Pymaging Documentation

Release 0.0.0

Jonas Obrist

February 22, 2016

CHAPTER 1

About

Pymaging is a pure Python imaging library that is compatible both with Python 2 and Python 3.

CHAPTER 2

User Documentation

If you want to learn more about how to use pymaging, this part of the documentation is for you:

2.1 Installation

2.1.1 Requirements

Python Compatiblity

Any of the following:

- Python 2.6 or a higher 2.x release
- Python 3.1 or higher
- PyPy 1.8 or higher
- Any other Python 2.6/Python 3.1 compatible Python implementation.

Required Python packages

- `distribute`

2.1.2 Installing

- Create a virtualenv (in theory this is optional, but just do it)
- `pip install -e git+git://github.com/ojii/pymaging.git#egg=pymaging`
- Install formats. (eg: `pip install -e git+git://github.com/ojii/pymaging-png.git#egg=pymaging-pr`

2.2 Quickstart

2.2.1 Resizing an image

Resizing `myimage.png` to 300x300 pixels and save it as `resized.png`:

```
from pymaging import Image

img = Image.open_from_path('myimage.png')
img = img.resize(300, 300)
img.save('resized.png')
```

CHAPTER 3

Developer Documentation

If you want to hack (or extend) pymaging, this part is for you:

3.1 Setup for development

Clone the git repository using `git clone https://github.com/ojii/pymaging.git`.

To run the tests, simply execute `setup.py test` with a Python version of your choice, or run `./runtests.sh` to run it against all installed (and supported) Python versions.

3.2 Overview

3.2.1 About Image objects

`pymaging.image.Image` vs `pymaging.image.LoadedImage`

There are two main classes for representing an image in pymaging, *`pymaging.image.Image`* and *`pymaging.image.LoadedImage`*. Other than their constructor their APIs are the same. The difference is that *`pymaging.image.Image`* didn't load all the image data from the file yet, whereas *`pymaging.image.LoadedImage`* did. As a general rule, any format should use *`pymaging.image.Image`* so opening an image will first load it's metadata (width, height) before loading all the pixel data (which can consume large amounts of memory). This is useful for users who just want to verify that what they have is an image supported by pymaging and maybe want to know the dimensions of the image before loading it.

`pymaging.image.LoadedImage` should only be used if you have all the pixel data in memory anyway or if there's no way around loading all the data at first, as it's required to extract the meta information.

About loaders

`pymaging.image.Image` takes a loader callable which will be called to actually load the image data. This loader should return a tuple `(pixel_array, palette). pixel_array` should be constructed with :func:`pymaging.pixelarray.get_pixel_array`, whereas ``palette` should either be a palette (list of colors) or `None`.

3.2.2 Pixel arrays

Pixel arrays are the core data structure in which image data is represented in pymaging. Their base class is *pymaging.pixelarray.GenericPixelArray*, but in practice they use one of the specialized subclasses. In almost all cases, you should use *pymaging.pixelarray.get_pixel_array()* to construct pixel arrays, instead of using the classes directly.

pymaging.pixelarray.get_pixel_array() takes the image **data** (as an `array.array`, more on this later), the **width** (in pixels), **height** (in pixels) and the **pixel size** as arguments and returns a, if possible specialized, pixel array.

Pixel size

The **pixel size** indicates how many bytes form a single pixel. It also describes how data is stored in the array passed into the pixel array. A pixel size of one indicates either an image with a palette (where the bytes in the image data are indices into the palette) or a monochrome image. Pixel size 3 is probably the most common and usually indicates RGB, whereas pixel size 4 indicates RGBA.

Given the **pixel size**, the **data** passed into the pixel array is translated into pixels at x/y coordinates through the APIs on pixel array.

Important methods

You should hardly ever manipulate the `data` attribute on pixel arrays directly, instead, you should use the provided APIs that handle things like x/y translation for the given width, height and pixel size.

Pixel array methods usually operate **in place**, if you wish to have a copy of the data, use `copy()`.

`get(x, y)`

Returns the **pixel** (a list of ints) at the given position.

`set(x, y, pixel)`

Sets the given pixel (list of ints) to the given position.

`remove_lines`, `remove_columns`, `add_lines` and `add_columns`

Those four methods are closely related and are used to resize a pixel array (and thus the image canvas). They all take two arguments: `amount` and `offset`.

