong></td>
<td>JpegImagePlugin.<strong>version</strong></td>
<td>PpmImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>DcxImagePlugin.<strong>version</strong></td>
<td>McIdasImagePlugin.<strong>version</strong></td>
<td>PsdImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>EpsImagePlugin.<strong>version</strong></td>
<td>MicImagePlugin.<strong>version</strong></td>
<td>SgiImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>FliImagePlugin.<strong>version</strong></td>
<td>MpegImagePlugin.<strong>version</strong></td>
<td>SunImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>FpxImagePlugin.<strong>version</strong></td>
<td>MpoImagePlugin.<strong>version</strong></td>
<td>TgaImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>GdImageFile.<strong>version</strong></td>
<td>MspImagePlugin.<strong>version</strong></td>
<td>TiffImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>GifImagePlugin.<strong>version</strong></td>
<td>PalmImagePlugin.<strong>version</strong></td>
<td>WmfImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>IcoImagePlugin.<strong>version</strong></td>
<td>PcdImagePlugin.<strong>version</strong></td>
<td>XbmImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>ImImagePlugin.<strong>version</strong></td>
<td>PcXImagePlugin.<strong>version</strong></td>
<td>XpmImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>ImtImagePlugin.<strong>version</strong></td>
<td>PdfImagePlugin.<strong>version</strong></td>
<td>XVThumbImagePlugin.<strong>version</strong></td>
</tr>
<tr>
<td>IptcImagePlugin.<strong>version</strong></td>
<td>PixarImagePlugin.<strong>version</strong></td>
<td></td>
</tr>
</tbody>
</table>

**PyQt4 and PySide**

Qt 4 reached end-of-life on 2015-12-19. Its Python bindings are also EOL: PyQt4 since 2018-08-31 and PySide since 2015-10-14.

Support for PyQt4 and PySide has been removed from `ImageQt`. Please upgrade to PyQt5 or PySide2.
Setting the size of TIFF images

Setting the size of a TIFF image directly (e.g., `im.size = (256, 256)`) throws an error. Use `Image.resize` instead.

Default resampling filter

The default resampling filter has been changed to the high-quality convolution `Image.BICUBIC` instead of `Image.NEAREST`. For the `resize()` method and the `pad()`, `scale()` and `fit()` functions, `Image.NEAREST` is still always used for images in “P” and “1” modes. See `Filters` to learn the difference. In short, `Image.NEAREST` is a very fast filter, but simple and low-quality.

Image.draft() return value

If the `draft()` method has no effect, it returns `None`. If it does have an effect, then it previously returned the image itself. However, unlike other chain methods, `draft()` does not return a modified version of the image, but modifies it in-place. So instead, if `draft()` has an effect, Pillow will now return a tuple of the image mode and a co-ordinate box. The box is the original coordinates in the bounds of resulting image. This may be useful in a subsequent `resize()` call.

API Additions

Custom unidentified image error

Pillow will now throw a custom `UnidentifiedImageError` when an image cannot be identified. For backwards compatibility, this will inherit from `OSError`.

New argument reducing_gap for Image.resize() and Image.thumbnail() methods

Speeds up resizing by resizing the image in two steps. The bigger `reducing_gap`, the closer the result to the fair resampling. The smaller `reducing_gap`, the faster resizing. With `reducing_gap` greater or equal to 3.0, the result is indistinguishable from fair resampling.

The default value for `resize()` is `None`, which means that the optimization is turned off by default.

The default value for `thumbnail()` is 2.0, which is very close to fair resampling while still being faster in many cases. In addition, the same gap is applied when `thumbnail()` calls `draft()`, which may greatly improve the quality of JPEG thumbnails. As a result, `thumbnail()` in the new version provides equally high speed and high quality from any source (JPEG or arbitrary images).

New Image.reduce() method

`reduce()` is a highly efficient operation to reduce an image by integer times. Normally, it shouldn’t be used directly. Used internally by `resize()` and `thumbnail()` methods to speed up resize when a new argument `reducing_gap` is set.
Loading WMF images at a given DPI

On Windows, Pillow can read WMF files, with a default DPI of 72. An image can now also be loaded at another resolution:

```python
from PIL import Image
with Image.open("drawing.wmf") as im:
    im.load(dpi=144)
```

Other Changes

**Image.__del__**

Implicitly closing the image’s underlying file in `Image.__del__` has been removed. Use a context manager or call `close()` instead to close the file in a deterministic way.

Previous method:

```python
im = Image.open("hopper.png")
im.save("out.jpg")
```

Use instead:

```python
with Image.open("hopper.png") as im:
    im.save("out.jpg")
```

Better thumbnail geometry

When calculating the new dimensions in `thumbnail()`, round to the nearest integer, instead of always rounding down. This better preserves the original aspect ratio.

When the image width or height is not divisible by 8 the last row and column in the image get the correct weight after JPEG DCT scaling.

1.6.7 6.2.2

Security

This release addresses several security problems.

CVE-2019-19911 is regarding FPX images. If an image reports that it has a large number of bands, a large amount of resources will be used when trying to process the image. This is fixed by limiting the number of bands to those usable by Pillow.

Buffer overruns were found when processing an SGI (CVE-2020-5311), PCX (CVE-2020-5312) or FLI image (CVE-2020-5313). Checks have been added to prevent this.

CVE-2020-5310: Overflow checks have been added when calculating the size of a memory block to be reallocated in the processing of a TIFF image.
1.6.8 6.2.1

API Changes

Deprecations

Python 2.7

Python 2.7 reaches end-of-life on 2020-01-01.

Pillow 7.0.0 will be released on 2020-01-01 and will drop support for Python 2.7, making Pillow 6.2.x the last release series to support Python 2.

Other Changes

Support added for Python 3.8

Pillow 6.2.1 supports Python 3.8.

1.6.9 6.2.0

API Additions

Text stroking

stroke_width and stroke_fill arguments have been added to text drawing operations. They allow text to be outlined, setting the width of the stroke and the color respectively. If not provided, stroke_fill will default to the fill parameter.

```python
from PIL import Image, ImageDraw, ImageFont

font = ImageFont.truetype("Tests/fonts/FreeMono.ttf", 40)
font.getsize_multiline("A", stroke_width=2)
font.getsize("ABC\nAaaa", stroke_width=2)

im = Image.new("RGB", (100, 100))
draw = ImageDraw.Draw(im)
draw.textsize("A", font, stroke_width=2)
draw.multiline_textsize("ABC\nAaaa", font, stroke_width=2)
draw.text((10, 10), "A", "#f00", font, stroke_width=2, stroke_fill="#0f0")
draw.multiline_text((10, 10), "A\nB", "#f00", font, stroke_width=2, stroke_fill="#0f0")

For example,

```python
from PIL import Image, ImageDraw, ImageFont

im = Image.new("RGB", (120, 130))
draw = ImageDraw.Draw(im)
font = ImageFont.truetype("Tests/fonts/FreeMono.ttf", 120)
draw.text((10, 10), "A", "#f00", font, stroke_width=2, stroke_fill="#0f0")
```
creates the following image:

![Image](image.png)

**ImageGrab on multi-monitor Windows**

An **all_screens** argument has been added to `ImageGrab.grab`. If `True`, all monitors will be included in the created image.

**API Changes**

**Image.getexif**

To allow for lazy loading of Exif data, `Image.getexif()` now returns a shared instance of `Image.Exif`.

**Deprecations**

**Image.frombuffer**

There has been a longstanding warning that the defaults of `Image.frombuffer` may change in the future for the “raw” decoder. The change will now take place in Pillow 7.0.

**Security**

This release catches several buffer overruns, as well as addressing CVE-2019-16865. The CVE is regarding DOS problems, such as consuming large amounts of memory, or taking a large amount of time to process an image.

In RawDecode.c, an error is now thrown if skip is calculated to be less than zero. It is intended to skip padding between lines, not to go backwards.

In PsdImagePlugin, if the combined sizes of the individual parts is larger than the declared size of the extra data field, then it looked for the next layer by seeking backwards. This is now corrected by seeking to (the start of the layer + the size of the extra data field) instead of (the read parts of the layer + the rest of the layer).

Decompression bomb checks have been added to GIF and ICO formats.

An error is now raised if a TIFF dimension is a string, rather than trying to perform operations on it.
Other Changes

Removed bdist_wininst .exe installers

.exe installers fell out of favour with PEP 527, and will be deprecated in Python 3.8. Pillow will no longer be distributing them. Wheels should be used instead.

Flags for libwebp in wheels

When building libwebp for inclusion in wheels, Pillow now adds the -O3 and -DNDEBUG CFLAGS. These flags would be used by default if building libwebp without debugging, and using them fixes a significant decrease in speed when a wheel-installed copy of Pillow performs libwebp operations.

1.6.10 6.1.0

Deprecations

Image.__del__

Deprecated since version 6.1.0. Implicitly closing the image’s underlying file in Image.__del__ has been deprecated. Use a context manager or call Image.close() instead to close the file in a deterministic way.

Deprecation:

```python
im = Image.open("hopper.png")
im.save("out.jpg")
```

Use instead:

```python
with Image.open("hopper.png") as im:
    im.save("out.jpg")
```

API Additions

Image.entropy

Calculates and returns the entropy for the image. A bilevel image (mode “1”) is treated as a greyscale (“L”) image by this method. If a mask is provided, the method employs the 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”).
ImageGrab.grab

An optional `include_layered_windows` parameter has been added to `ImageGrab.grab`, defaulting to `False`. If true, layered windows will be included in the resulting image on Windows.

ImageSequence.all_frames

A new method to facilitate applying a given function to all frames in an image, or to all frames in a list of images. The frames are returned as a list of separate images. For example, `ImageSequence.all_frames(im, lambda im_frame: im_frame.rotate(90))` could be used to return all frames from an image, each rotated 90 degrees.

Variation fonts

Variation fonts are now supported, allowing for different styles from the same font file. `ImageFont.FreeTypeFont` has four new methods, `PIL.ImageFont.FreeTypeFont.get_variation_names()` and `PIL.ImageFont.FreeTypeFont.set_variation_by_name()` for using named styles, and `PIL.ImageFont.FreeTypeFont.get_variation_axes()` and `PIL.ImageFont.FreeTypeFont.set_variation_by_axes()` for using font axes instead. An IOError will be raised if the font is not a variation font. FreeType 2.9.1 or greater is required.

Other Changes

ImageTk.getimage

This function is now supported. It returns the contents of an `ImageTk.PhotoImage` as an RGBA `Image.Image` instance.

Image quality for JPEG compressed TIFF

The TIFF encoder accepts a `quality` parameter for jpeg compressed TIFF files. A value from 0 (worst) to 100 (best) controls the image quality, similar to the JPEG encoder. The default is 75. For example:

