Native Imaging Documentation

Release 0.0.1

Chris Adams, Dan Krech, Ed Summers

Nov 07, 2017

Contents

This is an experiment in seeing how far you can get using platform-provided packages such as GraphicsMagick, CoreImage, etc. to provide a PIL-like interface but taking advantage of their support for more advanced features such as threading, broader format support (including JPEG-2000), vectorization, etc.

The goal is simple: a user should be able to install NativeImaging and do something like this to a program which is currently using PIL:

```
from NativeImaging import get_image_class

Image = get_image_class("GraphicsMagick")
```

CHAPTER 1

Status

1.1 aware

Very fast JPEG 2000 thumbnail generation compared to GraphicsMagick. Requires the non-OSS AWARE library: http://www.aware.com/imaging/jpeg2000.htm

1.2 GraphicsMagick

Currently supports typical web application usage: loading an image, resizing it and saving the result. Testing reveals mixed results, beating PIL when producing thumbnails from large TIFFs and underperforming when thumbnailing equivalent JPEGs, both by about 2:1.

Both CPython and PyPy are supported, with PyPy seeing performance gains using the CFFI backend instead of ctypes. Significant optimization gains are likely possible, particularly where the I/O functions marshall data in and out of the non-filename-based APIs where data is currently being copied.

1.3 Jython

Currently supports basic usage: loading an image, resizing it, and saving the result. Performance is generally quite decent as the Java Advanced Imaging API is quite tuned, if somewhat baroque in design.

CHAPTER 2

Table of Contents

2.1 Common API

class NativeImaging.api.**Image**
Base class for all NativeImaging backends

Should be compatible with PIL.Image or raise NotImplementedError()

convert(*mode=None*, *data=None*, *dither=None*, *palette=0*, *colors=256*)
Convert to other pixel format

copy()
Returns an exact copy of the current image which may be destructively modified without affecting the original. Backends may choose to implement Copy-On-Write for performance so callers should not expect resource handles or object ids to change simply by calling copy().

crop(*box=None*)
Return a cropped version of the image

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

Return type :class:Image object

draft(*mode*, *size*)
Configures the image file loader so it returns a version of the image that as closely as possible matches the given mode and size. For example, you can use this method to convert a colour JPEG to greyscale while loading it, or to extract a 128x192 version from a PCD file.

filter(*filter*)
Apply environment filter to image

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

Parameters **filter** – Filter kernel.

Return type :class:Image object

format = None

format_description = None

fromstring(*data*, *decoder_name='raw'*, **args*)
Load data to image from binary string

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.

getbbox()
Get bounding box of actual data (non-zero pixels) in image

Calculates the bounding box of the non-zero regions in the image.

Returns The bounding box is returned as a 4-tuple defining the left, upper, right, and lower pixel coordinate. If the image is completely empty, this method returns None.

getcolors(*maxcolors=256*)
Get colors from image, up to given limit 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.

getdata(*band=None*)
Get image data as sequence object

getextrema()
Get min/max value

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.

getpalette()
Get palette contents

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

getpixel(*xy*)
Get pixel value

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

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

getprojection()
Get projection to x and y axes

Returns the horizontal and vertical projection.

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

histogram(*mask=None*, *extrema=None*)
Take histogram of image

Returns a histogram for the image. The histogram is returned as a list of pixel counts, one for each pixel value in the source image. If the image has more than one band, the histograms for all bands are concatenated (for example, the histogram for an "RGB" image contains 768 values).

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

If a mask is provided, the method returns a histogram for those parts of the image where the mask image is non-zero. The mask image must have the same size as the image, and be either a bi-level image (mode "1") or a greyscale image ("L").

Parameters **mask** – An optional mask.

Returns A list containing pixel counts.

im = None

info = {}

load()
Explicitly load pixel data.

mode = ''

classmethod open(*fp*, *mode='r'*)

palette = None

paste(*im*, *box=None*, *mask=None*)
Paste other image into region

Pastes another image into this image. The box argument is either a 2-tuple giving the upper left corner, a 4-tuple defining the left, upper, right, and lower pixel coordinate, or None (same as (0, 0)). If a 4-tuple is given, the size of the pasted image must match the size of the region.

If the modes don't match, the pasted image is converted to the mode of this image (see the *Image.convert()* method for details).

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

If a mask is given, this method updates only the regions indicated by the mask. You can use either "1", "L" or "RGBA" images (in the latter case, the alpha band is used as mask). Where the mask is 255, the given image is copied as is. Where the mask is 0, the current value is preserved. Intermediate values can be used for transparency effects.