Warning: There is an important performance caveat with those four methods. Manipulating columns (`add_columns` and `remove_columns`) is slower the more lines there are. Therefore the column manipulating methods should always be called **before add_lines** or **after remove_lines** to keep the amount of lines where columns are changed the lowest.

3.3 Internal API

3.3.1 `pymaging.image`

class `pymaging.image.`**`Image`**(*mode*, *width*, *height*, *loader*, *meta=None*)

The image class. This is the core class of pymaging.

Parameters

- **`mode`** – The color mode. A *`pymaging.colors.ColorType`* instance.
- **`width`** – Width of the image (in pixels).
- **`height`** – Height of the image (in pixels).
- **`loader`** – A callable which when called returns a tuple containing a *`pymaging.pixelarray.GenericPixelArray`* instance and a palette (which can be `None`). If you already have all the pixels for your image loaded, use *`pymaging.image.LoadedImage`* instead.
- **`meta`** – Any further information your format wants to pass along. Your format should document what users can expect in `meta`.

`mode`

The color mode used in this image. A *`pymaging.colors.ColorType`* instance.

`palette`

The palette used in this image. A list of *`pymaging.colors.Color`* instances or `None`.

`reverse_palette`

Cache for *`get_reverse_palette()`*.

`pixels`

The *`pymaging.pixelarray.GenericPixelArray`* (or subclass thereof) instance holding the image data.

`width`

The width of the image (in pixels)

`height`

The height of the image (in pixels)

`pixelsize`

The size of a pixel (in *`pixels`*). `1` usually indicates an image with a palette. `3` is an standard RGB image. `4` is a RGBA image.

classmethod `open`(*fileobj*)

Creates a new image from a file object

Parameters **`fileobj`** – A file like object open for reading.

classmethod `open_from_path`(*filepath*)

Creates a new image from a file path

Parameters **`fileobj`** – A string pointing at a image file.

classmethod `new`(*mode*, *width*, *height*, *background_color*, *palette=None*, *meta=None*)

Creates a new image with a solid background color.

Parameters

- **`mode`** – The color mode. Must be an instance of *`pymaging.colors.ColorType`*.

- **width** – Width of the new image.
- **height** – Height of the new image.
- **background_color** – The color to use for the background. Must be an instance of *pymaging.colors.Color*.
- **palette** – If given, the palette to use for the image.
- **meta** – Any further information your format wants to pass along. Your format should document what users can expect in `meta`.

save (*fileobj*, *format*)
Saves the image.

Parameters

- **fileobj** – A file-like object (opened for writing) to which the image should be saved.
- **format** – The format to use for saving (as a string).

save_to_path(filepath, format=None):
Saves the image to a path.

Parameters

- **filepath** – A string pointing at a (writable) file location where the image should be saved.
- **format** – If given, the format (string) to use for saving. If `None`, the format will be guessed from the file extension used in `filepath`.

get_reverse_palette ()
Returns *reverse_palette*. If *reverse_palette* is `None`, calls *_fill_reverse_palette()*. The reverse palette is a dictionary. If the image has no palette, an empty dictionary is returned.

_fill_reverse_palette ()
Populates the reverse palette, which is a mapping of *pymaging.colors.Color* instances to their index in the palette. Sets *reverse_palette*.

_copy (*pixles*, ***kwargs*)
Creates a copy of this instances meta information, but setting pixel array to `pixels`. `kwargs` can override any argument to the *pymaging.image.LoadedImage* constructor. By default the values of this image are used.

This method is mostly used by other APIs that return a new copy of the image.

Returns a *pymaging.image.LoadedImage*.

resize (*width*, *height*, *resample_algorithm=nearest*, *resize_canvas=True*)
Resizes the image to the given `width` and `height`, using given `resample_algorithm`. If `resize_canvas` is `False`, the actual image dimensions do not change, in which case the excess pixels will be filled by a background color (usually black). Returns the resized copy of this image.

