# Pystacia Documentation

***Release 0.1***

**Paweł Piotr Przeradowski**

January 22, 2014

Contents

Pystacia is a new image manipulation library written to meet practical needs of developers. It's simple, it's cross-platform, runs on Python 2.5+, 3.x, *PyPy* and *IronPython*. It's compact but still appropriate for most of your day to day image handling tasks. Pystacia leverages powerful *ImageMagick* library as a back-end exposing easily comprehensible pythonic *API*.

Here is one of the simplest code snippets showing what you can do with it:

```
from pystacia.image import read


image = read('example.png')

image.rescale(320, 240)
image.rotate(30)
image.show()
image.write('output.jpeg')

# free acquired resources
image.close()
```

When saved to `simple.py`, the above script can be run via:

```
$ pip install pystacia
$ python simple.py
```

Provided you have file `example.png` in the same directory it would output version of the file which would be scaled to 320240 pixels and rotated by 30 degrees. It would also display it in your default system image viewing program.

CHAPTER 1

# What's new

## 1.1 What's new in Pystacia 0.2

This release focuses on thread-safety, reducing code boiler-plate, performance, increasing quality and test-coverage.

### 1.1.1 Architecture

The C API intermediate interface has been rewritten from scratch. It introduces a concept of a bridge that isolates code execution from several threads into so called "C-thread" that takes care of correct synchronization and ensures thread-safety. Imaging operations are still performed in parallel thanks to *OMP* inside *ImageMagick*. Actually this approach yields even better performance as *OMP* may have strange interactions with threads resulting in excessive context switching.

`pystacia.image.Image` and `pystacia.color.Color` inherit now from common base `pystacia.common.Resource` that encapsulate all the details of tracing instances, allocating, freeing memory at right time and preventing memory leaks.

Underlying C function prototypes and *ImageMagick* itself are loaded lazily on-demand resulting in faster startup times when importing modules. Also cross-module dependencies have been greatly reduced.

### 1.1.2 Enumeration and color handling simplification

From now on function calls that accept enumerations and color specifications can handle many formats without explicit casting. This approach reduces dependencies in your code and will be used in the documentation. Old way is still supported and in fact it gets auto-cast to it underneath.

```
>>> img1.overlay(img2, composite=composites.dst_over)  # old way
>>> img1.overlay(img2, composite='dst_over')  # new preferred way

>>> img1.get_pixel(0, 0) == from_rgb(1, 0, 0)  # old way
>>> img1.get_pixel(0, 0) == (1, 0, 0)  # new way
>>> img1.get_pixel(0, 0) == 'red'  # even simpler
>>> img1.get_pixel(0, 0) == 0xFF0000  # or like that
```

TODO: Link it More information can be found in Enum handling and Working with color chapters.

### 1.1.3 The registry

The registry is a concept introducing control over global Pystacia behavior. E.g. you can use it to set class used instead of `pystacia.image.Image` when instantiating image objects:

```
>>> class MyImage(Image):
>>>     def cool_stuff(self): pass
>>>
>>> from pystacia import registry
>>> registry.image_factory = MyImage
>>>
>>> blank(30, 30).__class__ == MyImage
True
```

TODO link: More on registry can be read in Controlling global behavior

### 1.1.4 New image handling functions

New image operations include but are not necessarily limited to:

- `pystacia.image.Image.fit()`
- `pystacia.image.Image.threshold()`
- `pystacia.image.Image.adpative_blur()`
- `pystacia.image.Image.adaptive_sharpen()`
- `pystacia.image.Image.add_noise()`
- `pystacia.image.Image.auto_level()`
- `pystacia.image.Image.auto_gamma()`
- `pystacia.image.Image.chop()`
- `pystacia.image.Image.charcoal()`
- `pystacia.image.Image.map()`
- `pystacia.image.Image.contrast_stretch()`
- `pystacia.image.Image.evaluate()`
- `pystacia.image.Image.total_colors()`
- `pystacia.image.Image.gaussian_blur()`
- `pystacia.image.Image.detect_edges()`
- `pystacia.image.Image.get_range()`
- `pystacia.image.Image.motion_blur()`
- `pystacia.image.Image.normalize()`
- `pystacia.image.Image.shade()`
- `pystacia.image.Image.sharpen()`
- `pystacia.image.Image.compare()`
- `pystacia.image.Image.is_same()`

