PurePNG Documentation

Release 0.3.0

Pavel Zlatovratskii

2017-11-10

Contents

Contents:

CHAPTER 1

What is PurePNG?

PurePNG is pure-Python package for reading and writing PNG.

PurePNG can read and write all PNG formats. PNG supports a generous variety of image formats: RGB or greyscale, with or without an alpha channel; and a choice of bit depths from 1, 2 or 4 (as long as you want greyscale or a pallete), 8, and 16 (but 16 bits is not allowed for palettes). A pixel can vary in size from 1 to 64 bits: 1/2/4/8/16/24/32/48/64. In addition a PNG file can be *interlaced* or not. An interlaced file allows an incrementally refined display of images being downloaded over slow links (yet it's not implemented in PurePNG for now).

PurePNG is written in pure Python(that's why it's called *Pure*). So if you write in Python you can understand code of PurePNG or inspect raw data while debugging.

1.1 Comparison to other PNG tools

The most obvious "competitor" to PurePNG is PIL. Depending on what job you want to do you might also want to use Netpbm (PurePNG can convert to and from the Netpbm PNM format), or use `ctypes` to interface directly to a compiled version of libpng. If you know of others, let me know.

PIL's focus is not PNG. PIL's focus is image processing, and this is where PurePNG sucks. If you want to actually process an image—resize, rotate, composite, crop–then you should use PIL. You may use *PIL Plugin* if you want to use both PurePNG and PIL. In PurePNG you get the image as basically an array of numbers. So some image processing is possible fairly easily, for example cropping to integer coordinates, or gamma conversion, but this very basic.

PurePNG can read and write Netpbm PAM files. PAM is useful as an intermediary format for performing processing; it allows the pixel data to be transferred in a simple format that is easily processed. Netpbm's support for PAM to PNG conversion is more limited than PurePNG's. Netpbm will only convert a source PAM that has 4 channels (for example it does not create greyscale–alpha PNG files from `GRAYSCALE_ALPHA` PAM files). Netpbm's usual tool for create PNG files, `pnmtopng`, requires an alpha channel to be specified in a separate file.

PurePNG has good support for PNG's `sBIT` chunk. This allows end to end processing of files with any bit depth from 1 to 16 (for example a 10-bit scanner may use the `sBIT` chunk to declare that the samples in a 16-bit PNG file are rescaled 10-bit samples; in this case, PurePNG delivers 10-bit samples). Netpbm handle's the `sBIT` chunk in a similar way, but other toolsets may not (e.g. PIL).

`libpng` is made by the PNG gods, so if want to get at all that goodness, then you may want to interface directly to libpng via `ctypes`. That could be a good idea for some things. Installation would be trickier.

1.2 Installation

Because PurePNG is written in Python it's trivial to install into a Python installation. Just use `python setup.py install`.

There is also "light" mode: you can just copy the `../code/png/png.py` file. You can even *curl* it straight into wherever you need it: `curl -LO https://raw.githubusercontent.com/Scondo/purepng/master/code/png/png.py`. This "light" module mode contains all features required for PNG reading and writing, while "full" package mode contains extra features like Cython speedup, other format support, PIL plugin etc.

1.3 PIL Plugin

In "full" package PurePNG provide plugin for usage with PIL instead of PIL's native PNG support. This plugin is in very early stage yet can be useful. Just try it with `from png import PngImagePlugin`

1.3.1 Benefit

- PurePNG rely on python's zlib instead of PIL. So this plugin can be useful when PIL built without zlib support.
- PurePNG handle `sBIT` chunk and rescale values if it's not correctly rescaled on write.
- PurePNG does not use separate palette or transparency when reading, providing full RGB and alpha channel instead.
- PurePNG should write gamma

1.3.2 Miss

- PurePNG does not save custom chunks
- PurePNG does not use zlib dictionary and method (compression level used)

1.4 PurePNG compare to PyPNG

PurePNG is fork of PyPNG - nice and simple module to work with png.

If you work with PyPNG in most cases you can use PurePNG as drop-in replace, but few things are changed:

1.4.1 Buffer, not array

PyPNG document that rows in boxed flat row could be any sequence, but in practice even unit-test check that it should be `array.array`. This changed from `array.array` to any buffer-compatible sequence.

