PurePNG Documentation

Release 0.2.0

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

Pure Python PNG Reader/Writer

This Python module implements support for PNG images (see PNG specification at http://www.w3.org/TR/2003/REC-PNG-20031110/). It reads and writes PNG files with all allowable bit depths (1/2/4/8/16/24/32/48/64 bits per pixel) and colour combinations: greyscale (1/2/4/8/16 bit); RGB, RGBA, LA (greyscale with alpha) with 8/16 bits per channel; colour mapped images (1/2/4/8 bit). Adam7 interlacing is supported for reading and writing. A number of optional chunks can be specified (when writing) and understood (when reading): `tRNS`, `bKGD`, `gAMA`.

For help, type `import png; help(png)` in your python interpreter.

A good place to start is the *`Reader`* and *`Writer`* classes.

Requires Python 2.3. Best with Python 2.6 and higher. Installation is trivial, but see the `README.txt` file (with the source distribution) for details.

This file can also be used as a command-line utility to convert Netpbm PNM files to PNG, and the reverse conversion from PNG to PNM. The interface is similar to that of the `pnmtopng` program from Netpbm. Type `python png.py --help` at the shell prompt for usage and a list of options.

3.1 A note on spelling and terminology

Generally British English spelling is used in the documentation. So that's "greyscale" and "colour". This not only matches the author's native language, it's also used by the PNG specification.

The major colour models supported by PNG (and hence by this module) are: greyscale, RGB, greyscale–alpha, RGB–alpha. These are sometimes referred to using the abbreviations: L, RGB, LA, RGBA. In this case each letter abbreviates a single channel: *L* is for Luminance or Luma or Lightness which is the channel used in greyscale images; *R*, *G*, *B* stand for Red, Green, Blue, the components of a colour image; *A* stands for Alpha, the opacity channel (used for transparency effects, but higher values are more opaque, so it makes sense to call it opacity).

3.2 A note on formats

When getting pixel data out of this module (reading) and presenting data to this module (writing) there are a number of ways the data could be represented as a Python value. Generally this module uses one of three formats called "flat row flat pixel", "boxed row flat pixel", and "boxed row boxed pixel". Basically the concern is whether each pixel and each row comes in its own little tuple (box), or not.

Consider an image that is 3 pixels wide by 2 pixels high, and each pixel has RGB components:

Boxed row flat pixel:

```
iter([R,G,B, R,G,B, R,G,B],
     [R,G,B, R,G,B, R,G,B])
```

Each row appears as its own sequence, but the pixels are flattened so that three values for one pixel simply follow the three values for the previous pixel. This is the most common format used, because it provides a good compromise between space and convenience. Row sequence supposed to be compatible with 'buffer' protocol in addition to standard sequence methods so 'buffer()' can be used to get fast per-byte access. All rows are contained in iterable or iterable-compatible container. (use 'iter()' to ensure)

Flat row flat pixel:

```
[R,G,B, R,G,B, R,G,B,
 R,G,B, R,G,B, R,G,B]
```

The entire image is one single giant sequence of colour values. Generally an array will be used (to save space), not a list.

Boxed row boxed pixel:

```
list([ (R,G,B), (R,G,B), (R,G,B) ],
     [ (R,G,B), (R,G,B), (R,G,B) ])
```

Each row appears in its own list, but each pixel also appears in its own tuple. A serious memory burn in Python.

In all cases the top row comes first, and for each row the pixels are ordered from left-to-right. Within a pixel the values appear in the order, R-G-B-A (or L-A for greyscale–alpha).

There is a fourth format, mentioned because it is used internally, is close to what lies inside a PNG file itself, and has some support from the public API. This format is called packed. When packed, each row is a sequence of bytes (integers from 0 to 255), just as it is before PNG scanline filtering is applied. When the bit depth is 8 this is essentially the same as boxed row flat pixel; when the bit depth is less than 8, several pixels are packed into each byte; when the bit depth is 16 (the only value more than 8 that is supported by the PNG image format) each pixel value is decomposed into 2 bytes (and *packed* is a misnomer). This format is used by the *Writer.write_packed()* method. It isn't usually a convenient format, but may be just right if the source data for the PNG image comes from something that uses a similar format (for example, 1-bit BMPs, or another PNG file).