```python
im.save("out.tif", compression="jpeg", quality=85)
```

Improve encoding of TIFF tags

The TIFF encoder supports more types, especially arrays. This is required for the GeoTIFF format which encodes geospatial information.

- Pass `tagtype` from v2 directory to libtiff encoder, instead of autodetecting type.
- Use explicit types eg. `uint32_t` for `TIFF_LONG` to fix issues on platforms with 64-bit longs.
- Add support for multiple values (arrays). Requires type in v2 directory and values must be passed as a tuple.
- Add support for signed types eg. `TIFFTypes.TIFF_SIGNED_SHORT`. 
Respect PKG_CONFIG environment variable when building

This variable is commonly used by other build systems and using it can help with cross-compiling. Falls back to pkg-config as before.

Top-to-bottom complex text rendering

Drawing text in the ‘ttb’ direction with ImageFont has been significantly improved and requires Raqm 0.7 or greater.

1.6.11 6.0.0

Backwards Incompatible Changes

Python 3.4 dropped

Python 3.4 is EOL since 2019-03-16 and no longer supported. We will not be creating binaries, testing, or retaining compatibility with this version. The final version of Pillow for Python 3.4 is 5.4.1.

Removed deprecated PIL.OleFileIO

PIL.OleFileIO was removed as a vendored file and in Pillow 4.0.0 (2017-01) in favour of the upstream olefile Python package, and replaced with an ImportError. The deprecated file has now been removed from Pillow. If needed, install from PyPI (eg. pip install olefile).

Removed deprecated ImageOps functions

Several undocumented functions in ImageOps were deprecated in Pillow 4.3.0 (2017-10) and have now been removed: gaussian_blur, gblur, unsharp_mask, usm and box_blur. Use the equivalent operations in ImageFilter instead.

Removed deprecated VERSION

VERSION (the old PIL version, always 1.1.7) has been removed. Use __version__ instead.

API Changes

Deprecations

Python 2.7

Python 2.7 reaches end-of-life on 2020-01-01.

Pillow 7.0.0 will be released on 2020-01-01 and will drop support for Python 2.7, making Pillow 6.x the last series to support Python 2.
PyQt4 and PySide

Qt 4 reached end-of-life on 2015-12-19. Its Python bindings are also EOL: PyQt4 since 2018-08-31 and PySide since 2015-10-14.

Support for PyQt4 and PySide has been deprecated from ImageQt and will be removed in a future version. Please upgrade to PyQt5 or PySide2.

PIL.*ImagePlugin.__version__ attributes

These version constants have been deprecated and will be removed in a future version.

- BmpImagePlugin.__version__
- CurImagePlugin.__version__
- DcxImagePlugin.__version__
- EpsImagePlugin.__version__
- FliImagePlugin.__version__
- FpxImagePlugin.__version__
- GdImageFile.__version__
- GifImagePlugin.__version__
- IcoImagePlugin.__version__
- ImImagePlugin.__version__
- ImtImagePlugin.__version__
- IptcImagePlugin.__version__
- Jpeg2KImagePlugin.__version__
- JpegImagePlugin.__version__
- McIdasImagePlugin.__version__
- MicImagePlugin.__version__
- MpegImagePlugin.__version__
- MpoImagePlugin.__version__
- MspImagePlugin.__version__
- PalmImagePlugin.__version__
- PcdImagePlugin.__version__
- PcxImagePlugin.__version__
- PdfImagePlugin.__version__
- PixarImagePlugin.__version__
- PngImagePlugin.__version__
- PpmImagePlugin.__version__
- PsdImagePlugin.__version__
- SgiImagePlugin.__version__
• SunImagePlugin.__version__
• TgaImagePlugin.__version__
• TiffImagePlugin.__version__
• WmfImagePlugin.__version__
• XbmImagePlugin.__version__
• XpmImagePlugin.__version__
• XVThumbImagePlugin.__version__

Use PIL.__version__ instead.

ImageCms.CmsProfile attributes

Some attributes in ImageCms.CmsProfile have been deprecated since Pillow 3.2.0. From 6.0.0, they issue a DeprecationWarning:

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>Use instead</th>
</tr>
</thead>
<tbody>
<tr>
<td>color_space</td>
<td>Padded xcolor_space</td>
</tr>
<tr>
<td>pcs</td>
<td>Padded connection_space</td>
</tr>
<tr>
<td>product_copyright</td>
<td>Unicode copyright</td>
</tr>
<tr>
<td>product_desc</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_description</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_manufacturer</td>
<td>Unicode manufacturer</td>
</tr>
<tr>
<td>product_model</td>
<td>Unicode model</td>
</tr>
</tbody>
</table>

MIME type improvements

Previously, all JPEG2000 images had the MIME type “image/jpx”. This has now been corrected. After the file format drivers have been loaded, Image.MIME["JPEG2000"] will return “image/jp2”. ImageFile.get_format_mimetype will return “image/jpx” if a JPX profile is present, or “image/jp2” otherwise.

Previously, all SGI images had the MIME type “image/rgb”. This has now been corrected. After the file format drivers have been loaded, Image.MIME["SGI"] will return “image/sgi”. ImageFile.get_format_mimetype will return “image/rgb” if RGB image data is present, or “image/sgi” otherwise.

MIME types have been added to the PPM format. After the file format drivers have been loaded, Image.MIME["PPM"] will now return the generic “image/x-portable-anymap”. ImageFile.get_format_mimetype will return a MIME type specific to the color type.

The TGA, PCX and ICO formats also now have MIME types: “image/x-tga”, “image/x-pcx” and “image/x-icon” respectively.
API Additions

DIB file format

Pillow now supports reading and writing the Device Independent Bitmap file format.

Image.quantize

The `dither` option is now a customisable parameter (was previously hardcoded to 1). This parameter takes the same values used in `convert()`.

New language parameter

These text-rendering functions now accept a `language` parameter to request language-specific glyphs and ligatures from the font:

- `ImageDraw.ImageDraw.multiline_text()`
- `ImageDraw.ImageDraw.multiline_textsize()`
- `ImageDraw.ImageDraw.text()`
- `ImageDraw.ImageDraw.textsize()`
- `ImageFont.ImageFont.getmask()`
- `ImageFont.ImageFont.getsize_multiline()`
- `ImageFont.ImageFont.getsize()`

Added EXIF class

`getexif()` has been added, which returns an `Exif` instance. Values can be retrieved and set like a dictionary. When saving JPEG, PNG or WEBP, the instance can be passed as an `exif` argument to include any changes in the output image.

Added `ImageOps.exif_transpose`

`exif_transpose()` returns a copy of an image, transposed according to its EXIF Orientation tag.

PNG EXIF data

EXIF data can now be read from and saved to PNG images. However, unlike other image formats, EXIF data is not guaranteed to be present in `info` until `load()` has been called.
Other Changes

Reading new DDS image format

Pillow can now read uncompressed RGB data from DDS images.

Reading TIFF with old-style JPEG compression

Added support reading TIFF files with old-style JPEG compression through LibTIFF. All YCbCr TIFF images are now always read as RGB.

TIFF compression codecs

Support has been added for the LZMA, Zstd and WebP TIFF compression codecs.

Improved support for transposing I;16 images

I;16, I;16L and I;16B are now supported image modes for all `transpose()` operations.

1.6.12 5.4.1

This release fixes regressions in 5.4.0.

Installation on Termux

A change to the way Pillow detects libraries during installed prevented installation on Termux, which does not have `/sbin/ldconfig`. This is now fixed.

PNG: Handle IDAT chunks after image end

Some PNG images have multiple IDAT chunks. In some cases, Pillow will stop reading image data before the IDAT chunks finish. A regression caused an `EOFError` exception when previously there was none. This is now fixed, and file reading continues in case there are subsequent text chunks.

PNG: MIME type

The addition of limited APNG support to the PNG plugin also overwrote the MIME type for PNG files, causing “image/apng” to be returned as the MIME type of both APNG and PNG files. This has been fixed so the MIME type of PNG files is “image/png”.
File closing

A regression caused an unsupported image file to report a `ValueError: seek of closed file` exception instead of an `OSError`. This has been fixed by ensuring that image plugins only close their internal `__fp` if they are not the same as `ImageFile`'s `fp`, allowing each to manage their own file pointers.

1.6.13 5.4.0

API Changes

APNG extension to PNG plugin

Animated Portable Network Graphics (APNG) images are not fully supported but can be opened via the PNG plugin to get some basic info:

```python
im = Image.open("image.apng")
print(im.mode)  # "RGBA"
print(im.size)  # (245, 245)
im.show()  # Shows a single frame
```

Check for libjpeg-turbo

You can check if Pillow has been built against the libjpeg-turbo version of the libjpeg library:

```python
from PIL import features
features.check_feature("libjpeg_turbo")  # True or False
```

Negative indexes in pixel access

When accessing individual image pixels, negative indexes are now also accepted. For example, to get or set the farthest pixel in the lower right of an image:

```python
px = im.load()
print(px[-1, -1])
px[-1, -1] = (0, 0, 0)
```

New custom TIFF tags

TIFF images can now be saved with custom integer, float and string TIFF tags:

```python
im = Image.new("RGB", (200, 100))
custom = {
    37000: 4,
    37001: 4.2,
    37002: "custom tag value",
    37003: u"custom tag value",
    37004: b"custom tag value",
}
im.save("output.tif", tiffinfo=custom)
```

(continues on next page)
```python
im2 = Image.open("output.tif")
print(im2.tag_v2[37000])  # 4
print(im2.tag_v2[37002])  # "custom tag value"
print(im2.tag_v2[37004])  # b"custom tag value"
```

**Other Changes**

**ImageOps.fit**

Now uses one resize operation with box parameter internally instead of a crop and scale operations sequence. This improves the performance and accuracy of cropping since the box parameter accepts float values.

**1.6.14 5.3.0**

**API Changes**

**Image size**

If you attempt to set the size of an image directly, e.g. `im.size = (100, 100)`, you will now receive an `AttributeError`. This is not about removing existing functionality, but instead about raising an explicit error to prevent later consequences. The `resize` method is the correct way to change an image’s size.

The exceptions to this are:

- The ICO and ICNS image formats, which use `im.size = (100, 100)` to select a subimage.
- The TIFF image format, which now has a `DeprecationWarning` for this action, as direct image size setting was previously necessary to work around an issue with tile extents.

**API Additions**

**Added line width parameter to rectangle and ellipse-based shapes**

An optional line `width` parameter has been added to `ImageDraw.Draw.arc, chord, ellipse, pieslice` and `rectangle`.

**Curved joints for line sequences**

`ImageDraw.Draw.line` draws a line, or lines, between points. Previously, when multiple points are given, for a larger width, the joints between these lines looked unsightly. There is now an additional optional argument, `joint`, defaulting to `None`. When it is set to `curved`, the joints between the lines will become rounded.
ImageOps.colorize

Previously ImageOps.colorize only supported two-color mapping with black and white arguments being mapped to 0 and 255 respectively. Now it supports three-color mapping with the optional mid parameter, and the positions for all three color arguments can each be optionally specified (blackpoint, whitepoint and midpoint). For example, with all optional arguments:

```python
ImageOps.colorize(im, black=(32, 37, 79), white='white', mid=(59, 101, 175),
                 blackpoint=15, whitepoint=240, midpoint=100)
```

ImageOps.pad

While ImageOps.fit allows users to crop images to a requested aspect ratio and size, new method ImageOps.pad pads images to fill a requested aspect ratio and size, filling new space with a provided color and positioning the image within the new area through a centering argument.

Other Changes

Added support for reading tiled TIFF images through LibTIFF. Compressed TIFF images are now read through LibTIFF.

RGB WebP images are now read as RGB mode, rather than RGBX.

1.6.15 5.2.0

API Changes

Deprecations

These version constants have been deprecated. VERSION will be removed in Pillow 6.0.0, and PILLOW_VERSION will be removed after that.

- PIL.VERSION (old PIL version 1.1.7)
- PIL.PILLOW_VERSION
- PIL.Image.VERSION
- PIL.Image.PILLOW_VERSION

Use PIL.__version__ instead.

API Additions

3D color lookup tables

Support for 3D color lookup table transformations has been added.

- https://en.wikipedia.org/wiki/3D_lookup_table

Color3DLUT.generate transforms 3-channel pixels using the values of the channels as coordinates in the 3D lookup table and interpolating the nearest elements.

It allows you to apply almost any color transformation in constant time by using pre-calculated decimated tables.
Color3DLUT.transform() allows altering table values with a callback.

If NumPy is installed, the performance of argument conversion is dramatically improved when a source table supports buffer interface (NumPy && arrays in Python >= 3).

**ImageColor.getrgb**

Previously Image.rotate only supported HSL color strings. Now HSB and HSV strings are also supported, as well as float values. For example, ImageColor.getrgb("hsv(180,100%,99.5%)").

**ImageFile.get_format_mimetype**

ImageFile.get_format_mimetype has been added to return the MIME type of an image file, where available. For example, Image.open("hopper.jpg").get_format_mimetype() returns "image/jpeg".

**ImageFont.getsize_multiline**

A new method to return the size of multiline text, for example font.getsize_multiline("ABC\nAaaa")

**Image.rotate**

A new named parameter, fillcolor, has been added to Image.rotate. This color specifies the background color to use in the area outside the rotated image. This parameter takes the same color specifications as used in Image.new.

**TGA file format**

Pillow can now read and write LA data (in addition to L, P, RGB and RGBA), and write RLE data (in addition to uncompressed).

**Other Changes**

**Support added for Python 3.7**

Pillow 5.2 supports Python 3.7.

**Build macOS wheels with Xcode 6.4, supporting older macOS versions**

The macOS wheels for Pillow 5.1.0 were built with Xcode 9.2, meaning 10.12 Sierra was the lowest supported version. Prior to Pillow 5.1.0, Xcode 8 was used, supporting El Capitan 10.11.

Instead, Pillow 5.2.0 is built with the oldest available Xcode 6.4 to support at least 10.10 Yosemite.
Fix `_i2f` compilation with some GCC versions

For example, this allows compilation with GCC 4.8 on NetBSD.

Resolve confusion getting PIL / Pillow version string

Re: “version constants deprecated” listed above, as user gnbl notes in #3082:

- it’s confusing that PIL.VERSION returns the version string of the former PIL instead of Pillow’s
- there does not seem to be documentation on this version number (why this, will it ever change, ..) e.g. at https://pillow.readthedocs.io/en/5.1.x/about.html#why-a-fork
- it’s confusing that PIL.version is a module and does not return the version information directly or hints on how to get it
- the package information header is essentially useless (placeholder, does not even mention Pillow, nor the version)
- PIL._version module documentation comment could explain how to access the version information

We have attempted to resolve these issues in #3083, #3090 and #3218.

1.6.16 5.1.0

New File Format

BLP File Format

Pillow now supports reading the BLP “Blizzard Mipmap” file format used for tiles in Blizzard’s engine.

API Changes

Optional channels for TIFF files

Pillow can now open TIFF files with base modes of RGB, YCbCr, and CMYK with up to 6 8-bit channels, discarding any extra channels if the content is tagged as UNSPECIFIED. Pillow still does not store more than 4 8-bit channels of image data.

Append to PDF Files

Images can now be appended to PDF files in place by passing in `append=True` when saving the image.
Other Changes

WebP memory leak

A memory leak when opening WebP files has been fixed.

1.6.17 5.0.0

Backwards Incompatible Changes

Python 3.3 Dropped

Python 3.3 is EOL and no longer supported due to moving testing from nose, which is deprecated, to pytest, which doesn’t support Python 3.3. We will not be creating binaries, testing, or retaining compatibility with this version. The final version of Pillow for Python 3.3 is 4.3.0.

Decompression Bombs now raise Exceptions

Pillow has previously emitted warnings for images that are unexpectedly large and may be a denial of service. These warnings are now upgraded to DecompressionBombErrors for images that are twice the size of images that trigger the DecompressionBombWarning. The default threshold is 128Mpx, or 0.5GB for an RGB or RGBA image. This can be disabled or changed by setting `Image.MAX_IMAGE_PIXELS = None`.

Scripts

The scripts formerly installed by Pillow have been split into a separate package, pillow-scripts, living at https://github.com/python-pillow/pillow-scripts.

API Changes

OleFileIO.py

The olefile module is no longer a required dependency when installing Pillow. Support for plugins requiring olefile will not be loaded if it is not installed. This allows library consumers to avoid installing this dependency if they choose. Some library consumers have little interest in the format support and would like to keep dependencies to a minimum.

Further, the vendored version was removed in Pillow 4.0.0 and replaced with a deprecation warning that PIL.OleFileIO would be removed in a future version. This warning has been upgraded to an import error pending future removal.

Check parameter on _save

Several image plugins supported a named `check` parameter on their nominally private `_save` method to preflight if the image could be saved in that format. That parameter has been removed.
API Additions

Image.transform

A new named parameter, fillcolor, has been added to Image.transform. This color specifies the background color to use in the area outside the transformed area in the output image. This parameter takes the same color specifications as used in Image.new.

GIF Disposal

Multiframe GIF images now take an optional disposal parameter to specify the disposal option for changed pixels.

Other Changes

Compressed TIFF Images

Previously, there were some compression modes (JPEG, Packbits, and LZW) that were supported with Pillow’s internal TIFF decoder. All compressed TIFFs are now read using the libtiff decoder, as it implements the compression schemes more correctly.

Libraqm is now Dynamically Linked

The libraqm dependency for complex text scripts is now linked dynamically at runtime rather than at packaging time. This allows us to release binaries with support for libraqm if it is installed on the user’s machine.

Source Layout Changes

The Pillow source is now stored within the src directory of the distribution. This prevents accidental imports of the PIL directory when running Python from the project directory.

Setup.py Changes

Multiarch support on Linux should be more robust, especially on Debian derivatives on ARM platforms. Debian’s multiarch platform configuration is run in preference to the sniffing of machine platform and architecture.

1.6.18 4.3.0

API Changes

Deprecations

Several undocumented functions in ImageOps have been deprecated: gaussian_blur, gblur, unsharp_mask, usm and box_blur. Use the equivalent operations in ImageFilter instead. These functions will be removed in a future release.
TIFF Metadata Changes

- TIFF tags with unknown type/quantity now default to being bare values if they are 1 element, where previously they would be a single element tuple. This is only with the new api, not the legacy api. This normalizes the handling of fields, so that the metadata with inferred or image specified counts are handled the same as metadata with count specified in the TIFF spec.

- The PhotoshopInfo, XMP, and JPEGTables tags now have a defined type (bytes) and a count of 1.
- The ImageJMetaDateByteCounts tag now has an arbitrary number of items, as there can be multiple items, one for UTF-8, and one for UTF-16.

Core Image API Changes

These are internal functions that should not have been used by user code, but they were accessible from the python layer.

Debugging code within Image.core.grabclipboard was removed. It had been marked as will be removed in future versions since PIL. When enabled, it identified the format of the clipboard data.

The PIL.Image.core.copy and PIL.Image.Image.im.copy2 methods have been removed.

The PIL.Image.core.getcount methods have been removed, use PIL.Image.core.get_stats()['new_count'] property instead.

API Additions

Get One Channel From Image

A new method PIL.Image.Image.getchannel() has been added to return a single channel by index or name. For example, image.getchannel("A") will return alpha channel as separate image. getchannel should work up to 6 times faster than image.split()[0] in previous Pillow versions.

Box Blur

A new filter, PIL.ImageFilter.BoxBlur, has been added. This is a filter with similar results to a Gaussian blur, but is much faster.

Partial Resampling

Added new argument box for PIL.Image.Image.resize(). This argument defines a source rectangle from within the source image to be resized. This is very similar to the image.crop(box).resize(size) sequence except that box can be specified with subpixel accuracy.
New Transpose Operation

The `Image.TRANSVERSE` operation has been added to `PIL.Image.Image.transpose()`. This is equivalent to a transpose operation about the opposite diagonal.

Multiband Filters

There is a new `PIL.ImageFilter.MultibandFilter` base class for image filters that can run on all channels of an image in one operation. The original `PIL.ImageFilter.Filter` class remains for image filters that can process only single band images, or require splitting of channels prior to filtering.

Other Changes

Loading 16-bit TIFF Images

Pillow now can read 16-bit multichannel TIFF files including files with alpha transparency. The image data is truncated to 8-bit precision.

Pillow now can read 16-bit signed integer single channel TIFF files. The image data is promoted to 32-bit for storage and processing.

SGI Images

Pillow can now read and write uncompressed 16-bit multichannel SGI images to and from RGB and RGBA formats. The image data is truncated to 8-bit precision.

Pillow can now read RLE encoded SGI images in both 8 and 16-bit precision.

Performance

This release contains several performance improvements:

- Many memory bandwidth-bounded operations such as crop, image allocation, conversion, split into bands and merging from bands are up to 2x faster.
- Upscaling of multichannel images (such as RGB) is accelerated by 5-10%
- JPEG loading is accelerated up to 15% and JPEG saving up to 20% when using a recent version of libjpeg-turbo.
- `Image.transpose` has been accelerated 15% or more by using a cache friendly algorithm.
- ImageFilters based on Kernel convolution are significantly faster due to the new `MultibandFilter` feature.
- All memory allocation for images is now done in blocks, rather than falling back to an allocation for each scan line for images larger than the block size.
CMYK Conversion

The basic CMYK->RGB conversion has been tweaked to match the formula from Google Chrome. This produces an image that is generally lighter than the previous formula, and more in line with what color managed applications produce.

1.6.19 4.2.1

There are no functional changes in this release.

Fixed Windows PyPy Build

A change in the 4.2.0 cycle broke the Windows PyPy build. This has been fixed, and PyPy is now part of the Windows CI matrix.

1.6.20 4.2.0

Added Complex Text Rendering

Pillow now supports complex text rendering for scripts requiring glyph composition and bidirectional flow. This optional feature adds three dependencies: harfbuzz, fribidi, and raqm. See the install documentation for further details. This feature is tested and works on Unix and Mac, but has not yet been built on Windows platforms.

New Optional Parameters

- `PIL.ImageDraw.floodfill()` has a new optional parameter: threshold. This specifies a tolerance for the color to replace with the flood fill.
- The TIFF and PDF image writers now support the `append_images` optional parameter for specifying additional images to create multipage outputs.

New DecompressionBomb Warning

`PIL.Image.Image.crop()` now may raise a DecompressionBomb warning if the crop region enlarges the image over the threshold specified by `PIL.Image.MAX_IMAGE_PIXELS`.

Removed Deprecated Items

Several deprecated items have been removed.

- The methods `PIL.ImageWin.Dib.fromstring`, `PIL.ImageWin.Dib.tostring` and `PIL.TiffImagePlugin.ImageFileDirectory_v2.as_dict` have been removed.
- Before Pillow 4.2.0, attempting to save an RGBA image as JPEG would discard the alpha channel. From Pillow 3.4.0, a deprecation warning was shown. From Pillow 4.2.0, the deprecation warning is removed and an IOError is raised.
Removed Core Image Function

The unused function `Image.core.new_array` was removed. This is an internal function that should not have been used by user code, but it was accessible from the python layer.

1.6.21 4.1.1

Fix Regression with reading DPI from EXIF data

Some JPEG images don’t contain DPI information in the image metadata, but do contain it in the EXIF data. A patch was added in 4.1.0 to read from the EXIF data, but it did not accept all possible types that could be included there. This fix adds the ability to read ints as well as rational values.

Incompatibility between 3.6.0 and 3.6.1

CPython 3.6.1 added a new symbol, PySlice_GetIndicesEx, which was not present in 3.6.0. This had the effect of causing binaries compiled on CPython 3.6.1 to not work on installations of C-Python 3.6.0. This fix undefines PySlice_GetIndicesEx if it exists to restore compatibility with both 3.6.0 and 3.6.1. See https://bugs.python.org/issue29943 for more details.

1.6.22 4.1.0

Removed Deprecated Items

Several deprecated items have been removed.

- Support for spaces in tiff kwargs in the parameters for ‘x resolution’, ‘y resolution’, ‘resolution unit’, and ‘date time’ has been removed. Underscores should be used instead.

Closing Files When Opening Images

The file handling when opening images has been overhauled. Previously, Pillow would attempt to close some, but not all image formats after loading the image data. Now, the following behavior is specified:

- For images where an open file is passed in, it is the responsibility of the calling code to close the file.
- For images where Pillow opens the file and the file is known to have only one frame, the file is closed after loading.
- If the file has more than one frame, or if it can’t be determined, then the file is left open to permit seeking to subsequent frames. It will be closed, eventually, in the close or __del__ methods.
- If the image is memory mapped, then we can’t close the mapping to the underlying file until we are done with the image. The mapping will be closed in the close or __del__ method.
Changes to GIF Handling When Saving

The PIL.GifImagePlugin code has been refactored to fix the flow when saving images. There are two external changes that arise from this:

- An PIL.ImagePalette.ImagePalette object is now accepted as a specified palette argument in PIL.Image.Image.save().
- The image to be saved is no longer modified in place by any of the operations of the save function. Previously it was modified when optimizing the image palette.

This refactor fixed some bugs with palette handling when saving multiple frame GIFs.

New Method: Image.remap_palette

The method PIL.Image.Image.remap_palette() has been added. This method was hoisted from the GifImagePlugin code used to optimize the palette.

Added Decoder Registry and Support for Python Based Decoders

There is now a decoder registry similar to the image plugin registries. Image plugins can register a decoder, and it will be called when the decoding is requested. This allows for the creation of pure Python decoders. While the Python decoders will not be as fast as their C based counterparts, they may be easier and quicker to develop or safer to run.

Tests

Many tests have been added, including correctness tests for image formats that have been previously untested.

We are now running automated tests in Docker containers against more Linux versions than are provided on Travis CI, which is currently Ubuntu 14.04 x64. This Pillow release is tested on 64-bit Alpine, Arch, Ubuntu 12.04 and 16.04, and 32-bit Debian Stretch and Ubuntu 14.04. This also covers a wider range of dependency versions than are provided on Travis natively.

1.6.23 4.0.0

Python 2.6 and 3.2 Dropped

Pillow 4.0 no longer supports Python 2.6 and 3.2. We will not be creating binaries, testing, or retaining compatibility with these releases. This release removes some workarounds for those Python releases, so the final working version of Pillow on 2.6 or 3.2 is 3.4.2.

Support added for Python 3.6

Pillow 4.0 supports Python 3.6.
OleFileIO.py

OleFileIO.py has been removed as a vendored file and is now installed from the upstream olefile pypi package. All internal dependencies are redirected to the olefile package. Direct accesses to PIL.OlefileIO raises a deprecation warning, then patches the upstream olefile into sys.modules in its place.

SGI image save

It is now possible to save images in modes L, RGB, and RGBA to the uncompressed SGI image format.

Zero sized images

Pillow 3.4.0 removed support for creating images with (0,0) size. This has been reenabled, restoring pre 3.4 behavior.

Internal handles_eof flag

The handles_eof flag for decoding images has been removed, as there were no internal users of the flag. Anyone maintaining image decoders outside of the Pillow source tree should consider using the cleanup function pointers instead.

Image.core.stretch removed

The stretch function on the core image object has been removed. This used to be for enlarging the image, but has been aliased to resize recently.

1.6.24 3.4.0

New resizing filters

Two new filters available for Image.resize() and Image.thumbnail() functions: BOX and HAMMING. BOX is the high-performance filter with two times shorter window than BILINEAR. It can be used for image reduction 3 and more times and produces a sharper result than BILINEAR.

HAMMING filter has the same performance as BILINEAR filter while providing the image downscaling quality comparable to BICUBIC. Both new filters don’t show good quality for the image upscaling.

Deprecation Warning when Saving JPEGs

JPEG images cannot contain an alpha channel. Pillow prior to 3.4.0 silently drops the alpha channel. With this release Pillow will now issue a DeprecationWarning when attempting to save a RGBA mode image as a JPEG. This will become an error in Pillow 4.2.
New DDS Decoders

Pillow can now decode DXT3 images, as well as the previously supported DXT1 and DXT5 formats. All three formats are now decoded in C code for better performance.

Append images to GIF

Additional frames can now be appended when saving a GIF file, through the append_images argument. The new frames are passed in as a list of images, which may be have multiple frames themselves.

Note that the append_images argument is only used if save_all is also in effect, e.g.:

```python
im.save(out, save_all=True, append_images=[im1, im2, ...])
```

Save multiple frame TIFF

Multiple frames can now be saved in a TIFF file by using the save_all option, e.g.:

```python
im.save("filename.tiff", format="TIFF", save_all=True)
```

Image.core.open_ppm removed

The nominally private/debugging function Image.core.open_ppm has been removed. If you were using this function, please use Image.open instead.

1.6.25 3.3.2

Integer overflow in Map.c

Pillow prior to 3.3.2 may experience integer overflow errors in map.c when reading specially crafted image files. This may lead to memory disclosure or corruption.

Specifically, when parameters from the image are passed into Image.core.map_buffer, the size of the image was calculated with xsize * ysize * bytes_per_pixel. This will overflow if the result is larger than SIZE_MAX. This is possible on a 32-bit system.

Furthermore this size value was added to a potentially attacker provided offset value and compared to the size of the buffer without checking for overflow or negative values.

These values were then used for creating pointers, at which point Pillow could read the memory and include it in other images. The image was marked readonly, so Pillow would not ordinarily write to that memory without duplicating the image first.

This issue was found by Cris Neckar at Divergent Security.
Sign Extension in Storage.c

Pillow prior to 3.3.2 and PIL 1.1.7 (at least) do not check for negative image sizes in ImagingNew in Storage.c. A negative image size can lead to a smaller allocation than expected, leading to arbitrary writes.

This issue was found by Cris Neckar at Divergent Security.

1.6.26 3.3.0

Libimagequant support

There is now support for using libimagequant as a higher quality quantization option in Image.quantize() on Unix-like platforms. This support requires building Pillow from source against libimagequant. We cannot distribute binaries due to licensing differences.

New Setup.py options

There are two new options to control the build_ext task in setup.py:

- --debug dumps all of the directories and files that are checked when searching for libraries or headers when building the extensions.
- --disable-platform-guessing removes many of the directories that are checked for libraries and headers for build systems or cross compilers that specify that information in via environment variables.

Resizing

Image resampling for 8-bit per channel images was rewritten using only integer computings. This is faster on most platforms and doesn’t introduce precision errors on the wide range of scales. With other performance improvements, this makes resampling 60% faster on average.

Color calculation for images in the LA mode on semitransparent pixels was fixed.

Rotation

Rotation for angles divisible by 90 degrees now always uses transposition. This greatly improves both quality and performance in this case. Also, the bug with wrong image size calculation when rotating by 90 degrees was fixed.

Image Metadata

The return type for binary data in version 2 Exif and Tiff metadata has been changed from a tuple of integers to bytes. This is a change from the behavior since 3.0.0.
1.6.27 3.2.0

New DDS and FTEX Image Plugins

The DdsImagePlugin reading DXT1 and DXT5 encoded .dds images was added. DXT3 images are not currently supported.

The FtexImagePlugin reads textures used for 3D objects in Independence War 2: Edge Of Chaos. The plugin reads a single texture per file, in the .ftc (compressed) and .ftu (uncompressed) formats.

Updates to the GbrImagePlugin

The GbrImagePlugin (GIMP brush format) has been updated to fix support for version 1 files and add support for version 2 files.

Passthrough Parameters for ImageDraw.text

ImageDraw.multiline_text and ImageDraw.multiline_size take extra spacing parameters above what are used in ImageDraw.text and ImageDraw.size. These parameters can now be passed into ImageDraw.text and ImageDraw.size and they will be passed through to the corresponding multiline functions.

ImageSequence.Iterator changes

ImageSequence.Iterator is now an actual iterator implementing the Iterator protocol. It is also now possible to seek to the first image of the file when using direct indexing.

1.6.28 3.1.2

CVE-2016-3076 – Buffer overflow in Jpeg2KEncode.c

Pillow between 2.5.0 and 3.1.1 may overflow a buffer when writing large Jpeg2000 files, allowing for code execution or other memory corruption.

This occurs specifically in the function j2k_encode_entry, at the line:

```
state->buffer = malloc (tile_width * tile_height * components * prec / 8);
```

This vulnerability requires a particular value for `height * width` such that `height * width * components * precision` overflows, at which point the malloc will be for a smaller value than expected. The buffer that is allocated will be `((height * width * components * precision) mod (2^31) / 8)`, where components is 1-4 and precision is either 8 or 16. Common values would be 4 components at precision 8 for a standard RGBA image.

The unpackers then split an image that is laid out:

```
RGBARGBARGBA....
```

into:
If this buffer is smaller than expected, the jpeg2k unpacker functions will write outside the allocation and onto the heap, corrupting memory.

This issue was found by Alyssa Besseling at Atlassian.

### 1.6.29 3.1.1

**CVE-2016-0740 – Buffer overflow in TiffDecode.c**

Pillow 3.1.0 and earlier when linked against libtiff >= 4.0.0 on x64 may overflow a buffer when reading a specially crafted tiff file.

Specifically, libtiff >= 4.0.0 changed the return type of TIFFScanlineSize from `int32` to machine dependent `int32|64`. If the scanline is sized so that it overflows an `int32`, it may be interpreted as a negative number, which will then pass the size check in TiffDecode.c line 236. To do this, the logical scanline size has to be > 2gb, and for the test file, the allocated buffer size is 64k against a roughly 4gb scan line size. Any image data over 64k is written over the heap, causing a segfault.

This issue was found by security researcher FourOne.

**CVE-2016-0775 – Buffer overflow in FliDecode.c**

In all versions of Pillow, dating back at least to the last PIL 1.1.7 release, FliDecode.c has a buffer overflow error.

Around line 192:

```c
case 16:
    /* COPY chunk */
    for (y = 0; y < state->ysize; y++) {
        UINT8* buf = (UINT8*) im->image[y];
        memcpy(buf+x, data, state->xsize);
        data += state->xsize;
    }
    break;
```

The memcpy has error where `x` is added to the target buffer address. `X` is used in several internal temporary variable roles, but can take a value up to the width of the image. `Im->image[y]` is a set of row pointers to segments of memory that are the size of the row. At the max `y`, this will write the contents of the line off the end of the memory buffer, causing a segfault.

This issue was found by Alyssa Besseling at Atlassian
CVE-2016-2533 – Buffer overflow in PcdDecode.c

In all versions of Pillow, dating back at least to the last PIL 1.1.7 release, `PcdDecode.c` has a buffer overflow error. The `state.buffer` for `PcdDecode.c` is allocated based on a 3 bytes per pixel sizing, where `PcdDecode.c` wrote into the buffer assuming 4 bytes per pixel. This writes 768 bytes beyond the end of the buffer into other Python object storage. In some cases, this causes a segfault, in others an internal Python malloc error.

Integer overflow in Resample.c

If a large value was passed into the new size for an image, it is possible to overflow an int32 value passed into malloc.

```
    kk = malloc(xsize * kmax * sizeof(float)); . . . xbounds = malloc(xsize * 2 * sizeof(int));
```

`xsize` is trusted user input. These multiplications can overflow, leading the malloc’d buffer to be undersized. These allocations are followed by a loop that writes out of bounds. This can lead to corruption on the heap of the Python process with attacker controlled float data.

This issue was found by Ned Williamson.

1.6.30  3.1.0

ImageDraw arc, chord and pieslice can now use floats

There is no longer a need to ensure that the start and end arguments for `arc`, `chord` and `pieslice` are integers.

Note that these numbers are not simply rounded internally, but are actually utilised in the drawing process.

Consistent multiline text spacing

When using the `ImageDraw` multiline methods, the spacing between lines was inconsistent, based on the combination on ascenders and descenders.

This has now been fixed, so that lines are offset by their baselines, not the absolute height of each line.

There is also now a default spacing of 4px between lines.

Exif, Jpeg and Tiff Metadata

There were major changes in the TIFF ImageFileDirectory support in Pillow 3.0 that led to a number of regressions. Some of them have been fixed in Pillow 3.1, and some of them have been extended to have different behavior.

TiffImagePlugin.IFDRational

Pillow 3.0 changed rational metadata to use a float. In Pillow 3.1, this has changed to allow the expression of 0/0 as a valid piece of rational metadata to reflect usage in the wild.

Rational metadata is now encapsulated in an `IFDRational` instance. This class extends the Rational class to allow a denominator of 0. It compares as a float or a number, but does allow access to the raw numerator and denominator values through attributes.

When used in a `ImageFileDirectory_v1`, a 2 item tuple is returned of the numerator and denominator, as was done previously.

This class should be used when adding a rational value to an ImageFileDirectory for saving to image metadata.
JpegImagePlugin._getexif

In Pillow 3.0, the dictionary returned from the private, experimental, but generally widely used _getexif function changed to reflect the ImageFileDirectory_v2 format, without a fallback to the previous format.

In Pillow 3.1, _getexif now returns a dictionary compatible with Pillow 2.9 and earlier, built with ImageFileDirectory_v1 instances. Additionally, any single item tuples have been unwrapped and return a bare element.

The format returned by Pillow 3.0 has been abandoned. A more fully featured interface for EXIF is anticipated in a future release.

Out of Spec Metadata

In Pillow 3.0 and 3.1, images that contain metadata that is internally consistent, but not in agreement with the TIFF spec, may cause an exception when reading the metadata. This can happen when a tag that is specified to have a single value is stored with an array of values.

It is anticipated that this behavior will change in future releases.

1.6.31 3.0.0

Saving Multipage Images

There is now support for saving multipage images in the GIF and PDF formats. To enable this functionality, pass in save_all=True as a keyword argument to the save:

```python
im.save('test.pdf', save_all=True)
```

Tiff ImageFileDirectory Rewrite

The Tiff ImageFileDirectory metadata code has been rewritten. Where previously it returned a somewhat arbitrary set of values and tuples, it now returns bare values where appropriate and tuples when the metadata item is a sequence or collection.

The original metadata is still available in the TiffImage.tags, the new values are available in the TiffImage.tags_v2 member. The old structures will be deprecated at some point in the future. When saving Tiff metadata, new code should use the TiffImagePlugin.ImageFileDirectory_v2 class.

Deprecated Methods

Several methods that have been marked as deprecated for many releases have been removed in this release:

```python
Image.tostring()
Image.fromstring()
Image.offset()
ImageDraw.setink()
ImageDraw.setfill()
The ImageFileIO module
The ImageFont.FreeTypeFont and ImageFont.truetype \`\`file\`\` keyword arg
The ImagePalette private _make functions
ImageWin.fromstring()
ImageWin.tostring()
```
LibJpeg and Zlib are Required by Default

The external dependencies on libjpeg and zlib are now required by default. If the headers or libraries are not found, then installation will abort with an error. This behaviour can be disabled with the \--disable-libjpeg and \--disable-zlib flags.

1.6.32 2.8.0

Open HTTP response objects with Image.open

HTTP response objects returned from urllib2.urlopen(url) or requests.get(url, stream=True). raw are ‘file-like’ but do not support .seek() operations. As a result PIL was unable to open them as images, requiring a wrap in cStringIO or BytesIO.

Now new functionality has been added to Image.open() by way of an .seek(0) check and catch on exception AttributeError or io.UnsupportedOperation. If this is caught we attempt to wrap the object using io. BytesIO (which will only work on buffer-file-like objects).

This allows opening of files using both urllib2 and requests, e.g.:

```
Image.open(urllib2.urlopen(url))
Image.open(requests.get(url, stream=True).raw)
```

If the response uses content-encoding (compression, either gzip or deflate) then this will fail as both the urllib2 and requests raw file object will produce compressed data in that case. Using Content-Encoding on images is rather non-sensical as most images are already compressed, but it can still happen.

For requests the work-around is to set the decode_content attribute on the raw object to True:

```
response = requests.get(url, stream=True)
response.raw.decode_content = True
image = Image.open(response.raw)
```

1.6.33 2.7.0

Sane Plugin

The Sane plugin has now been split into its own repo: https://github.com/python-pillow/Sane.

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.
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.

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.

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 depend 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.
TIFF 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’)

1.7 Deprecations and removals

This page lists Pillow features that are deprecated, or have been removed in past major releases, and gives the alternatives to use instead.

1.7.1 Deprecated features

Below are features which are considered deprecated. Where appropriate, a DeprecationWarning is issued.

Image.show command parameter

Deprecated since version 7.2.0. The command parameter was deprecated and will be removed in a future release. Use a subclass of ImageShow.Viewer instead.

Image._showxv

Deprecated since version 7.2.0. Image._showxv has been deprecated. Use Image.Image.show() instead. If custom behaviour is required, use ImageShow.register() to add a custom ImageShow.Viewer class.

ImageFile.raise_ioerror

Deprecated since version 7.2.0. IOError was merged into OSError in Python 3.3. So, ImageFile.raise_ioerror is now deprecated and will be removed in a future release. Use ImageFile.raise_oserror instead.

PILLOW_VERSION constant

Deprecated since version 5.2.0. PILLOW_VERSION has been deprecated and will be removed in a future release. Use __version__ instead. It was initially removed in Pillow 7.0.0, but brought back in 7.1.0 to give projects more time to upgrade.
1.7.2 Removed features

Deprecated features are only removed in major releases after an appropriate period of deprecation has passed.

**im.offset**

Deprecated since version 1.1.2.
Removed in version 8.0.0.
im.offset() has been removed, call ImageChops.offset() instead.

It was documented as deprecated in PIL 1.1.2, raised a DeprecationWarning since 1.1.5, an Exception since Pillow 3.0.0 and NotImplementedError since 3.3.0.

**Image.fromstring, im.fromstring and im.tostring**

Deprecated since version 2.0.0.
Removed in version 8.0.0.

- Image.fromstring() has been removed, call Image.frombytes() instead.
- im.fromstring() has been removed, call frombytes() instead.
- im.tostring() has been removed, call tobytes() instead.

They issued a DeprecationWarning since 2.0.0, an Exception since 3.0.0 and NotImplementedError since 3.3.0.

**ImageCms.CmsProfile attributes**

Deprecated since version 3.2.0.
Removed in version 8.0.0.

Some attributes in PIL.ImageCms.CmsProfile have been removed. From 6.0.0, they issued a DeprecationWarning:

<table>
<thead>
<tr>
<th>Removed</th>
<th>Use instead</th>
</tr>
</thead>
<tbody>
<tr>
<td>color_space</td>
<td>Padded xcolor_space</td>
</tr>
<tr>
<td>pcs</td>
<td>Padded connection_space</td>
</tr>
<tr>
<td>product_copyright</td>
<td>Unicode copyright</td>
</tr>
<tr>
<td>product_desc</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_description</td>
<td>Unicode profile_description</td>
</tr>
<tr>
<td>product_manufacturer</td>
<td>Unicode manufacturer</td>
</tr>
<tr>
<td>product_model</td>
<td>Unicode model</td>
</tr>
</tbody>
</table>
Python 2.7

Deprecated since version 6.0.0.
Removed in version 7.0.0.

Python 2.7 reached end-of-life on 2020-01-01. Pillow 6.x was the last series to support Python 2.

Image.__del__

Deprecated since version 6.1.0.
Removed in version 7.0.0.

Implicitly closing the image’s underlying file in Image.__del__ has been removed. Use a context manager or call Image.close() instead to close the file in a deterministic way.

Previous method:

```python
im = Image.open("hopper.png")
im.save("out.jpg")
```

Use instead:

```python
with Image.open("hopper.png") as im:
    im.save("out.jpg")
```

PIL.*ImagePlugin.__version__ attributes

Deprecated since version 6.0.0.
Removed in version 7.0.0.

The version constants of individual plugins have been removed. Use PIL.__version__ instead.
### Removed

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td>ImImagePlugin.</td>
<td>PcxImagePlugin.</td>
<td>XpmImagePlugin.</td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
</tr>
<tr>
<td>IptcImagePlugin.</td>
<td>PixarImagePlugin.</td>
<td></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td><strong>version</strong></td>
<td></td>
</tr>
</tbody>
</table>

### PyQt4 and PySide

Deprecated since version 6.0.0.

Removed in version 7.0.0.

Qt 4 reached end-of-life on 2015-12-19. Its Python bindings are also EOL: PyQt4 since 2018-08-31 and PySide since 2015-10-14.

Support for PyQt4 and PySide has been removed from ImageQt. Please upgrade to PyQt5 or PySide2.

### Setting the size of TIFF images

Deprecated since version 5.3.0.

Removed in version 7.0.0.

Setting the size of a TIFF image directly (eg. `im.size = (256, 256)`) throws an error. Use `Image.resize` instead.
VERSION constant

Deprecated since version 5.2.0.
Removed in version 6.0.0.

VERSION (the old PIL version, always 1.1.7) has been removed. Use __version__ instead.

Undocumented ImageOps functions

Deprecated since version 4.3.0.
Removed in version 6.0.0.

Several undocumented functions in ImageOps have been removed. Use the equivalents in ImageFilter instead:

<table>
<thead>
<tr>
<th>Removed</th>
<th>Use instead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageOps.box_blur</td>
<td>ImageFilter.BoxBlur</td>
</tr>
<tr>
<td>ImageOps.gaussian_blur</td>
<td>ImageFilter.GaussianBlur</td>
</tr>
<tr>
<td>ImageOps.gblur</td>
<td>ImageFilter.GaussianBlur</td>
</tr>
<tr>
<td>ImageOps.usm</td>
<td>ImageFilter.UnsharpMask</td>
</tr>
<tr>
<td>ImageOps.unsharp_mask</td>
<td>ImageFilter.UnsharpMask</td>
</tr>
</tbody>
</table>

PIL.OleFileIO

Deprecated since version 4.0.0.
Removed in version 6.0.0.

PIL.OleFileIO was removed as a vendored file and in Pillow 4.0.0 (2017-01) in favour of the upstream olefile Python package, and replaced with an ImportError in 5.0.0 (2018-01). The deprecated file has now been removed from Pillow. If needed, install from PyPI (eg. python3 -m pip install olefile).
CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search
p
PIL, 129
PIL._binary, 158
PIL._imaging, 159
PIL._tkinter_finder, 159
PIL._util, 159
PIL._version, 159
PIL.BdfFontFile, 129
PIL.BmpImagePlugin, 136
PIL.BufrStubImagePlugin, 136
PIL.ContainerIO, 129
PIL.CurImagePlugin, 137
PIL.DcxImagePlugin, 137
PIL.EpsImagePlugin, 137
PIL.ExifTags, 121
PIL.features, 126
PIL.FitsStubImagePlugin, 138
PIL.FliImagePlugin, 138
PIL.FontFile, 130
PIL.FpxImagePlugin, 138
PIL.GbrImagePlugin, 139
PIL.GdImageFile, 130
PIL.GifImagePlugin, 139
PIL.GimpGradientFile, 131
PIL.GimpPaletteFile, 131
PIL.GribStubImagePlugin, 140
PIL.Hdf5StubImagePlugin, 140
PIL.IcnsImagePlugin, 140
PIL.IcoImagePlugin, 141
PIL.Image, 49
PIL.Image.core, 159
PIL.ImageChops, 70
PIL.ImageCms, 73
PIL.ImageColor, 85
PIL.ImageDraw, 86
PIL.ImageDraw2, 131
PIL.ImageEnhance, 94
PIL.ImageFile, 95
PIL.ImageFilter, 97
PIL.ImageFont, 100
PIL.ImageGrab, 106
PIL.ImageMath, 107
PIL.ImageMorph, 109
PIL.ImageOps, 110
PIL.ImagePalette, 114
PIL.ImagePath, 115
PIL.ImageQt, 116
PIL.ImageSequence, 116
PIL.ImageShow, 116
PIL.ImageStat, 118
PIL.ImageTk, 118
PIL.ImageTransform, 133
PIL.ImageWin, 119
PIL.ImImagePlugin, 142
PIL.ImtImagePlugin, 142
PIL.IptcImagePlugin, 143
PIL.Jpeg2KImagePlugin, 144
PIL.JpegImagePlugin, 143
PIL.JpegPresets, 122
PIL.McIdasImagePlugin, 144
PIL.MicImagePlugin, 144
PIL.MpegImagePlugin, 145
PIL.MspImagePlugin, 145
PIL.PaletteFile, 134
PIL.PalmImagePlugin, 145
PIL.PcdImagePlugin, 145
PIL.PcffFontFile, 134
PIL.PcxImagePlugin, 146
PIL.PdfImagePlugin, 146
PIL.PixarImagePlugin, 146
PIL.PngImagePlugin, 146
PIL.PpmImagePlugin, 148
PIL.PsdImagePlugin, 148
PIL.PSDraw, 123
PIL.PyAccess, 125
PIL.SgiImagePlugin, 149
PIL.SpiderImagePlugin, 149
PIL.SunImagePlugin, 150
PIL.TarIO, 135
PIL.TgaImagePlugin, 150
PIL.TiffImagePlugin, 150
PIL.TiffTags, 121
PIL.WalImageFile, 135
PIL.WebPImagePlugin, 154
PIL.WmfImagePlugin, 154
PIL.XbmImagePlugin, 155
PIL.XpmImagePlugin, 155
PIL.XVThumbImagePlugin, 154
INDEX

Symbols

_Enhance (class in PIL.ImageEnhance), 94
__init__ () (PIL.TiffTags.TagInfo method), 121
__new__ () (PIL.PngImagePlugin.iTXt method), 134
__version__ (in module PIL._version), 159

A
ADAPTIVE (in module PIL.Image), 70
add () (in module PIL.ImageChops), 70
add () (PIL.PngImagePlugin.PngInfo method), 135
add_itxt () (PIL.PngImagePlugin.PngInfo method), 135
add_modulo () (in module PIL.ImageChops), 70
add_text () (PIL.PngImagePlugin.PngInfo method), 135
AFFINE (in module PIL.Image), 69
AffineTransform (class in PIL.ImageTransform), 133
alpha_composite () (in module PIL.Image), 51
alpha_composite () (PIL.Image.Image method), 55
ANTIALIAS (in module PIL.Image), 69
APNG_BLEND_OP_OVER (in module PIL.PngImagePlugin), 148
APNG_BLEND_OP_SOURCE (in module PIL.PngImagePlugin), 148
APNG_DISPOSE_OP_BACKGROUND (in module PIL.PngImagePlugin), 148
APNG_DISPOSE_OP_NONE (in module PIL.PngImagePlugin), 148
APNG_DISPOSE_OP_PREVIOUS (in module PIL.PngImagePlugin), 148
APP () (in module PIL.JpegImagePlugin), 143
AppendingTiffWriter (class in PIL.TiffImagePlugin), 150
applyTransform () (in module PIL.ImageCms), 73
arc () (PIL.ImageDraw.ImageDraw method), 88
arc () (PIL.ImageDraw2.Draw method), 132
attributes (PIL.ImageCms.CmsProfile attribute), 83
autocontrast () (in module PIL.ImageOps), 110

B
bdf_char () (in module PIL.BdfFontFile), 129
BdfFontFile (class in PIL.BdfFontFile), 129
begin_document () (PIL.PSDraw.PSDraw method), 123
bestsize () (PIL.IcnsImagePlugin.IcnsFile method), 140
BICUBIC (in module PIL.Image), 23
BILINEAR (in module PIL.Image), 23
BITFIELDS (PIL.BmpImagePlugin.BmpImageFile attribute), 136
bitmap (PIL.FontFile.FontFile attribute), 130
bitmap () (PIL.ImageDraw.ImageDraw method), 89
BitmapImage (class in PIL.ImageTk), 119
BitStream (class in PIL.MpegImagePlugin), 145
blend () (in module PIL.Image), 51
blend () (in module PIL.ImageChops), 70
blue_colorant (PIL.ImageCms.CmsProfile attribute), 83
blue_primary (PIL.ImageCms.CmsProfile attribute), 84
BmpImageFile (class in PIL.BmpImagePlugin), 136
BOX (in module PIL.Image), 22
BoxBlur (class in PIL.ImageFilter), 99
Brightness (class in PIL.ImageEnhance), 95
Brush (class in PIL.ImageDraw2), 131
BufrStubImageFile (class in PIL.BufrStubImagePlugin), 136
build_prototype_image () (in module PIL.PalmImagePlugin), 145
buildProofTransform () (in module PIL.ImageCms), 74
buildProofTransformFromOpenProfiles () (in module PIL.ImageCms), 75
buildTransform () (in module PIL.ImageCms), 76
buildTransformFromOpenProfiles () (in module PIL.ImageCms), 77

C
call () (PIL.PngImagePlugin.ChunkStream method), 146
check () (in module PIL.features), 126
check_codec () (in module PIL.features), 127
check_feature () (in module PIL.features), 128

211
Pillow (PIL Fork) Documentation, Release 8.0.0.dev0

decode() (PIL.MspImagePlugin.MspDecoder method), 145
decode() (PIL.GiGlImagePlugin.SGI16Decoder method), 149
deferred_error (class in PIL._util), 159
deform() (in module PIL.ImageOps), 112
denominator() (PIL.TiffImagePlugin.FRational property), 151
device_class (PIL.ImageCms.CmsProfile attribute), 82
dib (class in PIL.ImageWin), 120
DibImageFile (class in PIL.BmpImagePlugin), 136
difference() (in module PIL.ImageChops), 71
DQT() (in module PIL.JpegImagePlugin), 143
draft() (PIL.Image.Image method), 57
draft() (PIL.JpegImagePlugin.JpegImageFile method), 143
Draw (class in PIL.ImageDraw2), 131
Draw() (in module PIL.ImageDraw), 88
draw() (PIL.ImageWin.Dib method), 120
dump() (in module PIL.IptcImagePlugin), 143
duplicate() (in module PIL.ImageChops), 71

effect_mandelbrot() (in module PIL.Image), 53
effect_noise() (in module PIL.Image), 53
effect_spread() (PIL.Image.Image method), 57
ellipse() (PIL.ImageDraw.ImageDraw method), 89
ellipse() (PIL.ImageDraw2.Draw method), 132
end_document() (PIL.PSDraw.PSDraw method), 123
endian (PIL.Image.Exif attribute), 68
enhance() (PIL.ImageEnhance._Enhance method), 94
entropy() (PIL.Image.Image method), 57
EPSILON (in module PIL.GimpGradientFile), 131
EpsImageFile (class in PIL.EpsImagePlugin), 137
equalize() (in module PIL.ImageOps), 112
ERRORS (in module PIL.ImageFile), 97
eval() (in module PIL.Image), 51
eval() (in module PIL.ImageMath), 108
Exif (class in PIL.Image), 68
exif_transpose() (in module PIL.ImageOps), 114
expand() (in module PIL.ImageOps), 112
expose() (PIL.ImageWin.Dib method), 120
EXTENT (in module PIL.Image), 69
ExtentTransform (class in PIL.ImageTransform), 133
extrema (PIL.ImageStat.Stat attribute), 118

F
FASTOCTREE (in module PIL.Image), 70
feed() (PIL.ImageFile.Parser method), 96
field() (PIL.IptcImagePlugin.IptcImageFile method), 143
fieldSizes (PIL.TiffImagePlugin.AppendingTiffWriter attribute), 150
filename (PIL.Image.Image attribute), 67
Filter (class in PIL.ImageFilter), 100
filter() (PIL.Image.Image method), 57
filter() (PIL.ImageFilter.Filter method), 100
filter() (PIL.ImageFilter.MultibandFilter method), 100
finalize() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
fit() (in module PIL.ImageOps), 112
FITSTStubImageFile (class in PIL.FitsStubImagePlugin), 138
fixIFD() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
fixOffsets() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
FlfImageFile (class in PIL.FlfImagePlugin), 138
flip() (in module PIL.ImageOps), 113
FLIP_LEFT_RIGHT (in module PIL.Image), 68
FLIP_TOP_BOTTOM (in module PIL.Image), 68
floodfill() (in module PIL.ImageDraw), 94
FLOYDSTEINBERG (in module PIL.Image), 69
flush() (PIL.ImageDraw2.Draw method), 132
Font (class in PIL.ImageDraw2), 131
font_variant() (PIL.ImageFont.FreeTypoFont method), 103
FontFile (class in PIL.FontFile), 130
format (PIL.BmpImagePlugin.BmpImageFile attribute), 136
format (PIL.BmpImagePlugin.DibImageFile attribute), 136
format (PIL.BmpImagePlugin.BmpImageFile attribute), 136
format (PIL.GifImagePlugin.BfrStubImageFile attribute), 136
format (PIL.CurImagePlugin.CurImageFile attribute), 137
format (PIL.DcxImagePlugin.DcxImageFile attribute), 137
format (PIL.EpsImagePlugin.EpsImageFile attribute), 137
format (PIL.FitsStubImagePlugin.FITSStubImageFile attribute), 138
format (PIL.FilImagePlugin.FilImageFile attribute), 138
format (PIL.FpxImagePlugin.FpxImageFile attribute), 138
format (PIL.GbrImagePlugin.GbrImageFile attribute), 139
format (PIL.GdImageFile.GdImageFile attribute), 139
format (PIL.GifImagePlugin.GifImageFile attribute), 139
format (PIL.GribStubImagePlugin.GribStubImageFile attribute), 140
format (PIL.Hdf5StubImagePlugin.HDF5StubImageFile attribute), 140

Index 213
format_description (PIL.ImtImagePlugin.ImtImageFile attribute), 142
format_description (PIL.IptcImagePlugin.IptcImageFile attribute), 143
format_description (PIL.Jpeg2KImagePlugin.Jpeg2KImageFile attribute), 144
format_description (PIL.JpegImagePlugin.JpegImageFile attribute), 143
format_description (PIL.McIdasImagePlugin.McIdasImageFile attribute), 144
format_description (PIL.MicImagePlugin.MicImageFile attribute), 144
format_description (PIL.MpegImagePlugin.MpegImageFile attribute), 145
format_description (PIL.MspImagePlugin.MspImageFile attribute), 145
format_description (PIL.PcdImagePlugin.PcdImageFile attribute), 145
format_description (PIL.PcxImagePlugin.PcxImageFile attribute), 146
format_description (PIL.PixarImagePlugin.PixarImageFile attribute), 146
format_description (PIL.PngImagePlugin.PngImageFile attribute), 147
format_description (PIL.PpmImagePlugin.PpmImageFile attribute), 148
format_description (PIL.PsdImagePlugin.PsdImageFile attribute), 148
format_description (PIL.SgilImagePlugin.SgilImageFile attribute), 149
format_description (PIL.SpiderImagePlugin.SpiderImageFile attribute), 149
format_description (PIL.SunImagePlugin.SunImageFile attribute), 150
format_description (PIL.TgaImagePlugin.TgaImageFile attribute), 150
format_description (PIL.TiffImagePlugin.TiffImageFile attribute), 153
format_description (PIL.WebPImagePlugin.WebPImageFile attribute), 154
format_description (PIL.WmfImagePlugin.WmfStubImageFile attribute), 154
format_description (PIL.XbmImagePlugin.XbmImageFile attribute), 155