### 1.1.5 Underlying ImageMagick

TODO: Bundled ImageMagick has been updated to version 6.3.7.X and built with

- libjpeg
- libtiff
- libpng
- libwebp
- libfftw
- libz

### 1.1.6 Color features

Color module gained support for `pystacia.color.from_hsl()`, `pystacia.color.from_int24()` and `pystacia.color.from_rgb8()` together with accompanying getters `pystacia.color.Color.get_hsl()`, `pystacia.color.Color.get_rgb8()`, `pystacia.color.Color.get_int24()`.

## 1.2 Migrating to 0.2

### 1.2.1 Deprecated symbols

Several symbols have been deprecated:

- `read`, `read_blob`, `read_raw`, `blank`, `checkerboard`, `lena`, `magick_logo`, `rose`, `wizard`, `granite`, `netscape`, `composites`, `types`, `filters`, `colorspaces`, `compressions`, `axes`, `Image` have been moved from `pystacia` to `pystacia.image`. Old imports will still worked and are proxied to corresponding imports in `pystacia.image` but they will be completely removed in 0.3.
- `pystacia.util.TinyException` is deprecated in favor of `pystacia.util.PystaciaException`. The old class is still in place but will be removed in 0.3.

There is easy way to detect if you are using one of those symbols. To get all warning information sent to *stdout* just run your script with `-W all` switch.

CHAPTER 2

# Front matter

## 2.1 Copyright, Trademarks, and Attributions

*The Pystacia image manipulation library, Version 0.2* by Paweł Piotr Przeradowski

No ponies were harmed during development of this software.



While the Pystacia documentation is offered under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License, the Pystacia *software* is offered under a less restrictive MIT license . It's completely free of charge both for open-source and commercial project use.

All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized. However, use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

Every effort has been made to make this documentation as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on as "as-is" basis. The author shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this documentation. No patent liability is assumed with respect to the use of the information contained herein.

This software relies on *ImageMagick* for its internal operation. *ImageMagick* is licensed under the ImageMagick License.

Logo image is a derivative work based on image by user `Ckim06` used in Wikipedia that can be found here: http://kk.wikipedia.org/wiki/%D0%A1%D1%83%D1%80%D0%B5%D1%82:PistachGrp_001_LR.jpg. Image is released into public domain.

General documentation structure and many paragraphs were adapted from Pyramid documentation by Chris McDonough.

Documentation theme is based on Flask theme by Armin Rochaster later adapted by Kenneth Reitz for *Requests* project.

CHAPTER 3

# Narrative documentation

## 3.1 Introduction

### 3.1.1 Overview

#### The philiosophy

##### Modules

Pacakge `pystacia` is divided into several submodules of which `pystacia.image` and `pystacia.color` are of most interest to readers. For convenience reasons symbols from *pystacia.image* are imported directly into `pystacia` and `pystacia.color` is accessible as `pystacia.color` attribute under `pystacia`.

For quick experiments in a console you can perform *from pystacia import **. Several attributes got pulled into your namespace including:

- `pystacia.image.read()`, `pystacia.image.read_blob()` and `pystacia.image.read_raw()` image factories
- `pystacia.image.composite`, `pystacia.image.types`, `pystacia.image.filters`, `pystacia.image.colorspaces`, `pystacia.image.axes` lazy enumerations
- `pystacia.lena()` and `pystacia.magick_logo()` sample image factories

##### Classes and factories

Pystacia is an object oriented imaging library. It uses factory functions to create objects representing concepts like `pystacia.image.Image` or `pystacia.color.Color`. You don't typically use class constructors to create objects and generally you shouldn't unless you really know what you are doing. You should rely on factories instead.

```
"""Create an image object from read factory"""

from pystacia import read
image = read('example.jpg')


"""Create a color object from from_string factory"""

from pystacia import color
red = color.from_string('red')
```

