PIL Documentation

Release 1.1.7

effbot

May 17, 2016

Contents

CHAPTER 1

Introduction

1.1 Overview

1.1.1 Introduction

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:

1.1.2 Image Archives

The Python Imaging Library is ideal for 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.

1.1.3 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.

1.1.4 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 Tutorial

1.2.1 Using the Image Class

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

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

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

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

```
>>> print im.format, im.size, im.mode
PPM (512, 512) RGB
```

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

If the file cannot be opened, an **IOError** exception is raised.

Once you have an instance of the **Image** class, you can use the methods defined by this class to process and manipulate the image. For example, let's display the image we just loaded:

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

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

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

1.2.2 Reading and Writing Images

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

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

Convert files to JPEG

```
import os, sys
import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print "cannot convert", infile
```

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

Create JPEG Thumbnails

```
import os, sys
import Image

size = 128, 128

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

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

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

Identify Image Files

```
import sys
import Image

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