Parameters

- **width** – The new width as integer in pixels.
- **height** – The new height as integer in pixels.
- **resample_algorithm** – The resample algorithm to use. Should be a *pymaging.resample.Resampler* instance.
- **resize_canvas** – Boolean flag whether to resize the canvas or not.

affine(*transform*, *resample_algorithm=nearest*, *resize_canvas=True*)
Advanced version of *resize()*. Instead of a `height` and `width`, a *pymaging.affine.AffineTransform* is passed according to which the image is transformed. Returns the transformed copy of the image.

rotate(*degrees*, *clockwise=False*, *resample_algorithm=nearest*, *resize_canvas=True*)
Rotates the image by `degrees` degrees counter-clockwise (unless `clockwise` is `True`). Interpolation of the pixels is done using `resample_algorithm`. Returns the rotated copy of this image.

get_pixel(*x*, *y*)
Returns the pixel at the given `x`/`y` location. If the pixel is outside the image, raises an `IndexError`. If the image has a palette, the palette lookup will be performed by this method. The pixel is returned as a list if integers.

get_color(*x*, *y*)
Same as *get_pixel()* but returns a *pymaging.colors.Color* instance.

set_color(*x*, *y*, *color*)
The core drawing API. This should be used to draw pixels to the image. Sets the pixel at `x`/`y` to the color given. The color should be a *pymaging.colors.Color* instance. If the image has a palette, only colors that are in the palette are supported.

flip_top_bottom()
Vertically flips the image and returns the flipped copy.

flip_left_right()
Horizontally flips the image and returns the flipped copy.

crop(*width*, *height*, *padding_top*, *padding_left*)
Crops the pixel to the new `width` and `height`, starting the cropping at the offset given with `padding_top` and `padding_left`. Returns the cropped copy of this image.

draw(*shape*, *color*)
Draws the shape using the given color to this image. The shape should be a *pymaging.shapes.BaseShape* subclass instance, or any object that has a `iter_pixels` method, which when called with a *pymaging.colors.Color* instance, returns an iterator that yields tuples of `(x, y, color)` of colors to be drawn to pixels.

This method is just a shortcut around *set_color()* which allows users to write shape classes that do the heavy lifting for them.

This method operates **in place** and does not return a copy of this image!

blit(padding_top, padding_left, image):
Draws the image passed in on top of this image at the location indicated with the padding.

This method operates **in place** and does not return a copy of this image!

class `pymaging.image.`**LoadedImage**(*mode*, *width*, *height*, *pixels*, *palette=None*, *meta=None*)
Subclass of *pymaging.image.Image* if you already have all pixels loaded. All parameters are the same as in *pymaging.image.Image* except for `loader` which is replaced with `pixels`. `pixels` must be an instance of *pymaging.pixelarray.GenericPixelArray* or a subclass thereof.

3.3.2 pymaging.affine

class `pymaging.affine.`**AffineTransform**(*matrix*)
Affine transformation matrix. Used by *pymaging.image.Image.affine()*.

The matrix should be given either as a sequence of 9 values or a sequence of 3 sequences of 3 values.

Note: Needs documentation about the actual values of the matrix.

`matrix`

> **Note:** Needs documentation.

`_determinant``()`

> **Note:** Needs documentation.

`inverse``()`

> **Note:** Needs documentation.

`rotate``(`*degrees*, *clockwise=False*`)`

> **Note:** Needs documentation.

`scale``(`*x_factor*, *y_factor=None*`)`

> **Note:** Needs documentation.

`translate``(`*dx*, *dy*`)`

> **Note:** Needs documentation.

3.3.3 `pymaging.colors`

`pymaging.colors.`**`_mixin_alpha`**`(`*colors*, *alpha*`)`

Applies the given alpha value to all colors. Colors should be a list of three items: `r`, `g` and `b`.

class `pymaging.colors.`**`Color`**`(`*red*, *green*, *blue alpha*`)`

Represents a color. All four parameters should be integers between 0 and 255.

`red`

`green`

`blue`

`alpha`

classmethod `from_pixel`(*pixel*)
  Given a pixel (a list of colors), create a *`Color`* instance.

classmethod `from_hexcode`(*hexcode*)
  Given a hexcode (a string of 3, 4, 6 or 8 characters, optionally prefixed by `'#'`), construct a *`Color`* instance.

`get_for_brightness`(*brightness*)
  Given a brightness (alpha value) between 0 and 1, return the current color for that brightness.

`cover_with`(*cover_color*)
  Covers the current color with another color respecting their respective alpha values. If the `cover_color` is a solid color, return a copy of the `cover_color`. `cover_color` must be an instance of *`Color`*.

`to_pixel`(*pixelsize*)
  Returns this color as a pixel (list of integers) for the given `pixelsize` (3 or 4).