3.3 And now, my famous members

class png.**Image**(*rows*, *info*)

A PNG image.

You can create an *Image* object from an array of pixels by calling *png.from_array()*. It can be saved to disk with the *save()* method.

The constructor is not public. Please do not call it.

save(*file*)

Save the image to *file*.

If *file* looks like an open file descriptor then it is used, otherwise it is treated as a filename and a fresh file is opened.

In general, you can only call this method once; after it has been called the first time and the PNG image has been saved, the source data will have been streamed, and cannot be streamed again.

class png.**Reader**(*_guess=None*, ***kw*)

PNG decoder in pure Python.

Create a PNG decoder object.

The constructor expects exactly one keyword argument. If you supply a positional argument instead, it will guess the input type. You can choose among the following keyword arguments:

filename Name of input file (a PNG file).

file A file-like object (object with a read() method).

bytes `array` or `string` with PNG data.

asDirect()
Returns the image data as a direct representation of an `x * y * planes` array. This method is intended to remove the need for callers to deal with palettes and transparency themselves. Images with a palette (colour type 3) are converted to RGB or RGBA; images with transparency (a `tRNS` chunk) are converted to LA or RGBA as appropriate. When returned in this format the pixel values represent the colour value directly without needing to refer to palettes or transparency information.

Like the *read()* method this method returns a 4-tuple:

(*width*, *height*, *pixels*, *meta*)

This method normally returns pixel values with the bit depth they have in the source image, but when the source PNG has an `sBIT` chunk it is inspected and can reduce the bit depth of the result pixels; pixel values will be reduced according to the bit depth specified in the `sBIT` chunk (PNG nerds should note a single result bit depth is used for all channels; the maximum of the ones specified in the `sBIT` chunk. An RGB565 image will be rescaled to 6-bit RGB666).

The *meta* dictionary that is returned reflects the *direct* format and not the original source image. For example, an RGB source image with a `tRNS` chunk to represent a transparent colour, will have `planes=3` and `alpha=False` for the source image, but the *meta* dictionary returned by this method will have `planes=4` and `alpha=True` because an alpha channel is synthesized and added.

pixels is the pixel data in boxed row flat pixel format (just like the *read()* method).

All the other aspects of the image data are not changed.

asFloat(*maxval=1.0*)
Return image pixels as per *asDirect()* method, but scale all pixel values to be floating point values between 0.0 and *maxval*.

asRGB()
Return image as RGB pixels.

RGB colour images are passed through unchanged; greyscales are expanded into RGB triplets (there is a small speed overhead for doing this).

An alpha channel in the source image will raise an exception.

The return values are as for the *read()* method except that the *metadata* reflect the returned pixels, not the source image. In particular, for this method `metadata['greyscale']` will be `False`.

asRGB8()
Return the image data as an RGB pixels with 8-bits per sample.

This is like the *asRGB()* method except that this method additionally rescales the values so that they are all between 0 and 255 (8-bit). In the case where the source image has a bit depth < 8 the transformation preserves all the information; where the source image has bit depth > 8, then rescaling to 8-bit values loses precision. No dithering is performed. Like *asRGB()*, an alpha channel in the source image will raise an exception.

This function returns a 4-tuple: (*width*, *height*, *pixels*, *metadata*). *width*, *height*, *metadata* are as per the *read()* method.

pixels is the pixel data in boxed row flat pixel format.

asRGBA()
Return image as RGBA pixels.

Greyscales are expanded into RGB triplets; an alpha channel is synthesized if necessary. The return values are as for the *read()* method except that the *metadata* reflect the returned pixels, not

the source image. In particular, for this method `metadata['greyscale']` will be `False`, and `metadata['alpha']` will be `True`.

asRGBA8()
Return the image data as RGBA pixels with 8-bits per sample.

This method is similar to *asRGB8()* and *asRGBA()*: The result pixels have an alpha channel, *and* values are rescaled to the range 0 to 255. The alpha channel is synthesized if necessary (with a small speed penalty).

chunk(*seek=None*, *lenient=False*)
Read the next PNG chunk from the input file

returns a (*chunk_type*, *data*) tuple. *chunk_type* is the chunk's type as a byte string (all PNG chunk types are 4 bytes long). *data* is the chunk's data content, as a byte string.