You can use `buffer()` or `memoryview()` functions to fetch row bytes depending on your version of python if you have used `tostring()` before. And of course you may just use rows as sequence.

1.4.2 Python 2.2 no longer supported

Most features were already broken in Python 2.2 and it couldn't be fixed. So support of Python 2.2 is completely removed.

Python 2.2 is pretty old, you know?

1.4.3 PNM|PBM|PAM deprecated in module

For now Netpbm image format kept in `png` module, but it will be moved to a separate module within package. So if you want to work with Netpbm images using PurePNG do not rely on "light" module mode, use "full" package. (see *Installation*)

CHAPTER 2

PurePNG Code Examples

This section discusses some example Python programs that use the png module for reading and writing PNG files.

2.1 Writing

The basic strategy is to create a `Writer` object (instance of `png.Writer`) and then call its `png.write()` method with an open (binary) file, and the pixel data. The `Writer` object encapsulates all the information about the PNG file: image size, colour, bit depth, and so on.

2.1.1 A Ramp

Create a one row image, that has all grey values from 0 to 255. This is a bit like Netpbm's `pgmramp`.

```
import png
f = open('ramp.png', 'wb')      # binary mode is important
w = png.Writer(255, 1, greyscale=True)
w.write(f, [range(256)])
f.close()
```

Note that our single row, generated by `range(256)`, must itself be enclosed in a list. That's because the `png.write()` method expects a list of rows.

From now on `import png` will not be mentioned.

2.1.2 A Little Message

A list of strings holds a graphic in ASCII graphic form. We convert it to a list of integer lists (the required form for the `write()` method), and write it out as a black-and-white PNG (bilevel greyscale).

```
s = ['110010010011',
     '101011010100',
     '110010110101',
     '100010010011']
s = map(lambda x: map(int, x), s)

f = open('png.png', 'wb')
```

```
w = png.Writer(len(s[0]), len(s), greyscale=True, bitdepth=1)
w.write(f, s)
f.close()
```

Note how we use `len(s[0])` (the length of the first row) for the *x* argument and `len(s)` (the number of rows) for the *y* argument.

2.1.3 A Palette

The previous example, "a little message", can be converted to colour simply by creating a PNG file with a palette. The only difference is that a *palette* argument is passed to the `write()` method instead of `greyscale=True`:

```
# Assume f and s have been set up as per previous example
palette=[(0x55,0x55,0x55), (0xff,0x99,0x99)]
w = png.Writer(len(s[0]), len(s), palette=palette, bitdepth=1)
w.write(f, s)
```

Note that the palette consists of two entries (the bit depth is 1 so there are only 2 possible colours). Each entry is an RGB triple. If we wanted transparency then we can use RGBA 4-tuples for each palette entry.

2.1.4 Colour

For colour images the input rows are generally 3 times as long as for greyscale, because there are 3 channels, RGB, instead of just one, grey. Below, the *p* literal has 2 rows of 9 values (3 RGB pixels per row). The spaces are just for your benefit, to mark out the separate pixels; they have no meaning in the code.

```
p = [(255,0,0, 0,255,0, 0,0,255),
     (128,0,0, 0,128,0, 0,0,128)]
f = open('swatch.png', 'wb')
w = png.Writer(3, 2)
w.write(f, p) ; f.close()
```

2.1.5 More Colour

A further colour example illustrates some of the manoeuvres you have to perform in Python to get the pixel data in the right format.

Say we want to produce a PNG image with 1 row of 8 pixels, with all the colours from a 3-bit colour system (with 1-bit for each channel; such systems were common on 8-bit micros from the 1980s).

We produce all possible 3-bit numbers:

```
>>> range(8)
[0, 1, 2, 3, 4, 5, 6, 7]
```

We can convert each number into an RGB triple by assigning bit 0 to blue, bit 1 to red, bit 2 to green (the convention used by a certain 8-bit micro):

```
>>> map(lambda x: (bool(x&2), bool(x&4), bool(x&1)), _)
[(False, False, False), (False, False, True), (True, False, False),
(True, False, True), (False, True, False), (False, True, True), (True,
True, False), (True, True, True)]
```

(later on we will convert False into 0, and True into 255, so don't worry about that just yet). Here we have each pixel as a tuple. We want to flatten the pixels so that we have just one row. In other words instead of [(R,G,B), (R,G,B), ...] we want [R,G,B,R,G,B,...]. It turns out that `itertools.chain(*...)` is just what we need:

```
>>> list(itertools.chain(*_))
[False, False, False, False, False, True, True, False, False, True,
False, True, False, True, False, False, True, True, True, True, False,
True, True, True]
```

Note that the `list` is not necessary, we can usually use the iterator directly instead. I just used `list` here so we can see the result.