`to_hexcode`()
  Returns this color as RGBA hexcode. (Without leading `'#'`).

class `pymaging.colors.`**`ColorType`**
  A named tuple holding the length of a color type (pixelsize) and whether this color type supports the alpha channel or not.

  `length`

  `alpha`

`pymaging.colors.`**`RGB`**
  RGB *`ColorType`*.

`pymaging.colors.`**`RGBA`**
  RGBA *`ColorType`*.

3.3.4 `pymaging.exception`

exception `pymaging.exceptions.`**`PymagingExcpetion`**
  The root exception type for all exceptions defined in this module.

exception `pymaging.exceptions.`**`FormatNotSupported`**
  Raised if an image is saved or loaded in a format not supported by pymaging.

exception `pymaging.exceptions.`**`InvalidColor`**
  Raised if an invalid color is used on an image (usually when the image has a palette).

3.3.5 `pymaging.formats`

Loads and maintains the formats supported in this installation.

class `pymaging.formats.`**`Format`**(*open*, *save*, *extensions*)
  A named tuple that should be used to define formats for pymaging. `open` and `save` are callables that decode and encode an image in this format. `extensions` is a list of file extensions this image type could have.

  `open`

  `save`

  `extensions`

class `pymaging.formats.`**`FormatRegistry`**
  A singleton class for format registration

_populate()
: Populates the registry using package resources.

register(*format*)
: Manually registers a format, which must be an instance of *Format*.

get_format_objects()
: Returns all formats in this registry.

get_format(*format*)
: Given a format name (eg file extension), returns the *Format* instance if it's registered, otherwise `None`.

`pymaging.formats.`**registry**
: The singleton instance of *FormatRegistry*.

`pymaging.formats.`**get_format_objects**()
: Shortcut to `registry.get_format_objects`.

`pymaging.formats.`**get_format**()
: Shortcut to `registry.get_format`.

`pymaging.formats.`**register**()
: Shortcut to `registry.register`.

3.3.6 `pymaging.helpers`

`pymaging.helpers.`**get_transformed_dimensions**(*transform*, *box*)
: Takes an affine transform and a four-tuple of (x0, y0, x1, y1) coordinates. Transforms each corner of the given box, and returns the (width, height) of the transformed box.

3.3.7 `pymaging.pixelarray`

class `pymaging.pixelarray.`**GenericPixelArray**(*data*, *width*, *height*, *pixelsize*)
: The base pixel array class. `data` should be a flat `array.array` instance of pixel data, `width` and `height` are the dimensions of the array and `pixelsize` defines how many items in the `data` array define a single pixel.

  Use *get_pixel_array()* to instantiate this class!

  data
  : The image data as array.

  width
  : The width of the pixel array.

  height
  : The height of the pixel array.

  pixelsize
  : The size of a single pixel

  line_length
  : The length of a line. (*width* multiplied with *pixelsize*).

  size
  : The size of the pixel array.

  _precalculate()
  : Precalculates `line_width` and *size*. Should be called whenever *width*, *height* or *pixelsize* change.

`_translate`(*x*, *y*)
Translates the logical `x`/`y` coordinates into the start of the pixel in the pixel array.

`get`(*x*, *y*)
Returns the pixel at `x`/`y` as list of integers.

`set`(*x*, *y*, *pixel*)
Sets the `pixel` to `x`/`y`.

`copy_flipped_top_bottom`()
Returns a copy of this pixel array with the lines flipped from top to bottom.

`copy_flipped_left_right`()
Returns a copy of this pixel array with the lines flipped from left to right.

`copy`()
Returns a copy of this pixel array.

`remove_lines`(*offset*, *amount*)
Removes `amount` lines from this pixel array after `offset` (from the top).

`remove_columns`(*offset*, *amount*)
Removes `amount` columns from this pixel array after `offset` (from the left).

Note: If `remove_columns()` and `remove_lines()` are used together, `remove_lines()` should always be called first, as that method is a lot faster and `remove_columns()` gets faster the fewer lines there are in a pixel array.

`add_lines`(*offset*, *amount*, *fill=0*)
Adds `amount` lines to the pixel array after `offset` (from the top) and fills it with `fill`.

`add_columns`(*offset*, *amount*, *fill=0*)
Adds `amount` columns to the pixel array after `offset` (from the left) and fill it with `fill`.