If the optional *seek* argument is specified then it will keep reading chunks until it either runs out of file or finds the chunk_type specified by the argument. Note that in general the order of chunks in PNGs is unspecified, so using *seek* can cause you to miss chunks.

If the optional *lenient* argument evaluates to *True*, checksum failures will raise warnings rather than exceptions.

chunklentype()
Reads just enough of the input to determine the next chunk's length and type, returned as a (*length*, *chunk_type*) pair where *chunk_type* is a string. If there are no more chunks, `None` is returned.

chunks()
Return an iterator that will yield each chunk as a (*chunktype*, *content*) pair.

deinterlace(*raw*)
Read raw pixel data, undo filters, deinterlace, and flatten.

Return in flat row flat pixel format.

idat(*lenient=False*)
Iterator that yields all the `IDAT` chunks as strings.

idatdecomp(*lenient=False*, *max_length=0*)
Iterator that yields decompressed `IDAT` strings.

iterboxed(*rows*)
Iterator that yields each scanline in boxed row flat pixel format.

rows should be an iterator that yields the bytes of each row in turn.

iterstraight(*raw*)
Iterator that undoes the effect of filtering

Yields each row in serialised format (as a sequence of bytes). Assumes input is straightlaced. *raw* should be an iterable that yields the raw bytes in chunks of arbitrary size.

palette(*alpha='natural'*)
Returns a palette that is a sequence of 3-tuples or 4-tuples

Synthesizing it from the `PLTE` and `tRNS` chunks. These chunks should have already been processed (for example, by calling the *preamble()* method). All the tuples are the same size: 3-tuples if there is no `tRNS` chunk, 4-tuples when there is a `tRNS` chunk. Assumes that the image is colour type 3 and therefore a `PLTE` chunk is required.

If the *alpha* argument is `'force'` then an alpha channel is always added, forcing the result to be a sequence of 4-tuples.

preamble(*lenient=False*)
Extract the image metadata

Extract the image metadata by reading the initial part of the PNG file up to the start of the `IDAT` chunk. All the chunks that precede the `IDAT` chunk are read and either processed for metadata or discarded.

If the optional *lenient* argument evaluates to *True*, checksum failures will raise warnings rather than exceptions.

process_chunk(*lenient=False*)
Process the next chunk and its data.

If the optional *lenient* argument evaluates to *True*, checksum failures will raise warnings rather than exceptions.

read(*lenient=False*)
Read the PNG file and decode it.

Returns (*width*, *height*, *pixels*, *metadata*).

May use excessive memory.

pixels are returned in boxed row flat pixel format.

If the optional *lenient* argument evaluates to True, checksum failures will raise warnings rather than exceptions.

read_flat()
Read a PNG file and decode it into flat row flat pixel format.

Returns (*width*, *height*, *pixels*, *metadata*).

May use excessive memory.

pixels are returned in flat row flat pixel format.

See also the *read()* method which returns pixels in the more stream-friendly boxed row flat pixel format.

serialtoflat(*raw*, *width=None*)
Convert serial format (byte stream) pixel data to flat row flat pixel.

validate_signature()
If signature (header) has not been read then read and validate it

class png.**Writer**(*width=None*, *height=None*, *greyscale=False*, *alpha=False*, *bitdepth=8*, *palette=None*, *transparent=None*, *background=None*, *gamma=None*, *compression=None*, *interlace=False*, *chunk_limit=1048576*, *filter_type=None*, *icc_profile=None*, *icc_profile_name='ICC Profile'*, ***kwargs*)
PNG encoder in pure Python.

Create a PNG encoder object.