### Constants

Many Pystacia methods take *C* enum-like mnemonics. For example to specify which axis to perform transformation along you can use `axes.x`. These names are symbolic representation of underlying *C* constants. They are lazily resolved during runtime to their integral value.

```
"""Skew an image by 5 pixels along Y axis"""

image.skew(5, axes.y)
```

There are several lazy enums defined. Most of them inside `pystacia.image` but they are also for you convenience imported into main `pystacia` module. Some of them are listed below:

- `pystacia.image.types`: image types such as `types.bilevel` for monochrome image, *types.pallette*, *types.grayscale* and *types.truecolor*
- `pystacia.image.colorspace`: color-spaces such as `colorspaces.rgb` or *colorspaces.ycbcr*.
- `pystacia.image.filters`: sampling filters used typically in rescaling algorithms including popular `filters.point`, `filters.bilinear` or `filters.sinc` typically used in `pystacia.image.Image.rescale()`.
- `pystacia.image.composites`: `composite.over` or `composite.hue` used with `pystacia.image.Image.overlay()`.
- `pystacia.image.axes`: `axes.x` and `axes.y` axes

### Method chaining

By default methods of `pystacia.image.Image` are chainable i.e. you can construct code using long string of methods forming a chain such as:

```
from pystacia import read
read('example.jpg').denoise().rescale(256, 256).rotate(45).write('output.png')
```

This style of programming is used a lot in some communities e.g. *jQuery* and some *Java* and *PHP* projects. This is unusual in Python and not entirely clear if appropriate. Instead a Python programmer could typically code like that:

```
from pystacia import read
image = read('example.jpg')
image.denoise()
image.rescale(256, 256)
image.rotate(45)
image.write('output.png')
```

Pystacia allows both styles or mixture of them. By default all methods that can be chained are chainable. It's up to you what you choose. One of the down sides of chaining is that when an exception occurs it can be not immediately clear where it comes from when you call one method several times on one line. If you want to explicitly forbid chaining you can do so by injection environment variable *PYSTACIA_NO_CHAINS* with non-false value before importing `pystacia`. When you do so methods that were previously chainable return `None`:

```
from os import environ
environ['PYSTACIA_NO_CHAINS'] = '1'

from pystacia import read
# chaining explicitely disabled above

image = read('example.jpg')
image.blur(3).rotate(45)  # this raises an Exception
```

or from shell:

```
$ PYSTACIA_NO_CHAINS=1 python helloworld.py
```

#### Behind the scenes

Pystcia uses *ImageMagick DLL* to perform its operation. Specifically *MagickWand* API is used which is contained in *libMagickWand.so*, *libMagickWand.dylib* or *libMagickWand.dll* depending on the platform used. Pystacia searches for the library in several places starting from the place where bundled binaries are normally stored and ending with system-wide locations. The details of search algorithms are detailed in *Skipping binary install*. Resolved library is loaded through *ctypes* and all Pystacia API calls are translated into their several C API low-level counterparts abstracting details for you. Pystacia can work with *ImageMagick* version 6.5.9.0 or later but more recent versions are bundled and advised to use.

### 3.1.2 Motivation

#### Why another Python imaging library

There exist quite a bunch of other Python imaging solutions with *PIL* being the most prominent one. Most of them don't satisfy at least one and typically more of the criteria that I consider quality and modern Python software.

