PyPNM – Reading, writing and displaying PPM and PGM images in pure Python

Introduction

PPM and PGM image formats

PPM (Portable Pixel Map) and PGM (Portable Gray Map) (particular cases of PNM format group) are simplest file formats for storing RGB and L images as files, correspondingly. This simplicity lead to some adverse consequences:

1. lack of strict official specification. Instead, you may find words like "usual" in format description. Surely, there is always someone who implement this part of image format in unprohibited, yet a totally unusual way.
2. unwillingness of many software developers to spend their precious time on such a trivial task as supporting simple open format. It took years for almighty Adobe Photoshop developers to include PNM module in distribution rather than count on third-party developers, and surely (see above) they took their chance to implement a header scheme nobody else uses. What as to PNM support in Python (say, Pillow), it's often incomplete and requires counterintuitive measures when dealing with specific data types (like 16-bit per channel).

As a result, novice Python user (like the writer of these words) may find it difficult to get simple yet reliable input/output modules for PPM and PGM image files.

Another possible usage of PPM and PGM is displaying images. Tkinter, a widely distributed Python GUI module, have a possibility to display PPM- and PGM-like objects as images. However, one should first convert image data from whatever form used for image processing by program to PNM-like object in memory, and again, it is not too easy to find a simple and ready solution for that.

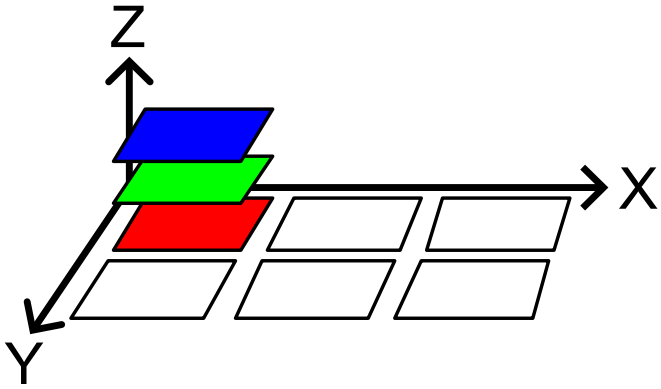
Therefore, main purpose of PyPNM module is providing suitable facilities for PPM and PGM image file reading, viewing and writing to facilitate developing image filtering and editing programs using Python, not suffering from compatibility issues etc. So PyPNM is sort of a bridge between files on a HD and suitable image representation in memory, converting images forth and back.

General information

Keeping PyPNM main goal (image $\rightleftarrows$ file bridge) in mind, let us consider typical image representation in computer memory.

Image ⇄ file: image's part

There is some sort of consensus between image editing software developers regarding RGB image coordinate system:



Pixel [0][0] of an RGB image

Pixel coordinate origin 0, 0 is top left, and color channels, numbered starting from 0, have red, green, blue order.

The most logical image representation using Python data types is a 3D nested list, i.e. an RGB pixel is a list of channels [r, g, b], pixels are combined into row list of pixels [[r, g, b], [r, g, b], [r, g, b]], and rows are combined into image list of rows [[[r, g, b], [r, g, b], [r, g, b]], [[r, g, b], [r, g, b], [r, g, b]]].

It is important to remember that PyPNM is supposed to work with one standard image representation. That is, images of all types are supposed to have the forementioned `list[list[list[int]]]` data type. There are no exceptions, and I mean it: even a greyscale (L) pixel is supposed to be a list of channels. It's just a list of one channel, but it's still a list. This approach allows using the same y, x, z loop for processing any image type instead of developing different algorithms for different image types.

Image ⇄ file: file's part

PyPNM compatibility with different PNM file formats is summarized below.