format_description (PIL.XpmImagePlugin.XpmImageFile attribute), 155
format_description (PIL.XVThumbImagePlugin.XVThumbImageFile attribute), 154
FpxImageFile (class in PIL.FpxImagePlugin), 138
frame() (PIL.IcolImagePlugin.IcoFile method), 141
FreeTypeFont (class in PIL.ImageFont), 103
from_v2() (PIL.TiffImagePlugin.ImageFileDirectory_v1 class method), 151
fromarray() (in module PIL.Image), 52
frombuffer() (in module PIL.Image), 53
frombytes() (in module PIL.Image), 52
frombytes() (PIL.ImageImage method), 57
frombytes() (PIL.ImageWin.Dib method), 120

G
GaussianBlur (class in PIL.ImageFilter), 99
GbrImageFile (class in PIL.GbrImagePlugin), 139
GdImageFile (class in PIL.GdImageFile), 130
generate() (PIL.ImageFilter.Color3DLUT class method), 99
get_command() (PIL.ImageShow.Viewer method), 117
get_display_profile() (in module PIL.ImageCms), 80
get_format() (PIL.ImageShow.Viewer method), 117
get_format_mimetype() (PIL.ImageFile.ImageFile method), 97
get_ifd() (PIL.Image.Exif method), 68
get_interlace() (in module PIL.GifImagePlugin), 139
get_sampling() (in module PIL.JpegImagePlugin), 144
get_supported() (in module PIL.features), 127
get_supported_codecs() (in module PIL.features), 128
get_supported_features() (in module PIL.features), 128
get_supported_modules() (in module PIL.features), 127
Pillow (PIL Fork) Documentation, Release 8.0.0.dev0

getvariation_axes()
    (PIL.ImageFont.FreeTypeFont method), 103
getvariation_names()
    (PIL.ImageFont.FreeTypeFont method), 103
getbands() (PIL.Image method), 58
getbbox() (PIL.Image method), 58
getbbox() (PIL.ImagePath.Path method), 115
getchannel() (PIL.Image method), 58
getchunks() (PIL.PngImagePlugin), 147
getcolor() (PIL.ImageColor), 86
getcolor() (PIL.ImagePalette.ImagePalette method), 114
getcolors() (PIL.Image method), 58
getdata() (PIL.GifImagePlugin), 139
getdata() (PIL.Image method), 58
getdata() (PIL.ImagePalette.ImagePalette method), 114
getdata() (PIL.ImageTransform.Transform method), 134
getDefaultIntent() (in module PIL.ImageCms), 78
getafter() (in module PIL.ImageDraw), 93
getentryindex() (PIL.ImagePlugin.IcoFile method), 141
getextrema() (PIL.Image method), 59
getextrema() (PIL.PngImagePlugin.PngImageFile method), 146
gextrema() (PIL.Image.Image method), 59
gextrema() (PIL.Image.Image method), 59
getfont() (PIL.ImageDraw.ImageDraw method), 88
getheader() (in module PIL.GifImagePlugin), 140
getimage() (PIL.IcnsImagePlugin.IcnsFile method), 140
getimage() (PIL.IcnsImagePlugin.IcnsFile method), 141
gimage() (PIL.PngImagePlugin.PngImageFile method), 143
gimptcinfo() (in module PIL.PngImagePlugin), 143
getmask() (PIL.ImageFont.FreeTypeFont method), 103
getmask() (PIL.ImageFont.ImageFont method), 103
getmask2() (PIL.ImageFont.FreeTypeFont method), 104
getmetrics() (PIL.ImageFont.FreeTypeFont method), 105
getname() (PIL.ImageFont.FreeTypeFont method), 105
getoffset() (PIL.ImageFont.FreeTypeFont method), 105
getopenprofile() (in module PIL.ImageCms), 79
getpalette() (PIL.GimpGradientFile.GradientFile method), 131
getpalette() (PIL.GimpPaletteFile.GimpPaletteFile method), 131
getpalette() (PIL.Image.Image method), 59
getpalette() (PIL.PaletteFile.PaletteFile method), 134
getpixel() (PIL.Image.Image method), 59
getpixel() (PIL.PyAccess.PyAccess method), 126
gprofilecopyright() (in module PIL.ImageCms), 79
gprofiledescription() (in module PIL.ImageCms), 79
gprofileinfo() (in module PIL.ImageCms), 79
getprofilemaker() (in module PIL.ImageCms), 80
getprofilemodel() (in module PIL.ImageCms), 80
gprofilename() (in module PIL.ImageCms), 80
gprojection() (PIL.Image.Image method), 59
getrgb() (PIL.ImageColor), 86
getsize() (PIL.ImageFont.FreeTypeFont method), 105
getsize() (PIL.ImageFont.ImageFont method), 103
getsize_multiline() (PIL.ImageFont.FreeTypeFont method), 105
ghostscript() (in module PIL.EpsImagePlugin), 137
gifimagefile (class in PIL.GifImagePlugin), 139
GimpGradientFile (class in PIL.GimpGradientFile), 131
GimpPaletteFile (class in PIL.GimpPaletteFile), 131
global_palette (PIL.GifImagePlugin.GifImageFile attribute), 139
goToend() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
GPSTAGS (in module PIL.ExifTags), 121
grab() (in module PIL.ImageGrab), 107
grabclipboard() (in module PIL.ImageGrab), 107
ggradient (PIL.GimpGradientFile.GradientFile attribute), 131
GradientFile (class in PIL.GimpGradientFile), 131
grayscale() (in module PIL.ImageOps), 113
green_colorant (PIL.ImageCms.CmsProfile attribute), 83
green_primary (PIL.ImageCms.CmsProfile attribute), 84
GribStubImageFile (class in PIL.GribStubImagePlugin), 140

H
HAMMING (in module PIL.Image), 23
hardlight() (in module PIL.ImageChops), 72
has_ghostscript() (in module PIL.EpsImagePlugin), 138
HDC (class in PIL.ImageWin), 121
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF5StubImageFile</td>
<td>PIL.Hdf5StubImagePlugin</td>
<td>140</td>
</tr>
<tr>
<td>header_flags</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>82</td>
</tr>
<tr>
<td>header_manufacturer</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>82</td>
</tr>
<tr>
<td>header_model</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>82</td>
</tr>
<tr>
<td>height</td>
<td>PIL.Image</td>
<td>67</td>
</tr>
<tr>
<td>height()</td>
<td>PIL.ImageTk.BitmapImage</td>
<td>119</td>
</tr>
<tr>
<td>height()</td>
<td>PIL.ImageTk.PhotofileImage</td>
<td>119</td>
</tr>
<tr>
<td>histogram()</td>
<td>PIL.Image</td>
<td>59</td>
</tr>
<tr>
<td>HWND</td>
<td>PIL.ImageWin</td>
<td>121</td>
</tr>
<tr>
<td>i()</td>
<td>PIL.Image</td>
<td>143</td>
</tr>
<tr>
<td>i16be()</td>
<td>PIL._binary</td>
<td>158</td>
</tr>
<tr>
<td>i16le()</td>
<td>PIL._binary</td>
<td>158</td>
</tr>
<tr>
<td>i32be()</td>
<td>PIL._binary</td>
<td>158</td>
</tr>
<tr>
<td>i32le()</td>
<td>PIL._binary</td>
<td>158</td>
</tr>
<tr>
<td>i8()</td>
<td>PIL._binary</td>
<td>158</td>
</tr>
<tr>
<td>icc_version</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>82</td>
</tr>
<tr>
<td>IcnsFile</td>
<td>PIL.IcnsImagePlugin</td>
<td>140</td>
</tr>
<tr>
<td>IcnsImageFile</td>
<td>PIL.IcnsImagePlugin</td>
<td>140</td>
</tr>
<tr>
<td>IcoFile</td>
<td>PIL.IcoImagePlugin</td>
<td>141</td>
</tr>
<tr>
<td>IcoImageFile</td>
<td>PIL.IcoImagePlugin</td>
<td>141</td>
</tr>
<tr>
<td>IFDRational</td>
<td>PIL.TiffImagePlugin</td>
<td>151</td>
</tr>
<tr>
<td>Image</td>
<td>PIL.Image</td>
<td>55</td>
</tr>
<tr>
<td>image()</td>
<td>PIL.PSDraw.PSDraw</td>
<td>123</td>
</tr>
<tr>
<td>ImageCmsTransform</td>
<td>PIL.ImageCms</td>
<td>73</td>
</tr>
<tr>
<td>ImageFile</td>
<td>PIL.ImageFile</td>
<td>97</td>
</tr>
<tr>
<td>ImageFileDirectory</td>
<td>PIL.TiffImagePlugin</td>
<td>151</td>
</tr>
<tr>
<td>ImageFileDirectory_v1</td>
<td>PIL.TiffImagePlugin</td>
<td>151</td>
</tr>
<tr>
<td>ImageFileDirectory_v2</td>
<td>PIL.TiffImagePlugin</td>
<td>152</td>
</tr>
<tr>
<td>ImageFont</td>
<td>PIL.ImageFont</td>
<td>103</td>
</tr>
<tr>
<td>ImagePalette</td>
<td>PIL.ImagePalette</td>
<td>114</td>
</tr>
<tr>
<td>ImagePointHandler</td>
<td>PIL.Image</td>
<td>68</td>
</tr>
<tr>
<td>ImageQt</td>
<td>PIL.Image</td>
<td>116</td>
</tr>
<tr>
<td>ImageTransformHandler</td>
<td>PIL.Image</td>
<td>68</td>
</tr>
<tr>
<td>Im1ImageFile</td>
<td>PIL.ImlImagePlugin</td>
<td>142</td>
</tr>
<tr>
<td>ImtImageFile</td>
<td>PIL.ImtImagePlugin</td>
<td>142</td>
</tr>
<tr>
<td>info</td>
<td>PIL.Image</td>
<td>67</td>
</tr>
<tr>
<td>init()</td>
<td>PIL.ImageFile.PyDecoder</td>
<td>96</td>
</tr>
<tr>
<td>intent_supported</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>85</td>
</tr>
<tr>
<td>invert()</td>
<td>PIL.ImageChops</td>
<td>71</td>
</tr>
<tr>
<td>invert()</td>
<td>PIL.ImageOps</td>
<td>113</td>
</tr>
<tr>
<td>IptcImageFile</td>
<td>PIL.IptcImagePlugin</td>
<td>143</td>
</tr>
<tr>
<td>is_animated()</td>
<td>PIL.GifImagePlugin</td>
<td>139</td>
</tr>
<tr>
<td>is_animated()</td>
<td>PIL.ImlImagePlugin</td>
<td>142</td>
</tr>
<tr>
<td>is_animated()</td>
<td>PIL.SpiderImagePlugin</td>
<td>149</td>
</tr>
<tr>
<td>is_cid()</td>
<td>PIL.PngImagePlugin</td>
<td>147</td>
</tr>
<tr>
<td>is_intent_supported()</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>85</td>
</tr>
<tr>
<td>is_matrix_shaper</td>
<td>PIL.ImageCms.CmsProfile</td>
<td>84</td>
</tr>
<tr>
<td>isatty()</td>
<td>PIL.ContainerIO.ContainerIO</td>
<td>129</td>
</tr>
<tr>
<td>isDirectory()</td>
<td>PIL._util</td>
<td>159</td>
</tr>
<tr>
<td>isInt()</td>
<td>PIL.SpiderImagePlugin</td>
<td>150</td>
</tr>
<tr>
<td>isIntentSupported()</td>
<td>PIL.ImageCms</td>
<td>80</td>
</tr>
<tr>
<td>isPath()</td>
<td>PIL._util</td>
<td>159</td>
</tr>
<tr>
<td>isSpiderHeader()</td>
<td>PIL.SpiderImagePlugin</td>
<td>150</td>
</tr>
<tr>
<td>isSpiderImage()</td>
<td>PIL.SpiderImagePlugin</td>
<td>150</td>
</tr>
<tr>
<td>Iterator</td>
<td>PIL.ImageSequence</td>
<td>116</td>
</tr>
<tr>
<td>itersizes()</td>
<td>PIL.IcnsImagePlugin.IcnsFile</td>
<td>140</td>
</tr>
<tr>
<td>iTxt</td>
<td>PIL.PngImagePlugin</td>
<td>134</td>
</tr>
<tr>
<td>JPEG</td>
<td>PIL.BmpImagePlugin</td>
<td>136</td>
</tr>
<tr>
<td>Jpeg2KImageFile</td>
<td>PIL.Jpeg2KImagePlugin</td>
<td>144</td>
</tr>
<tr>
<td>JpegImageFile</td>
<td>PIL.JpegImagePlugin</td>
<td>143</td>
</tr>
<tr>
<td>k</td>
<td>PIL.BmpImagePlugin</td>
<td>136</td>
</tr>
<tr>
<td>Kernel</td>
<td>PIL.ImageFilter</td>
<td>99</td>
</tr>
<tr>
<td>LANCZOS</td>
<td>PIL.Image</td>
<td>23</td>
</tr>
<tr>
<td>legacy_api()</td>
<td>PIL.TiffImagePlugin.ImageFileDirectory_v2</td>
<td>152</td>
</tr>
<tr>
<td>LIBIMAGEQUANT</td>
<td>PIL.Image</td>
<td>70</td>
</tr>
<tr>
<td>lighter()</td>
<td>PIL.ImageChops</td>
<td>71</td>
</tr>
<tr>
<td>limit_rational()</td>
<td>PIL.TiffImagePlugin.IFDRational</td>
<td>151</td>
</tr>
<tr>
<td>line()</td>
<td>PIL.ImageDraw.ImageDraw</td>
<td>89</td>
</tr>
<tr>
<td>line()</td>
<td>PIL.ImageDraw.ImageDraw</td>
<td>132</td>
</tr>
<tr>
<td>line()</td>
<td>PIL.PSDraw.PSDraw</td>
<td>123</td>
</tr>
<tr>
<td>LINEAR</td>
<td>PIL.Image</td>
<td>69</td>
</tr>
<tr>
<td>linear()</td>
<td>PIL.GimpGradientFile</td>
<td>131</td>
</tr>
<tr>
<td>linear_gradient()</td>
<td>PIL.Image</td>
<td>53</td>
</tr>
</tbody>
</table>
PIL.TarIO, 135
PIL.TgaImagePlugin, 150
PIL.TiffImagePlugin, 150
PIL.TiffTags, 121
PIL.WalImagePlugin, 135
PIL.WebPImagePlugin, 154
PIL.WmfImagePlugin, 154
PIL.XbmImagePlugin, 155
PIL.XpmImagePlugin, 155
PIL.XVThumbImagePlugin, 154
MpegImageFile (class in PIL.MpegImagePlugin), 145
MspDecoder (class in PIL.MspImagePlugin), 145
MspImageFile (class in PIL.MspImagePlugin), 145
MultibandFilter (class in PIL.ImageFilter), 100
multiline_text() (PIL.ImageDraw.ImageDraw method), 92
multiline_textsize() (PIL.ImageDraw.ImageDraw method), 93
multiply() (in module PIL.ImageChops), 72

N
n_frames (PIL.Image.Image attribute), 67
n_frames() (PIL.GifImagePlugin.GifImageFile property), 139
n_frames() (PIL.ImImagePlugin.ImImageFile property), 142
n_frames() (PIL.SpiderImagePlugin.SpiderImageFile property), 149
n_frames() (PIL.TiffImagePlugin.TiffImageFile property), 153
name (PIL.PcfFontFile.PcfFontFile attribute), 134
named() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 152
NEAREST (in module PIL.Image), 22
new() (in module PIL.Image), 52
newFrame() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
next() (PIL.MpegImagePlugin.BitStream method), 145
nextHeader() (in module PIL.IcnsImagePlugin), 141
NONE (in module PIL.Image), 68
number() (in module PIL.ImImagePlugin), 142
numerator() (PIL.TiffImagePlugin.IFDRational property), 151

O
o16be() (in module PIL._binary), 158
o16le() (in module PIL._binary), 158
o32be() (in module PIL._binary), 158
o32le() (in module PIL._binary), 158
o8 () (in module PIL._binary), 158
offset () (in module PIL.ImageChops), 72
offset () (PIL.TiffImagePlugin.ImageFileDirectory_v2 property), 152
open() (in module PIL.GdImageFile), 130
open() (in module PIL.WallImageFile), 135
options (PIL.ImageShow.Viewer attribute), 117
overlay() (in module PIL.ImageChops), 72

P
pad() (in module PIL.ImageOps), 111
palette (PIL.Image.Image attribute), 67
PaletteFile (class in PIL.PaletteFile), 134
Parser (class in PIL.ImageFile), 96
paste() (PIL.Image.Image method), 59
paste() (PIL.ImageTk.PhotoImage method), 119
paste() (PIL.PngImagePlugin.PngImageFile method), 146
PcdImageFile (class in PIL.PcdImagePlugin), 145
PcfFontFile (class in PIL.PcfFontFile), 134
PcxImageFile (class in PIL.PcxImagePlugin), 146
peek() (PIL.MpegImagePlugin.BitStream method), 145