Note: As with `remove_columns()`, the cost of this method grows with the amount of lines in the pixe array. If it is used together with `add_lines()`, `add_columns()` should be called first.

class `pymaging.pixelarray.`**`PixelArray1`**(*data*, *width*, *height*)
Subclass of `GenericPixelArray`, optimized for pixelsize 1.

Use `get_pixel_array()` to instantiate this class!

class `pymaging.pixelarray.`**`PixelArray2`**(*data*, *width*, *height*)
Subclass of `GenericPixelArray`, optimized for pixelsize 2.

Use `get_pixel_array()` to instantiate this class!

class `pymaging.pixelarray.`**`PixelArray3`**(*data*, *width*, *height*)
Subclass of `GenericPixelArray`, optimized for pixelsize 3.

Use `get_pixel_array()` to instantiate this class!

class `pymaging.pixelarray.`**`PixelArray4`**(*data*, *width*, *height*)
Subclass of `GenericPixelArray`, optimized for pixelsize 4.

Use `get_pixel_array()` to instantiate this class!

`pymaging.pixelarray.`**`get_pixel_array`**(*data*, *width*, *height*, *pixelsize*)
Returns the most optimal pixel array class for the given pixelsize. Use this function instead of instantating the pixel array classes directly.

3.3.8 `pymaging.resample`

class `pymaging.resample.`**`Resampler`**

Base class for resampler algorithms. Should never be instantated directly.

`affine`(*source*, *transform*, *resize_canvas=True*)

> **Note:** Document.

`resize`(*source*, *width*, *height*, *resize_canvas=True*)

> **Note:** Document.

class `pymaging.resample.`**`Nearest`**

Subclass of *`Resampler`*. Implements the nearest neighbor resampling algorithm which is very fast but creates very ugly resampling artifacts.

class `pymaging.resample.`**`Bilinear`**

Subclass of *`Resampler`* implementing the bilinear resampling algorithm, which produces much nicer results at the cost of computation time.

`pymaging.resample.`**`nearest`**

Singleton instance of the *`Nearest`* resampler.

`pymaging.resample.`**`bilinear`**

Singleton instance of the *`Bilinear`* resampler.

3.3.9 `pymaging.shapes`

Shapes are the high level drawing API used by *`pymaging.image.Image.draw()`*.

class `pymaging.shapes.`**`BaseShape`**

Dummy base class for shapes.

`iter_pixels`(*color*)

In subclasses, this is the API used by *`pymaging.image.Image.draw()`* to draw to an image. Should return an iterator that yields `x`, `y`, `color` tuples.

class `pymaging.shapes.`**`Pixel`**(*x*, *y*)

A simple single-pixel drawing object.

class `pymaging.shapes.`**`Line`**(*start_x*, *start_y*, *end_x*, *end_y*)

Simple line drawing algorithm using the Bresenham Line Algorithm. Draws non-anti-aliased lines, which is very fast but for lines that are not exactly horizontal or vertical, this produces rather ugly lines.

class `pymaging.shapes.`**`AntiAliasedLine`**(*start_x*, *start_y*, *end_x*, *end_y*)

Draws an anti-aliased line using Xiaolin Wu's line algorithm. This has a lot higher computation costs than *`Line`* but produces much nicer results. When used on an image with a palette, this shape might cause errors.

3.3.10 `pymaging.test_utils`

`pymaging.test_utils.`**`image_factory`**(*colors*, *alpha=True*)

Creates an image given a list of lists of `pymaging.color.Color` instances. The `alpha` parameter defines the pixel size of the image.

class `pymaging.test_utils.`**`PymagingBaseTestCase`**

`assertImage`(*image*, *colors*, *alpha=True*)
Checks that an image is the same as the dummy image given. `colors` and `alpha` are passed to *`image_factory()`* to create a comparison image.

3.3.11 `pymaging.utils`

`pymaging.utils.`**`fdiv`**(*a*, *b*)
Does a float division of `a` and `b` regardless of their type and returns a float.

`pymaging.utils.`**`get_test_file`**(*testfile*, *fname*)
Returns the full path to a file for a given test.

3.3.12 `pymaging.webcolors`

Defines constant `pymaging.color.Color` instances for web colors.