Arguments:

width, height Image size in pixels, as two separate arguments.

greyscale Input data is greyscale, not RGB.

alpha Input data has alpha channel (RGBA or LA).

bitdepth Bit depth: from 1 to 16.

palette Create a palette for a colour mapped image (colour type 3).

transparent Specify a transparent colour (create a `tRNS` chunk).

background Specify a default background colour (create a `bKGD` chunk).

gamma Specify a gamma value (create a `gAMA` chunk).

compression zlib compression level: 0 (none) to 9 (more compressed); default: -1 or None.

interlace Create an interlaced image.

chunk_limit Write multiple `IDAT` chunks to save memory.

filter_type Enable and specify PNG filter

icc_profile Write ICC Profile

icc_profile_name Name for ICC Profile

Extra keywords:

text see *set_text()*

modification_time see *set_modification_time()*

resolution see *set_resolution()*

The image size (in pixels) can be specified either by using the *width* and *height* arguments, or with the single *size* argument. If *size* is used it should be a pair (*width*, *height*).

greyscale and *alpha* are booleans that specify whether an image is greyscale (or colour), and whether it has an alpha channel (or not).

bitdepth specifies the bit depth of the source pixel values. Each source pixel value must be an integer between 0 and `2**bitdepth-1`. For example, 8-bit images have values between 0 and 255. PNG only stores images with bit depths of 1,2,4,8, or 16. When *bitdepth* is not one of these values, the next highest valid bit depth is selected, and an `sBIT` (significant bits) chunk is generated that specifies the original precision of the source image. In this case the supplied pixel values will be rescaled to fit the range of the selected bit depth.

The details of which bit depth / colour model combinations the PNG file format supports directly, are somewhat arcane (refer to the PNG specification for full details). Briefly: "small" bit depths (1,2,4) are only allowed with greyscale and colour mapped images; colour mapped images cannot have bit depth 16.

For colour mapped images (in other words, when the *palette* argument is specified) the *bitdepth* argument must match one of the valid PNG bit depths: 1, 2, 4, or 8. (It is valid to have a PNG image with a palette and an `sBIT` chunk, but the meaning is slightly different; it would be awkward to press the *bitdepth* argument into service for this.)

The *palette* option, when specified, causes a colour mapped image to be created: the PNG colour type is set to 3; *greyscale* must not be set; *alpha* must not be set; *transparent* must not be set; the bit depth must be 1, 2, 4, or 8. When a colour mapped image is created, the pixel values are palette indexes and the *bitdepth* argument specifies the size of these indexes (not the size of the colour values in the palette).

The palette argument value should be a sequence of 3- or 4-tuples. 3-tuples specify RGB palette entries; 4-tuples specify RGBA palette entries. If both 4-tuples and 3-tuples appear in the sequence then all the 4-tuples must come before all the 3-tuples. A `PLTE` chunk is created; if there are 4-tuples then a `tRNS` chunk is created as well. The `PLTE` chunk will contain all the RGB triples in the same sequence; the `tRNS` chunk will contain the alpha channel for all the 4-tuples, in the same sequence. Palette entries are always 8-bit.

If specified, the *transparent* and *background* parameters must be a tuple with three integer values for red, green, blue, or a simple integer (or singleton tuple) for a greyscale image.

If specified, the *gamma* parameter must be a positive number (generally, a *float*). A `gAMA` chunk will be created. Note that this will not change the values of the pixels as they appear in the PNG file, they are assumed to have already been converted appropriately for the gamma specified.

The *compression* argument specifies the compression level to be used by the `zlib` module. Values from 1 to 9 specify compression, with 9 being "more compressed" (usually smaller and slower, but it doesn't always work out that way). 0 means no compression. -1 and `None` both mean that the default level of compession will be picked by the `zlib` module (which is generally acceptable).

If *interlace* is true then an interlaced image is created (using PNG's so far only interace method, *Adam7*). This does not affect how the pixels should be presented to the encoder, rather it changes how they are arranged into the PNG file. On slow connexions interlaced images can be partially decoded by the browser to give a rough view of the image that is successively refined as more image data appears.

Note: Enabling the *interlace* option requires the entire image to be processed in working memory.