Now to convert False to 0 and True to 255 we can multiply by 255 (Python use's Iverson's convention, so `False==0`, `True==1`). We could do that with `map(lambda x:255*x, _)`. Or, we could use a "magic" bound method:

```
>>> map((255).__mul__, _)
[0, 0, 0, 0, 0, 255, 255, 0, 0, 255, 0, 255, 0, 255, 0, 0, 255, 255,
255, 255, 0, 255, 255, 255]
```

Now we write the PNG file out:

```
>>> p=_
>>> f=open('speccy.png', 'wb')
>>> w.write(f, [p]) ; f.close()
```

2.2 Reading

The basic strategy is to create a `Reader` object (a `png.Reader` instance), then call its `png.read()` method to extract the size, and pixel data.

2.2.1 PngSuite

The `Reader()` constructor can take either a filename, a file-like object, or a sequence of bytes directly. Here we use `urllib` to download a PNG file from the internet.

```
>>> r=png.Reader(file=urllib.urlopen('http://www.schaik.com/pngsuite/basn0g02.png
↪'))
>>> r.read()
(32, 32, <itertools.imap object at 0x10b7eb0>, {'greyscale': True,
'alpha': False, 'interlace': 0, 'bitdepth': 2, 'gamma': 1.0})
```

The `png.read()` method returns a 4-tuple. Note that the pixels are returned as an iterator (not always, and the interface doesn't guarantee it; the returned value might be an iterator or a sequence).

```
>>> l=list(_[2])
>>> l[0]
array('B', [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 0, 0, 0, 0,
1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3])
```

We have extracted the top row of the image. Note that the row itself is an `array` (see module `array`), but in general any suitable sequence type may be returned by `read()`. The values in the row are all integers less than 4, because the image has a bit depth of 2.

2.3 NumPy

NumPy is a package for scientific computing with Python. It is not part of a standard Python installation, it is downloaded and installed separately if needed. Numpy's array manipulation facilities make it good for doing certain type of image processing, and scientific users of NumPy may wish to output PNG files for visualisation.

PyPNG does not have any direct integration with NumPy, but the basic data format used by PyPNG, an iterator over rows, is fairly easy to get into two- or three-dimensional NumPy arrays.

The code in this section is extracted from `exnumpy.py`, which is a complete runnable example in the `code/` subdirectory of the source distribution. Code was originally written by Mel Raab, but has been hacked around since then.

2.3.1 PNG to NumPy array (reading)

The best thing to do (I think) is to convert each PyPNG row to a 1-dimensional numpy array, then stack all of those arrays together to make a 2-dimensional array. A number of features make this surprising compact. Say *pngdata* is the row iterator returned from `png.Reader.asDirect()`. The following code will slurp it into a 2-dimensional numpy array:

```
image_2d = numpy.vstack(itertools.imap(numpy.uint16, pngdata))
```

Note that the use of `numpy.uint16`, above, means that an array with data type `numpy.uint16` is created which is suitable for bit depth 16 images. Replace `numpy.uint16` with `numpy.uint8` to create an array with a byte data type (suitable for bit depths up to 8).

2.3.2 Reshaping

For some operations it's easier to have the image data in a 3-dimensional array. This plays to NumPy's strengths:

```
image_3d = numpy.reshape(image_2d,
                         (row_count,column_count,plane_count))
```

2.3.3 NumPy array to PNG (writing)

Reshape your NumPy data into a 2-dimensional array, then use the fact that a NumPy array is an iterator over its rows:

```
    pngWriter.write(pngfile,
                    numpy.reshape(image_3d, (-1, column_count*plane_count)))
```

Currently (writing on 2009-04-16) this generates a warning; this warning appears to be a bug/limitation in NumPy, but it is harmless.

CHAPTER 3

The png Module

CHAPTER 4

Acceleration with Cython

Part of png.py can be compiled with Cython to achieve better performance. Compiled part is `png.BaseFilter()` class now. Compilation use `pngfilters.pxd` file do declare types and override functions.