`pymaging.webcolors.`**`IndianRed`**

`pymaging.webcolors.`**`LightCoral`**

`pymaging.webcolors.`**`Salmon`**

`pymaging.webcolors.`**`DarkSalmon`**

`pymaging.webcolors.`**`LightSalmon`**

`pymaging.webcolors.`**`Red`**

`pymaging.webcolors.`**`Crimson`**

`pymaging.webcolors.`**`FireBrick`**

`pymaging.webcolors.`**`DarkRed`**

`pymaging.webcolors.`**`Pink`**

`pymaging.webcolors.`**`LightPink`**

`pymaging.webcolors.`**`HotPink`**

`pymaging.webcolors.`**`DeepPink`**

`pymaging.webcolors.`**`MediumVioletRed`**

`pymaging.webcolors.`**`PaleVioletRed`**

`pymaging.webcolors.`**`LightSalmon`**

`pymaging.webcolors.`**`Coral`**

`pymaging.webcolors.`**`Tomato`**

`pymaging.webcolors.`**`OrangeRed`**

`pymaging.webcolors.`**`DarkOrange`**

`pymaging.webcolors.`**`Orange`**

`pymaging.webcolors.`**`Gold`**

pymaging.webcolors.**Yellow**

pymaging.webcolors.**LightYellow**

pymaging.webcolors.**LemonChiffon**

pymaging.webcolors.**LightGoldenrodYellow**

pymaging.webcolors.**PapayaWhip**

pymaging.webcolors.**Moccasin**

pymaging.webcolors.**PeachPuff**

pymaging.webcolors.**PaleGoldenrod**

pymaging.webcolors.**Khaki**

pymaging.webcolors.**DarkKhaki**

pymaging.webcolors.**Lavender**

pymaging.webcolors.**Thistle**

pymaging.webcolors.**Plum**

pymaging.webcolors.**Violet**

pymaging.webcolors.**Orchid**

pymaging.webcolors.**Fuchsia**

pymaging.webcolors.**Magenta**

pymaging.webcolors.**MediumOrchid**

pymaging.webcolors.**MediumPurple**

pymaging.webcolors.**BlueViolet**

pymaging.webcolors.**DarkViolet**

pymaging.webcolors.**DarkOrchid**

pymaging.webcolors.**DarkMagenta**

pymaging.webcolors.**Purple**

pymaging.webcolors.**Indigo**

pymaging.webcolors.**DarkSlateBlue**

pymaging.webcolors.**SlateBlue**

pymaging.webcolors.**MediumSlateBlue**

pymaging.webcolors.**GreenYellow**

pymaging.webcolors.**Chartreuse**

pymaging.webcolors.**LawnGreen**

pymaging.webcolors.**Lime**

pymaging.webcolors.**LimeGreen**

pymaging.webcolors.**PaleGreen**

pymaging.webcolors.**LightGreen**

pymaging.webcolors.**MediumSpringGreen** pymaging.webcolors.**SpringGreen**

pymaging.webcolors.**MediumSeaGreen**

pymaging.webcolors.**SeaGreen**

pymaging.webcolors.**ForestGreen**

pymaging.webcolors.**Green**

pymaging.webcolors.**DarkGreen**

pymaging.webcolors.**YellowGreen**

pymaging.webcolors.**OliveDrab**

pymaging.webcolors.**Olive**

pymaging.webcolors.**DarkOliveGreen**

pymaging.webcolors.**MediumAquamarine**

pymaging.webcolors.**DarkSeaGreen**

pymaging.webcolors.**LightSeaGreen**

pymaging.webcolors.**DarkCyan**

pymaging.webcolors.**Teal**

pymaging.webcolors.**Aqua**

pymaging.webcolors.**Cyan**

pymaging.webcolors.**LightCyan**

pymaging.webcolors.**PaleTurquoise**

pymaging.webcolors.**Aquamarine**

pymaging.webcolors.**Turquoise**

pymaging.webcolors.**MediumTurquoise**

pymaging.webcolors.**DarkTurquoise**

pymaging.webcolors.**CadetBlue**

pymaging.webcolors.**SteelBlue**

pymaging.webcolors.**LightSteelBlue**

pymaging.webcolors.**PowderBlue**

pymaging.webcolors.**LightBlue**

pymaging.webcolors.**SkyBlue**

pymaging.webcolors.**LightSkyBlue**

pymaging.webcolors.**DeepSkyBlue**

pymaging.webcolors.**DodgerBlue**

pymaging.webcolors.**CornflowerBlue**

pymaging.webcolors.**RoyalBlue**

pymaging.webcolors.**Blue**

pymaging.webcolors.**MediumBlue**

pymaging.webcolors.**DarkBlue**

pymaging.webcolors.**Navy**

pymaging.webcolors.**MidnightBlue**

pymaging.webcolors.**Cornsilk**

pymaging.webcolors.**BlanchedAlmond**

pymaging.webcolors.**Bisque**

pymaging.webcolors.**NavajoWhite**

pymaging.webcolors.**Wheat**

pymaging.webcolors.**BurlyWood**

pymaging.webcolors.**Tan**