---

chunk_limit is used to limit the amount of memory used whilst compressing the image. In order to avoid using large amounts of memory, multiple `IDAT` chunks may be created.

filter_type is number or name of filter type for better compression see http://www.w3.org/TR/PNG/#9Filter-types for details It's also possible to use adaptive strategy for choosing filter type per row. Predefined strategies are *sum* and *entropy*. Custom strategies can be added with *register_extra_filter()* or be callable passed with this argument. (see more at *register_extra_filter()*)

array_scanlines (*pixels*)
Generates boxed rows (flat pixels) from flat rows (flat pixels) in an array.

array_scanlines_interlace (*pixels*)
Generator for interlaced scanlines from an array.

pixels is the full source image in flat row flat pixel format. The generator yields each scanline of the reduced passes in turn, in boxed row flat pixel format.

convert_pnm (*infile*, *outfile*)
Convert a PNM file containing raw pixel data into a PNG file with the parameters set in the writer object. Works for (binary) PGM, PPM, and PAM formats.

convert_ppm_and_pgm (*ppmfile*, *pgmfile*, *outfile*)
Convert a PPM and PGM file containing raw pixel data into a PNG outfile with the parameters set in the writer object.

file_scanlines (*infile*)
Generates boxed rows in flat pixel format, from the input file.

It assumes that the input file is in a "Netpbm-like" binary format, and is positioned at the beginning of the first pixel. The number of pixels to read is taken from the image dimensions (*width*, *height*, *planes*) and the number of bytes per value is implied by the image *bitdepth*.

idat (*rows*, *packed=False*)
Generator that produce IDAT chunks from rows

set_modification_time (*modification_time=True*)
Add time to be written as last modification time

When called after initialisation configure to use time of writing file

set_rendering_intent (*rendering_intent*)
Set rendering intent variant for sRGB chunk

set_resolution (*resolution=None*)
Add physical pixel dimensions

resolution supposed two be tuple of two parameterts: pixels per unit and unit type; unit type may be omitted pixels per unit could be simple integer or tuple of (ppu_x, ppu_y) Also possible to use all three parameters im row

- resolution = ((1, 4),) # wide pixels (4:1) without unit specifier
- resolution = (300, 'inch') # 300dpi in both dimensions
- resolution = (4, 1, 0) # tall pixels (1:4) without unit specifier

set_rgb_points (*rgb_points*, **args*)
Set rgb points part of cHRM chunk

set_text (*text=None*, ***kwargs*)
Add textual information.

All pairs in dictionary will be written, but keys should be latin-1; registered keywords could be used as arguments.

When called more than once overwrite exist data.

set_white_point(*white_point*, *point2=None*)
Set white point part of cHRM chunk

write(*outfile*, *rows*)
Write a PNG image to the output file.

rows should be an iterable that yields each row in boxed row flat pixel format. The rows should be the rows of the original image, so there should be `self.height` rows of `self.width * self.planes` values. If *interlace* is specified (when creating the instance), then an interlaced PNG file will be written. Supply the rows in the normal image order; the interlacing is carried out internally.

Note: Interlacing will require the entire image to be in working memory.

---

write_array(*outfile*, *pixels*)
Write an array in flat row flat pixel format as a PNG file on the output file. See also *write()* method.

write_idat(*outfile*, *idat_sequence*)
Write png with IDAT to file

idat_sequence should be iterable that produce IDAT chunks compatible with *Writer* configuration.

write_packed(*outfile*, *rows*)
Write PNG file to *outfile*.

The pixel data comes from *rows* which should be in boxed row packed format. Each row should be a sequence of packed bytes.

Technically, this method does work for interlaced images but it is best avoided. For interlaced images, the rows should be presented in the order that they appear in the file.

This method should not be used when the source image bit depth is not one naturally supported by PNG; the bit depth should be 1, 2, 4, 8, or 16.

write_passes(*outfile*, *rows*, *packed=False*)
Write a PNG image to the output file.

Most users are expected to find the *write()* or *write_array()* method more convenient.

The rows should be given to this method in the order that they appear in the output file. For straight-laced images, this is the usual top to bottom ordering, but for interlaced images the rows should have already been interlaced before passing them to this function.

rows should be an iterable that yields each row. When *packed* is `False` the rows should be in boxed row flat pixel format; when *packed* is `True` each row should be a packed sequence of bytes.

exception `png.`**Error**
Generic PurePNG error

exception `png.`**FormatError**
Problem with input file format.