4.1 Compilation

Compilation will be done automatically during setup process while Cython and c-compiler installed. If you do not want to install binary-compiled part you may skip compilation using `--no-cython` option for setup.py.

When you use pypng without installation you may build cythonized code using `setup.py build_ext --inplace`

4.2 Developing with Cython

If you want to see how Cython compile it's part you can extract compiled part into `pngfilters.py` using `unimport.py` and later compile with Cython like `cython pngfilters.py` Be careful! You should remove `pngfilters.py` after compilation to avoid errors!

Main idea of PurePNG is polyglot so don't use any Cython-specific construction in `png.py` - you will broke pure-python mode which is core of all. If you have want to improve performance using such things - separate this in function and write twice: in `png.py` using pure-python syntax and in `pngfilters.pxd` using cython and `cdef inline`.

If you modify part of `png.py` that should be compiled and know nothing about cython feel free to commit and pull request - someone should fix things you can break before release. So if you want to make release - pass unittest both with and without compiled part.

CHAPTER 5

Roadmap and versions

PurePNG use odd/even minor version numbering with odd for development and even for stable versions.

5.1 PyPNG

PyPNG with it’s 0.0.* version could be treated as previous stable version of PurePNG. David Jones works carefully on this.

5.2 0.2

- Reworked Cython concept.
- Add optional filtering on save.
- Module/package duality
- Python 2/3 polyglot (and partitial Cython)
- Using bytearray when possible.
- PIL plugin
- More chunks: text, resolution, colour intent

5.3 0.3 ==> 0.4

- Provide optimisation functions like ‘try to pallete’ or ‘try to greyscale’
- Separate pnm support to module within package
- Rework iccp module to become part of package
- Better text support
- Enhance PIL plugin, support ‘raw’ reading with palette handled by PIL

5.4 Future

- Cython-accelerated scaling
- Support more chunks at least for direct reading|embeding.
- Integrate most tools (incl. picture formats) into package
- Other Cython acceleration when possible

Reports:

CHAPTER 6

PNG: Chunk by Chunk

The PNG specification defines 18 chunk types. This document is intended to help users who are interested in a particular PNG chunk type. If you have a particular PNG chunk type in mind, you can look here to see what support PurePNG provides for it.

6.1 Critical Chunks

6.1.1 `IHDR`

Generated automatically by PurePNG. The `IHDR` chunk specifies image size, colour model, bit depth, and interlacing. All possible (valid) combinations can be produced with suitable arguments to the `png.Writer` class.

6.1.2 `PLTE`

Correctly handled when a PNG image is read. Can be generated for a colour type 3 image by using the `palette` argument to the `png.Writer` class. PNG images with colour types other than 3 can also have a `PLTE` chunk (a suggested palette); it is not currently possible to add a `PLTE` chunk for these images using PyPNG.

6.1.3 `IDAT`

Generated automatically from the pixel data presented to PurePNG. Multiple `IDAT` chunks (of bounded size) can be generated by using `chunk_limit` argument to the `png.Writer` class.

6.1.4 `IEND`

Generated automatically.

6.2 Ancillary Chunks

6.2.1 `tRNS`

Generated for most colour types when the `transparent` argument is supplied to the `png.Writer` to specify a transparent colour. For colour type 3, colour mapped images, a `tRNS` chunk will be generated automatically from the `palette` argument when a palette with alpha (opacity) values is supplied.

6.2.2 `cHRM`

When reading a PNG image the `cHRM` chunk is converted to a tuples `white_point` (2-tuple of floating point values) and `rgb_points` (3-tuple of 2-tuple of floating point) in the `info` dictionary. When writing, `white_point` and `rgb_points` arguments to the `png.Writer` class or calling apropriate `set_` methods generate a `cHRM` chunk (only both, single will be ignored).

6.2.3 `gAMA`

When reading a PNG image the `gAMA` chunk is converted to a floating point gamma value; this value is returned in the `info` dictionary: `info['gamma']`. When writing, the `gamma` argument to the `png.Writer` class will generate a `gAMA` chunk.