Image format	File format	Read	Write
16 bits per channel RGB	P6 Binary PPM	✅	✅
8 bits per channel RGB	P3 ASCII PPM	✅	✅
	P6 Binary PPM	✅	✅
	P3 ASCII PPM	✅	✅
16 bits per channel L	P5 Binary PGM	✅	✅
	P2 ASCII PGM	✅	✅
8 bits per channel L	P5 Binary PGM	✅	✅
	P2 ASCII PGM	✅	✅
1 bit ink on/off	P4 Binary PBM	✅	❌
	P1 ASCII PBM	✅	❌

Note that PyPNM provides PBM files reading, but not writing, since 1 bit per channel images are next to useless for image editing in general – most of image processing algorithms simply do not work with them. So the same reason, when reading PBM file, PyPNM promotes image from 1 bit ink on/off to 8 bit L, turning PBM “1” to image “0” and PBM “0” to image “255”, that is, keeping black and white in different color model. As a result, PBM turns into PGM when reading with PyPNM.

PyPNM versions download

There are several download options for PyPNM, summarized below:

Download site	PyPNM version	Content
Git main	Current main version (Python 3.11 and above)	PyPNM module and sample viewer application, illustrating all reading, displaying and writing functions in action.
Git py34	Extended compatibility version (Python 3.4 and above)	PyPNM module and sample viewer application, illustrating all reading, displaying and writing functions in action. Viewer in .34 version include PNG support via PyPNG. PyPNG is not a part of PyPNM and included here because it's a rare case of format support for Python 3.4, allowing to add viewer multiformat reading and conversion support.
PyPI	PyPI package	PyPNM module only, intended for developers. Version match Git py34 branch unless otherwise stated.

Note that .34 version (made compatible with earlier versions of Python) is functionally equivalent to main Git branch, but at some update stages may miss some internal optimizations etc. implemented in main branch. Also, to overcome old Tkinter versions limitations, .34 contains some workarounds which also may slow it down (see Image modes and compatibility issues). From a developer point of view, .34 version misses type hints and other modern stuff simplifying development and usage, and uses rather ugly .join combinations instead of elegant f-strings. Main functions names and their arguments, however, are the same for both versions.

PyPNM functions

PyPNM contains a set of functions, included into main pnmlpnm.py (“pnm-list-pnm”) file. PyPNM may be installed via different methods.

Automated installation with `pip`:

```
pip install pypnm
```

Also, you may acquire PyPNM some other way, for example, from Github, and simply place `pypnm` folder under your main program folder. In both cases, further usage of PyPNM will begin with

```
from pypnm import pnmlpnm
```

which gives you access to all functions, contained in the module. Functions usage is discussed below.

pnm2list

`pnm2list` is a function for reading PPM/PGM file from disk to 3D nested list (image) in memory for image processing. Usage example:

```
X, Y, Z, maxcolors, image3D = pnmlpnm.pnm2list(sourcefilename)
```

where:

`X, Y, Z`: image dimensions (int). X and Y are image width and height in pixels, respectively; Z is the number of channels, 1 for L and 3 for RGB images.

Actually, image dimensions may be detected from `image3D` lengths, but they are needed so often that it's cheaper to have them exported together with image data.

`maxcolors`: number of colors per channel for current image (int). Typically its either 255 or 65535 for 8 bpc and 16 bpc images respectively; while other values are not deliberately prohibited by the specification, they are unlikely to be used in actual files. Note that unlike X, Y and X, maxcolors cannot be logically decided from pixel data so this is required variable.

`image3D`: image pixel data as list(list(list(int))). Note that this structure is used for all image modes, i.e. for L images a pixel is represented as a list of 1 int value, not as just int.

`sourcefilename`: PPM/PGM input file name (str).

list2bin

`list2bin` is a function for converting 3D nested list (image) in memory to PNM-like bytes object in memory to be used for viewing with Tkinter. Usage example:

```
preview_data = pnmlpnm.list2bin(image3D, maxcolors,
show_chessboard=True)
```

where:

`image3D`: Y * X * Z list (image) of lists (rows) of lists (pixels) of ints (channel values);

`maxcolors`: number of colors per channel for current image (int);

`show_chessboard`: optional bool, set `True` to show LA and RGBA images against chessboard pattern; `False` or absent show existing L or RGB data for transparent areas as opaque. Default is `False` for backward compatibility.