Note that if you paste an "RGBA" image, the alpha band is ignored. You can work around this by using the same image as both source image and mask.

Parameters

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

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

Return type :class:Image object

point(*lut*, *mode=None*)
Maps this image through a lookup table or function.

Parameters

- **lut** – A lookup table, containing 256 values per band in the image. A function can be used instead, it should take a single argument. The function is called once for each possible pixel value, and the resulting table is applied to all bands of the image.
- **mode** – Output mode (default is same as input). In the current version, this can only be used if the source image has mode "L" or "P", and the output has mode "1".

Return type :class:Image object

putalpha(*alpha*)
Set alpha layer 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 **im** – 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.

putdata(*data*, *scale=1.0*, *offset=0.0*)
Put data from a sequence object into an image

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.

putpalette(*data*, *rawmode='RGB'*)
Put palette data into an image.

putpixel(*xy*, *value*)
Modifies the pixel at the given position. The colour is given as a single numerical value for single-band images, and a tuple for multi-band images.

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

Parameters

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

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

readonly = 0

resize(*size*, *resample=0*)
Returns a resized copy of this image.

Parameters

- **size** (*tuple*) – The requested size in pixels, as a 2-tuple: (width, height).
- **filter** – An optional resampling filter. This can be one of NEAREST (use nearest neighbour), BILINEAR (linear interpolation in a 2x2 environment), BICUBIC (cubic spline interpolation in a 4x4 environment), or ANTIALIAS (a high-quality downsampling filter).

Return type :class:Image object

rotate(*angle*, *filter=0*, *expand=False*)
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.
- **filter** – An optional resampling filter. This can be one of NEAREST (use nearest neighbour), BILINEAR (linear interpolation in a 2x2 environment), or BICUBIC (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode "1" or "P", it is set NEAREST.
- **expand** – Optional expansion flag. If true, expands the output image to make it large enough to hold the entire rotated image. If false or omitted, make the output image the same size as the input image.

Return type :class:Image object

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 later in this handbook.

You can use a file object instead of a filename. In this case, you must always specify the format. The file object must implement the seek, tell, and write methods, and be opened in binary mode.

Parameters

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

Returns None

Raises `KeyError` If the output format could not be determined from the file name. Use the format option to solve this.

Raises `IOError` If the file could not be written. The file may have been created, and may contain partial 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.

Note that in the current version of the library, most sequence formats only allows you to seek to the next frame.

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

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

See *Image.tell()*

show(*title=None*, *command=None*)
Displays this image. This method is mainly intended for debugging purposes.

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

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

Parameters **title** (*None or string*) – Optional title to use for the image window, where possible.

size = (0, 0)

split()
Split this image into individual bands. This method returns a tuple of individual image bands from an image. For example, splitting an "RGB" image creates three new images each containing a copy of one of the original bands (red, green, blue).

Returns A tuple containing bands.

tell()
Returns the current frame number.

Returns Frame number, starting with 0.

See *Image.seek()*

thumbnail(*size*, *resample=0*)
Create thumbnail representation (modifies image in place)

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 *Image.draft()* method to configure the file reader (where applicable), and finally resizes the image.

Also note that this function modifies the Image object in place. If you need to use the full resolution image as well, apply this method to a *Image.copy()* of the original image.

Parameters

- **size** – Requested size.
- **resample** – Optional resampling filter. This can be one of of NEAREST, BILINEAR, BICUBIC, or ANTIALIAS (best quality). If omitted, it defaults to NEAREST (this will be changed to ANTIALIAS in a future version).

Return type None

tobitmap(*name='image'*)
Return image as an XBM bitmap

tostring(*encoder_name='raw'*, **args*)

transform(*size*, *method*, *data=None*, *resample=0*, *fill=1*)
Transforms this image. This method creates a new image with the given size, and the same mode as the original, and copies data to the new image using the given transform.

Parameters

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

interpolation in a 4x4 environment). If omitted, or if the image has mode “1” or “P”, it is set to NEAREST.

Return type :class:Image object

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

verify()
Verify file contents.

2.2 AWARE Backend

2.3 Graphics Magick Backend

2.3.1 High-Level

2.3.2 Low-Level

2.4 Java Backend

2.4.1 High-Level

2.5 Development

You’ll need to install Sphinx to build the documentation. For convenience, a requirements-devel.pip file has been provided and you may simply use “pip install -r requirements-devel.pip” to keep your dependencies current.

Python Module Index

n

Index

V