6.2.4 `iCCP`

When reading a PNG image the `iCCP` chunk is saved as raw bytes and name. These data returned in the `info` dictionary: `info['icc_profile']`, `info['icc_profile_name']`. When writing, the `icc_profile` argument to the `png.Writer` class will generate a `iCCP` chunk, with name supplied in `icc_profile_name` argument or “ICC Profile” as default.

6.2.5 `sBIT`

When reading a PNG image the `sBIT` chunk will make PyPNG rescale the pixel values so that they all have the width implied by the `sBIT` chunk. It is possible for a PNG image to have an `sBIT` chunk that specifies 3 different values for the significant bits in each of the 3 colour channels. In this case PyPNG only uses the largest value. When writing a PNG image, an `sBIT` chunk will be generated if need according to the `bitdepth` argument specified. Values other than 1, 2, 4, 8, or 16 will generate an `sBIT` chunk, as will values less than 8 for images with more than one plane.

6.2.6 `sRGB`

When reading a PNG image the `sRGB` chunk is read to an integer value; this value is returned in the `info` dictionary: `info['rendering_intent']` and can be compared to values like `png.PERCEPTUAL`. When writing, the `rendering_intent` argument to the `png.Writer` class will generate a `sRGB` chunk.

6.2.7 `tEXt`

When reading a PNG image the `tEXt` chunks are converted to a dictionary of keywords and unicode values in the `info` dictionary: `info['text']`. When writing, the `text` argument with same dict to the `png.Writer` class or arguments with registered keywords names will generate `tEXt` chunks.

6.2.8 `zTXt`

When reading a PNG image the `zTXt` chunks are converted to a dictionary of keywords and unicode values in the `info` dictionary: `info['text']`. It's not possible to write `zTXt` chunsk for now, only `tEXt` will be written with `text` keyword.

6.2.9 `iTXt`

When reading append to `text` info same as `tEXt` or `zTXt`, translated keyword and language tags ignored.

Keywords within `text` that does not fit latin-1 will be saved as `iTXt`

6.2.10 `bKGD`

When a PNG image is read, a `bKGD` chunk will add the `background` key to the `info` dictionary. When writing a PNG image, a `bKGD` chunk will be generated when the `background` argument is used.

6.2.11 `hIST`

Ignored when reading. Not generated.

6.2.12 `pHYs`

When reading a PNG image the `pHYs` chunk is converted to form ((<pixel_per_unit_x>, <pixel_per_unit_y>), <unit_is_meter>) This tuple is returned in the `info` dictionary: `info['resolution']`. When writing, the `resolution` argument to the `png.Writer` class will generate a `pHYs` chunk. Argument could be tuple same as reading result, but also possible some usability modificatuion:

- if both resolutions are same it could be written as single number instead of tuple: (<pixel_per_unit_x>, <unit_is_meter>)
- all three parameters could be written in row: (<pixel_per_unit_x>, <pixel_per_unit_y>, <unit_is_meter>)
- **instead of <unit_is_meter> bool it's possible to use some unit specification:**
    1. omit this part if no unit specified ((<pixel_per_unit_x>, <pixel_per_unit_y>),)
    2. use text name of unit (300, 'i') 'i', 'cm' and 'm' supported for now.

6.2.13 `sPLT`

Ignored when reading. Not generated.

6.2.14 `tIME`

When reading generate `last_mod_time` tuple which is time.structtime compatible.

`png.Writer` have method `png.Writer.set_modification_time()` which could be used to specify `tIME` value or indicate that it should be calculated as file writing time.

6.3 PNG Extensions Chunks

See ftp://ftp.simplesystems.org/pub/png/documents/pngextensions.html

6.3.1 `oFFs`

Ignored when reading. Not generated.

6.3.2 `pCAL`

Ignored when reading. Not generated.

6.3.3 `sCAL`

Ignored when reading. Not generated.

6.3.4 `gIFg`

Ignored when reading. Not generated.

6.3.5 `gIFx`

Ignored when reading. Not generated.

6.3.6 `sTER`

Ignored when reading. Not generated.

6.3.7 `dSIG`

Ignored when reading. Not generated.

6.3.8 `fRAc`

Ignored when reading. Not generated.

6.3.9 `gIFt`

Ignored when reading. Not generated.

6.4 Non-standard Chunks

Generally it is not possible to generate PNG images with any other chunk types. When reading a PNG image, processing it using the chunk interface, `png.Reader.chunks`, will allow any chunk to be processed (by user code).

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index