`preview_data`: PNM-structured binary data.

Note that `preview_data` above is a bytes object, in-memory copy of PPM/PGM file. To be shown with Tkinter, it should be further converted to PhotoImage object:

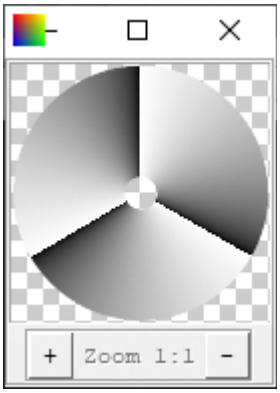
```
preview = PhotoImage(data=preview_data)
```

followed by showing PhotoImage as part of Label, Button etc (see Attachment).

Image modes, alpha channel and `show_chessboard`

`list2bin` function takes any number Z of image channels as input, but skips any channel with number higher than 3 (that is, anything above 4-th channel, which correspond to alpha in RGBA). Images with 1 channel (channel number 0) and 3 channels are treated as L and RGB, correspondingly. Images with 2 and 4 channels are treated as LA and RGBA images, correspondingly, that is, channel number 1 or 3 is considered as alpha channel, and list2bin tries to simulate transparency for it. However, PNM format does not support transparency. As a workaround, list2bin may simulate transparent image preview over chessboard, similar to

Photoshop or GIMP, by premixing image data with artificially generated chess pattern and then generating bytes of such a chess-containing PNM. This behaviour is controlled with `show_chessboard` bool, which is set to `False` (i.e., skip alpha) by default for backward compatibility reasons. With `show_chessboard` set `True`, list2bin generates chessboard as described above and passes the result to display as shown below:



Example of viewer window showing 16 bpc LA image

Note that chessboard pattern is generated algorithmically by list2bit, its size and color, chosen to match Photoshop default “Medium Light” preview, are not controlled by any option in current PyPNM, so edit PyPNM source is you want to change it.

Keep in mind that, while primary goal of this function is showing image data, it may be used for visualizing any other data (e.g., some purely artificial functions etc.) as soon as data fit rectagular nested list and may be mapped to integer in, say, 0..255 range.

Image modes and compatibility issues

Due to its extended compatibility, extended compatibility version (.34) have an ample opportunities to have a compatibility issues. For example, Tkinter bundled with CPython 3.10 can’t preview 16 bit/channel data correctly, and earlier versions simply crash when encountering it, with different version-dependent sorts of errors. This problem is neither PyPNM nor Python-related per se – list2bin flawlessly produces absolutely correct data, but old Tkinter can’t handle it. However, although this is not PyPNM problem, it is still both ungood in general and discombobulating in particular.

As a workaround, list2bin in extended compatibility version (.34) currently includes a routine for color depth reduction from 16 bpc to 8 bpc when generating a preview. Surely list2bin tries to avoid such a remapping until it’s absolutely necessary since remapping requires extra calculation and thus slows the function down; decision on remapping, however, is based on indirect correlation between Python version and bundled Tkinter version (because any attempt to determine Tkinter version directly will require importing Tkinter into PyPNM, while the latter is supposed to have no external dependencies at all, except those included into Python itself). So instead of trying to find Tkinter on your machine PyPNM detects Python version and makes assumptions on what Tkinter version may be bundled with it. Assumption is based on typical official CPython versions distributions content, and may fail if you have custom builds of Tkinter, or Python, or both.

If you have standard installation of CPython 3.11 or above, your version of Tkinter is likely to handle everything without exceptions, and you may use main version which doesn’t spend CPU time on all that old versions workarounds.

list2pnmbin

`list2pnmbin` is a function for converting 3D nested list (image) in memory to binary PNM file on disk. Usage example:

```
pnmlpnm.list2pnmbin(savefilename, image3D, maxcolors)
```

where:

image3D: X * Y * Z list (image) of lists (rows) of lists (pixels) of ints (channels);

maxcolors: number of colors per channel for current image (int);

savefilename: resulting PNM file name.