In other words, PNG file does not conform to the specification in some way and is invalid.

exception `png.`**ChunkError**
Error in chunk handling

`png.`**register_extra_filter**(*selector*, *name*)
Register adaptive filter selection strategy for futher usage.

selector - callable like `def(line, cfg, filter_obj)`

- line - line for filtering
- cfg - dict with optional tuning
- filter_obj - instance of this class to get context or apply base filters

callable should return chosen line

name - name which may be used later to recall this strategy

png.**write_chunks**(*out*, *chunks*)
Create a PNG file by writing out the chunks.

png.**from_array**(*a*, *mode=None*, *info=None*)
Create a PNG *Image* object from a 2- or 3-dimensional array.

One application of this function is easy PIL-style saving: `png.from_array(pixels, 'L').save('foo.png')`.

Unless they are specified using the *info* parameter, the PNG's height and width are taken from the array size. For a 3 dimensional array the first axis is the height; the second axis is the width; and the third axis is the channel number. Thus an RGB image that is 16 pixels high and 8 wide will use an array that is 16x8x3. For 2 dimensional arrays the first axis is the height, but the second axis is `width*channels`, so an RGB image that is 16 pixels high and 8 wide will use a 2-dimensional array that is 16x24 (each row will be 8*3 = 24 sample values).

mode is a string that specifies the image colour format in a PIL-style mode. It can be:

'L' greyscale (1 channel)

'LA' greyscale with alpha (2 channel)

'RGB' colour image (3 channel)

'RGBA' colour image with alpha (4 channel)

The mode string can also specify the bit depth (overriding how this function normally derives the bit depth, see below). Appending `';16'` to the mode will cause the PNG to be 16 bits per channel; any decimal from 1 to 16 can be used to specify the bit depth.

When a 2-dimensional array is used *mode* determines how many channels the image has, and so allows the width to be derived from the second array dimension.

The array is expected to be a `numpy` array, but it can be any suitable Python sequence. For example, a list of lists can be used: `png.from_array([[0, 255, 0], [255, 0, 255]], 'L')`. The exact rules are: `len(a)` gives the first dimension, height; `len(a[0])` gives the second dimension; `len(a[0][0])` gives the third dimension, unless an exception is raised in which case a 2-dimensional array is assumed. It's slightly more complicated than that because an iterator of rows can be used, and it all still works. Using an iterator allows data to be streamed efficiently.

The bit depth of the PNG is normally taken from the array element's datatype (but if *mode* specifies a bitdepth then that is used instead). The array element's datatype is determined in a way which is supposed to work both for `numpy` arrays and for Python `array.array` objects. A 1 byte datatype will give a bit depth of 8, a 2 byte datatype will give a bit depth of 16. If the datatype does not have an implicit size, for example it is a plain Python list of lists, as above, then a default of 8 is used.

The *info* parameter is a dictionary that can be used to specify metadata (in the same style as the arguments to the *png.Writer* class). For this function the keys that are useful are:

height overrides the height derived from the array dimensions and allows *a* to be an iterable.

width overrides the width derived from the array dimensions.

bitdepth overrides the bit depth derived from the element datatype (but must match *mode* if that also specifies a bit depth).

Generally anything specified in the *info* dictionary will override any implicit choices that this function would otherwise make, but must match any explicit ones. For example, if the *info* dictionary has a `greyscale` key then this must be true when mode is `'L'` or `'LA'` and false when mode is `'RGB'` or `'RGBA'`.

png.**parse_mode**(*mode*, *default_bitdepth=None*)
Parse PIL-style mode and return tuple (grayscale, alpha, bitdeph)

png.**read_pam_header**(*infile*)
Read (the rest of a) PAM header.

infile should be positioned immediately after the initial 'P7' line (at the beginning of the second line). Returns are as for *read_pnm_header*.

png.**read_pnm_header**(*infile*, *supported=('P5', 'P6')*)
Read a PNM header, returning (format,width,height,depth,maxval).

width and *height* are in pixels. *depth* is the number of channels in the image; for PBM and PGM it is synthesized as 1, for PPM as 3; for PAM images it is read from the header. *maxval* is synthesized (as 1) for PBM images.

png.**write_pnm**(*file*, *width*, *height*, *pixels*, *meta*)
Write a Netpbm PNM/PAM file.

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

p

Index