pymaging.webcolors.**RosyBrown**

pymaging.webcolors.**SandyBrown**

pymaging.webcolors.**Goldenrod**

pymaging.webcolors.**DarkGoldenrod**

pymaging.webcolors.**Peru**

pymaging.webcolors.**Chocolate**

pymaging.webcolors.**SaddleBrown**

pymaging.webcolors.**Sienna**

pymaging.webcolors.**Brown**

pymaging.webcolors.**Maroon**

pymaging.webcolors.**White**

pymaging.webcolors.**Snow**

pymaging.webcolors.**Honeydew**

pymaging.webcolors.**MintCream**

pymaging.webcolors.**Azure**

pymaging.webcolors.**AliceBlue**

pymaging.webcolors.**GhostWhite**

pymaging.webcolors.**WhiteSmoke**

pymaging.webcolors.**Seashell**

pymaging.webcolors.**Beige**

pymaging.webcolors.**OldLace**

pymaging.webcolors.**FloralWhite**

pymaging.webcolors.**Ivory**

pymaging.webcolors.**AntiqueWhite**

pymaging.webcolors.**Linen**

pymaging.webcolors.**LavenderBlush**

`pymaging.webcolors.`**`MistyRose`**

`pymaging.webcolors.`**`Gainsboro`**

`pymaging.webcolors.`**`LightGrey`**

`pymaging.webcolors.`**`Silver`**

`pymaging.webcolors.`**`DarkGray`**

`pymaging.webcolors.`**`Gray`**

`pymaging.webcolors.`**`DimGray`**

`pymaging.webcolors.`**`LightSlateGray`**

`pymaging.webcolors.`**`SlateGray`**

`pymaging.webcolors.`**`DarkSlateGray`**

`pymaging.webcolors.`**`Black`**

3.4 Writing formats

Formats in pymaging are represented as *`pymaging.formats.Format`* instances. To make your own format, create an instance of that class, giving a method to **decode**, a method to **encode** and a list of **extensions** for this format as arguments.

3.4.1 Decoder

The decoder function takes one file-like object as argument. It should return `None` if the file object passed in is not in the format handled by this decoder, otherwise it should return an instance of *`pymaging.image.Image`*. For help with image objects, see *About Image objects*.

3.4.2 Encoder

The encoder takes an instance of *`pymaging.image.Image`* and a file-like object as arguments and should save the image to that file object. For help with image objects, see *About Image objects*.

3.5 Contributing to Pymaging

3.5.1 Community

People interested in developing for the Pymaging should join the #pymaging IRC channel on freenode for help and to discuss the development.

3.5.2 Contributing Code

General

- Code **must** be tested. Untested patches will be declined.
- If a patch affects the public facing API, it must document these changes.

- If a patch changes code, the internal documentation (docs/dev/) must be updated to reflect the changes.

Since we're hosted on GitHub, pymaging uses git as a version control system.

If you're not familiar with git, check out the GitHub help page.

Syntax and conventions

We try to conform to PEP8 as much as possible. This means 4 space indentation.

Process

This is how you fix a bug or add a feature:

1. fork us on GitHub.
2. Checkout your fork.
3. Hack hack hack, test test test, commit commit commit, test again.
4. Push to your fork.
5. Open a pull request.

Tests

If you're unsure how to write tests, feel free to ask for help on IRC.

Running the tests

To run the tests we recommend using `nose`. If you have `nose` installed, just run `nosetests` in the root directory. If you don't, you can also use `python -m unittest discover`.

3.5.3 Contributing Documentation

The documentation is written using Sphinx/restructuredText.

Section style

We use Python documentation conventions fo section marking:

- `#` with overline, for parts
- `*` with overline, for chapters
- `=`, for sections
- `-`, for subsections
- `^`, for subsubsections
- `"`, for paragraphs

3.6 Indices and tables

- genindex
- modindex
- search

Python Module Index