Note that while byte structure of list2bin output in memory and list2pnm output on disk match, the functions themselves are different because list2pnm tries to reduce memory usage by generating PNM bytes per image row and immediately flushing every row to HD, while list2bin must fully assemble and place whole PNM bytes object into memory anyway.

list2pnmascii

`list2pnmascii` is a function for converting 3D nested list (image) in memory to ASCII PNM file on disk. Usage example:

```
pnmlpnm.list2pnmascii(savefilename, image3D, maxcolors)
```

where:

image3D: X * Y * Z list (image) of lists (rows) of lists (pixels) of ints (channels);

maxcolors: number of colors per channel for current image (int);

savefilename: resulting PNM file name.

As can be seen, arguments for list2pnmascii are the same as for list2pnm.

list2pnm

`list2pnm` is a function for converting 3D nested list (image) in memory to either binary or ASCII PNM file on disk. Usage example:

```
pnmlpnm.list2pnm(savefilename, image3D, maxcolors, bin)
```

where:

image3D: X * Y * Z list (image) of lists (rows) of lists (pixels) of ints (channels);

maxcolors: number of colors per channel for current image (int);

bin: switch defining whether resulting file will be binary or ASCII (bool);

savefilename: resulting PNM file name.

Note that internally list2pnm is merely a switch between list2pnmbin and list2pnmascii, controlled by `bin` value. Default is True, meaning written file will be binary. Using this switch function simplifies writing applications; for example, for Tkinter-based applications you may use single function for saving all sorts of PNM, simply passing arguments via lambda.

create_image

`create_image` function creates 3D image list, filled with zeroes. This function is unnecessary for this particular module, but often needed when working with images, so it appears to be suitable to have it at hand. Usage example:

```
image3D = pnmlpnm.create_image(X, Y, Z)
```

where:

`image3D`: list of zeroes having `X, Y, Z` size, a placeholder to be used for further image editing.

References

1. Netpbm file formats description.

2. PyPNM at PyPI - installing PyPN with pip. Does not contain viewer example etc., only core converter, but provides regular pip-driven automated updates.

3. PyPNM main at Github containing example viewer application, illustrating using list2bin to produce data for Tkinter PhotoImage(data=...) to display, as well as opening/saving various portable map formats.

4. PyPNM for Python 3.4 at Github containing example viewer application, illustrating using list2bin to produce data for Tkinter PhotoImage(data=...) to display, as well as opening/saving various portable map formats. This particular version was tested and shown to work under Python 3.4, Windows XP. This particular distribution also contain PyPNG, providing universal pure Python viewer and converter for PNG and all flavours of PGM and PPM.

5. PyPNM documentation (PDF) – effective version of this very document.

6. Dnyarri website – this and other Python freeware by the same author.

Attachment

PyPNM demo

Below is very short demo program, illustrating all PyPNM functions – opening PPM as image, writing obtained image as binary PPM, writing obtained image as ASCII PPM, and displaying image with Tkinter. This program requires `'example.ppm'` source file (not included into current document) to exist in program directory. Note that resulting files are unlikely to be byte-identical to source file: first, PyPNM never writes unnecessary info like comments, second, as for an ASCII files, PyPNM opinion regarding appropriate formatting may differ from other software one.

```
#!/usr/bin/env python3

from tkinter import Button, PhotoImage, Tk

from pypnm import pnmlpnm

X, Y, Z, maxcolors, image3D = pnmlpnm.pnm2list('example.ppm')  # Open
pnmlpnm.list2pnmbin('binary.ppm', image3D, maxcolors)  # Save as binary
pnmlpnm.list2pnmascii('ascii.ppm', image3D, maxcolors)  # Save as ascii

main_window = Tk()
main_window.title('PyPNM demo')
preview_data = pnmlpnm.list2bin(image3D, maxcolors)  # Generating preview bytes
preview = PhotoImage(data=preview_data)  # Generating preview object from bytes
preview_button = Button(main_window, text='Example\n(click to exit)',
            image=preview, compound='top', command=lambda: main_window.destroy())
preview_button.pack()
main_window.mainloop()
```