Pen (class in PIL.ImageDraw2), 131
perceptual_rendering_intent_gamut (PIL.ImageCms.CmsProfile attribute), 84
PERSPECTIVE (in module PIL.Image), 69
PhotoImage (class in PIL.ImageTk), 119
pieslice() (PIL.ImageDraw.ImageDraw method), 90
pieslice() (PIL.ImageDraw2.Draw method), 132
PIL
    module, 129
    _binary module, 158
    _imaging module, 159
    _tkinter_finder module, 159
    _util module, 159
    _version module, 159
    BdfFontFile module, 129
    BmpImagePlugin module, 136
    BufrStubImagePlugin module, 136
    ContainerIO module, 129
    CurImagePlugin module, 137
    DcxImagePlugin module, 137
    EpsImagePlugin module, 137
    ExifTags module, 121
    features module, 126

220 Index
PIL.FitsStubImagePlugin module, 138
PIL.FliImagePlugin module, 138
PIL.FontFile module, 130
PIL.FpxImagePlugin module, 138
PIL.GbrImagePlugin module, 139
PIL.GdImageFile module, 130
PIL.GifImagePlugin module, 139
PIL.GimpGradientFile module, 131
PIL.GimpPaletteFile module, 131
PIL.GribStubImagePlugin module, 140
PIL.Hdf5StubImagePlugin module, 140
PIL.IcnsImagePlugin module, 140
PIL.IcoImagePlugin module, 141
PIL.Image module, 49
PIL.Image.core module, 159
PIL.ImageChops module, 70
PIL.ImageCms module, 73
PIL.ImageColor module, 85
PIL.ImageDraw module, 86
PIL.ImageDraw2 module, 131
PIL.ImageEnhance module, 94
PIL.ImageFile module, 95
PIL.ImageFilter module, 97
PIL.ImageFont module, 100
PIL.ImageFont.LAYOUT_BASIC module
PIL.ImageFont.LAYOUT_RAQM module
PIL.ImageGrab module, 106
PIL.ImageMath module, 107
PIL.ImageMorph module, 109
PIL.ImageOps module, 110
PIL.ImagePalette module, 114
PIL.ImagePath module, 115
PIL.ImagePath.Path (class in PILImagePath), 115
PIL.ImageQt module, 116
PIL.ImageSequence module, 116
PIL.ImageShow module, 116
PIL.ImageStat module, 118
PIL.ImageTk module, 118
PIL.ImageTransform module, 133
PIL.ImageWin module, 119
PIL.ImImagePlugin module, 142
PIL.ImtImagePlugin module, 142
PIL.IptcImagePlugin module, 143
PIL.Jpeg2KImagePlugin module, 144
PIL.JpegImagePlugin module, 143
PIL.JpegPresets module, 122
PIL.McIdasImagePlugin module, 144
PIL.MicImagePlugin module, 144
PIL.MpegImagePlugin module, 145
PIL.MspImagePlugin module, 145
PIL.PaletteFile module, 134
PIL.PalmImagePlugin module, 145
PIL.PcdImagePlugin module, 145
PIL.PcffFontFile module, 134
PIL.PcxImagePlugin module,
module, 146
PIL.PdfImagePlugin module, 146
PIL.PixarImagePlugin module, 146
PIL.PngImagePlugin module, 146
PIL.PpmImagePlugin module, 148
PIL.PsdImagePlugin module, 148
PIL.PSDraw module, 123
PIL.PyAccess module, 125
PIL.SgiImagePlugin module, 149
PIL.SpiderImagePlugin module, 149
PIL.SunImagePlugin module, 150
PIL.TarIO module, 135
PIL.TgaImagePlugin module, 150
PIL.TiffImagePlugin module, 150
PIL.TiffTags module, 121
PIL.TiffTags.LIBTIFF_CORE (in module PIL.TiffTags), 122
PIL.TiffTags.TAGS (in module PIL.TiffTags), 122
PIL.TiffTags.TAGS_V2 (in module PIL.TiffTags), 122
PIL.TiffTags.TYPES (in module PIL.TiffTags), 122
PIL.WallImageFile module, 135
PIL.WebPImagePlugin module, 154
PIL.WmfImagePlugin module, 154
PIL.XbmImagePlugin module, 155
PIL.XpmImagePlugin module, 155
PIL.XVThumbImagePlugin module, 154
piinfo() (in module PIL.features), 126
PixarImageFile (class in PIL.PixarImagePlugin), 146
PixelAccess (built-in class), 125
PNG (PIL.BmpImagePlugin.BmpImageFile attribute), 136
PngImageFile (class in PIL.PngImagePlugin), 146
PngInfo (class in PIL.PngImagePlugin), 135
PngStream (class in PIL.PngImagePlugin), 147
point() (PIL.Image.Image method), 60
point() (PIL.ImageDraw.ImageDraw method), 90
polygon() (PIL.ImageDraw.ImageDraw method), 90
polygon() (PIL.ImageDraw2.Draw method), 132
posterior() (in module PIL.ImageOps), 113
PpmImageFile (class in PIL.PpmImagePlugin), 148
prefix() (PIL.TiffImagePlugin.ImageFileDirectory_v2 property), 153
presets (in module PIL.JpegPresets), 123
profile_description (PIL.ImageCms.CmsProfile attribute), 83
profile_id (PIL.ImageCms.CmsProfile attribute), 83
profileToProfile() (in module PIL.ImageCms), 81
PsdImageFile (class in PIL.PsdImagePlugin), 148
PSDraw (class in PIL.PSDraw), 123
PSFile (class in PIL.EpsImagePlugin), 137
push() (PIL.PngImagePlugin.ChunkStream method), 146
putalpha() (PIL.Image.Image method), 60
putchunk() (in module PIL.PngImagePlugin), 148
putdata() (PIL.Image.Image method), 60
puti16() (in module PIL.FontFile), 130
putpalette() (PIL.Image.Image method), 61
putpixel() (PIL.PyAccess.PyAccess method), 126
PyAccess (class in PIL.PyAccess), 126
PyCMSError, 73
PyDecoder (class in PIL.ImageFile), 96
Q
QUAD (in module PIL.Image), 69
QuadTransform (class in PIL.ImageTransform), 133
quantize() (PIL.Image.Image method), 61
query_palette() (PIL.ImageWin.Dib method), 120
R
radial_gradient() (in module PIL.Image), 54
RankFilter (class in PIL.ImageFilter), 100
RAW (PIL.BmpImagePlugin.BmpImageFile attribute), 136
rawmode (PIL.GimpPaletteFile.GimpPaletteFile attribute), 131
read() (PIL.ContainerIO.ContainerIO attribute), 134
read() (PIL.MpegImagePlugin.BitStream method), 145
read() (PIL.PngImagePlugin.ChunkStream method), 146
read_32() (in module PIL.IcnsImagePlugin), 141
read_32t() (in module PIL.IcnsImagePlugin), 141
read_mk() (in module PIL.IcnsImagePlugin), 141
read_png_or_jpeg2000() (in module PIL.ImagePlugin), 141
readline() (PIL.ContainerIO.ContainerIO method), 129
readline() (PIL.EpsImagePlugin.PSFile method), 137
readlines() (PIL.ContainerIO.ContainerIO method), 129
readLong() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
readShort() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
rectangle() (PIL.ImageDraw.ImageDraw method), 91
rectangle() (PIL.ImageDraw2.Draw method), 132
red_colorant (PIL.ImageCms.CmsProfile attribute), 83
red_primary (PIL.ImageCms.CmsProfile attribute), 84
reduce() (PIL.Image.Image method), 61
reduce() (PIL.Jpeg2KImagePlugin.Jpeg2KImageFile property), 144
register() (in module PIL.ImageShow), 117
register_decoder() (in module PIL.Image), 55
register_encoder() (in module PIL.Image), 55
register_extension() (in module PIL.Image), 54
register_extensions() (in module PIL.Image), 54
register_handler() (in module PIL.BufStubImagePlugin), 136
register_handler() (in module PIL.FitsStubImagePlugin), 138
register_handler() (in module PIL.GribStubImagePlugin), 140
register_handler() (in module PIL.Hdf5StubImagePlugin), 140
register_handler() (in module PIL.WmfImagePlugin), 154
register_mime() (in module PIL.Image), 54
register_open() (in module PIL.Image), 54
register_save() (in module PIL.Image), 54
register_save_all() (in module PIL.Image), 54
registered_extensions() (in module PIL.Image), 55
regular_polygon() (PIL.ImageDraw.ImageDraw method), 90
remap_palette() (PIL.Image.Image method), 62
render() (PIL.ImageDraw2.Draw method), 132
rendering_intent (PIL.ImageCms.CmsProfile attribute), 83
reset() (PIL.ImageFile.Parser method), 96
reset() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
resize() (PIL.Image.Image method), 62
rewind() (PIL.PngImagePlugin.PngStream method), 147
rewriteLastLong() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
rewriteLastShort() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
rewriteLastShortToLong() (PIL.TiffImagePlugin.AppendingTiffWriter method), 150
RLE4 (PIL.BmpImagePlugin.BmpImageFile attribute), 136
RLE8 (PIL.BmpImagePlugin.BmpImageFile attribute), 136
rms (PIL.ImageStat.Stat attribute), 118
rotate() (PIL.Image.Image method), 62
ROTATE_180 (in module PIL.Image), 68
ROTATE_270 (in module PIL.Image), 68
ROTATE_90 (in module PIL.Image), 68
S
saturation_rendering_intent_gamut (PIL.ImageCms.CmsProfile attribute), 84
save() (PIL.FontFile.FontFile method), 130
save() (PIL.Image.Image method), 63
save() (PIL.ImagePalette.ImagePalette method), 114
save() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
save_image() (PIL.ImageShow.Viewer method), 117
save_rewind() (PIL.PngImagePlugin.PngStream method), 147
scale() (in module PIL.ImageOps), 112
screen() (in module PIL.ImageChops), 72
screening_description (PIL.ImageCms.CmsProfile attribute), 84
seek() (PIL.ContainerIO.ContainerIO method), 129
seek() (PIL.DcxImagePlugin.DcxImageFile method), 137
seek() (PIL.EpsImagePlugin.PSFile method), 138
seek() (PIL.FilImagePlugin.FilImageFile method), 138
seek() (PIL.GifImagePlugin.GifImageFile method), 139
seek() (PIL.Image.Image method), 63
seek() (PIL.ImImagePlugin.ImImageFile method), 142
seek() (PIL.MicImagePlugin.MicImageFile method), 144
seek() (PIL.PngImagePlugin.PngImageFile method), 146
seek() (PIL.PsdImagePlugin.PsdImageFile method), 148
tell() (PIL.WebPImagePlugin.WebPImageFile method), 154
text() (PIL.ImageDraw.ImageDraw method), 91
text() (PIL.ImageDraw2.Draw method), 132
text() (PIL.PngImagePlugin.PngImageFile property), 147
textsize() (PIL.ImageDraw.ImageDraw method), 92
textsize() (PIL.ImageDraw2.Draw method), 133
tgaImageFile (class in PIL.TgaImagePlugin), 150
thumbnail() (PIL.Image.Image method), 64
TiffImageFile (class in PIL.TiffImagePlugin), 153
tile (PIL.ImageFile.ImageFile attribute), 97
tkPhotoImage() (PIL.SpiderImagePlugin.SpiderImageFile method), 149
to_v2() (PIL.TiffImagePlugin.ImageFileDirectory_v1 method), 151
tobitmap() (PIL.Image.Image method), 65
tobytes() (PIL.Image.Exif method), 68
tobytes() (PIL.Image.Image method), 65
tobytes() (PIL.ImagePalette.ImagePalette method), 115
tobytes() (PIL.ImageWin.Dib method), 121
tobytes() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
tolist() (PILImagePath.PIL.ImagePath.Path method), 115
tostring() (PIL.ImagePalette.ImagePalette method), 115
Transform (class in PIL.ImageTransform), 134
transform() (PIL.Image.Image method), 65
transform() (PIL.ImageFilter.Color3DLUT method), 99
transform() (PILImagePath.PIL.ImagePath.Path method), 116
transform() (PIL.ImageTransform.Transform method), 134
TRANSPOSE (in module PIL.Image), 68
transpose() (PIL.Image.Image method), 66
TransposedFont (class in PIL.ImageFont), 106
TRANSVERSE (in module PIL.Image), 68
truetype() (in module PIL.ImageFont), 101

U
UnidentifiedImageError, 129
UnixViewer (class in PIL.ImageShow), 117
UnixViewer.DisplayViewer (class in PIL.ImageShow), 117
UnixViewer.EogViewer (class in PIL.ImageShow), 117
UnixViewer.XVViewer (class in PIL.ImageShow), 117
UnsharpMask (class in PIL.ImageFilter), 99

V
v (PIL.BmpImagePlugin.BmpImageFile attribute), 136
var (PIL.ImageStat.Stat attribute), 118
verify() (PIL.Image.Image method), 66
verify() (PIL.ImageFile.ImageFile method), 97
verify() (PIL.PngImagePlugin.ChunkStream method), 146
verify() (PIL.PngImagePlugin.PngImageFile method), 147
version (PIL.ImageCms.CmsProfile attribute), 82
version() (in module PIL.PILimage), 127
version_codec() (in module PIL.PILimage), 128
version_feature() (in module PIL.PILimage), 128
version_module() (in module PIL.PILimage), 127
versions() (in module PIL.PILimage), 82
Viewer (class in PIL.ImageShow), 117
viewing_condition (PIL.ImageCms.CmsProfile attribute), 84

W
WEB (in module PIL.Image), 70
WebPImageFile (class in PIL.WebPImagePlugin), 154
width (PIL.Image.Image attribute), 67
width() (PIL.ImageTk.BitmapImage method), 119
width() (PIL.ImageTk.PhotoImage method), 119
WindowsViewer (class in PIL.ImageShow), 117
WmfStubImageFile (class in PIL.WmfImagePlugin), 154
write() (PIL.TiffImagePlugin.AppendingTiffWriter method), 151
write_byte() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_double() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_float() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_long() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_rational() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_short() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_signed_byte() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_signed_long() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_signed_rational() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_signed_short() (PIL.TiffImagePlugin.ImageFileDirectory_v2 method), 153
write_string() (PIL.TiffImagePlugin.ImageFileDirectory_v2
    method), 153
write_undefined()
    (PIL.TiffImagePlugin.ImageFileDirectory_v2
    method), 153
writeLong() (PIL.TiffImagePlugin.AppendingTiffWriter
    method), 151
writeShort() (PIL.TiffImagePlugin.AppendingTiffWriter
    method), 151

X
XbmImageFile (class in PIL.XbmImagePlugin), 155
xcolor_space (PIL.ImageCms.CmsProfile attribute), 82
XpmImageFile (class in PIL.XpmImagePlugin), 155
XVThumbImageFile (class in PIL.XVThumbImagePlugin), 154