- They are hardly documented or not at all
- They don't have complete test suite with reasonable coverage
- They don't work with one of: Python 2.5, Python 2.6, Python 2.7, Python 3.2, *PyPy* or *IronPython*
- They can't run processing algorithms on many cores in parallel leaving them idle.
- They are limited to narrow scope of image formats
- They don't allow operation on *RAW* data formats such as *YCbCr* or *RGB* triplets
- They can only process images internally in 8bit depth per channel what can be not sufficient for some operations
- They are not actively maintained anymore or largely out-dated
- They don't provide Pythonic API because they are modeled after *C* code
- They provide interfaces that are not easily extensible e.g. don't allow subclassing
- They are not easily installed on widely used OSes such as *Windows*, *Linux* and *MacOS X*
- They always require *C* compiler during installation phase
- They don't have transparent open-source development cycle
- They try to hide complexity of image processing preventing you from performing custom advance operations
- You can find a lot of ranting on the Internet and they drive you crazy

#### Why another *ImageMagick* wrapper

While you can find a lot of *ImageMagick* wrappers most of them feel like doing *C++* in Python. Some of them are not maintained and typically don't work on one of supported Python runtime + platform combinations.

#### What Pystacia does about that

- Pystacia has documentation that I try to make as complete as my resources let me
- Pystacia has large test suite consisting of ~70 tests with 90% coverage at the time of writing.
- Pystacia works both on Python 2.5+ and Python 3.x as well as *PyPy* and *IronPython*. *Jython* support is planned through *JNA* interface or *ctypes* if it gets reasonably stable and complete by the time of implementation.
- Pystacia ships version of *ImageMagick* built with *OMP* support which lets you use all your cores for image processing.
- Pystacia comes with wide spectrum of supported formats in standard distribution. *JPEG*, *PNG*, *TIFF*, *GIF*, *BMP*, *ICO*, *JNG*, *PCX*, *PNM*, *HDR*, *EXR* to name a few. See http://www.imagemagick.org/script/formats.php for complete list.
- Pystacia can read and write *RAW* data blobs such as *RGB*, *YCbCr*, *YUV* and *CMYK* in both 8 and 16 bits per channel with or without alpha channel directly from Python strings or streams.
- The default *ImageMagick* distribution coming with Pystacia processes data in 16 bit precision internally.
- I did put a lot of effort into making this code as good as possible. All the Python code is continously tested against PEP8 validity and inspected with *Pyflakes* to detect common problems.
- Pystacia tries to hide *ImageMagick* quirks, shield you from ABI changes, provide monolithic Python API. It supports any version of Imagick that is newer or equal to 6.5.9.0. Pystacia itself comes with reasonable recent versions of *ImageMagick*.
- Pystacia strives to provide Pythonic API employing as many idioms and common patterns as possible. Pystacia classes are fully subclassable and factory methods can accept factory parameters which specify which class is to be used.
- Pystacia comes with prebuild *ImageMagick* binaries for Windows, Linux and MacOS X. Still it's possible to build it yourself if the default distribution doesn't fit your needs.
- Pystacia is completely free of charge both for open-source and commercial uses as it's licensed under MIT license.
- Pystacia exposes wand property on objects so you can use raw *ctypes* calls if you really want to but normally it shouldn't be necessary.

## 3.2 Installation

### 3.2.1 Before you install

You will need Python version 2.5 or better or Python 3.x to use Pystacia.

If you don't know how to install Python on your machine please refer to appropriate section in Pyramid documentation where it has been elaborated.

Pystacia is also known to work with *PyPy* and *IronPython* 2.7.1. Instructions for running on *PyPy* don't differ from those for *cPython*. *IronPython* doesn't support *setuptools* yet so installation must be performed manually by downloading and unpacking several packages. Refer to *Installing tinying with IronPython on Windows and .NET* for more details.

### 3.2.2 Installing Pystacia with cPython or PyPy on Windows, Linux and MacOS

Before proceeding on Windows ensure that Python binary is on your system search path. When you type *python* in console you should be greeted with python interactive console.

It is best practice to install develop your applicaion in a "virtual" Python environment in order to obtain isolation from any "system" packages you've got installed in your Python version. This can be done by using the *virtualenv* package. Using a virtualenv will also prevent Pystacia from globally installing versions of packages that are not compatible with your system Python.

Let's assume that you have following folder structure:

```
/workspace
     /my-project
          helloworld.py
```

Workspace folder is typically a common ancestor folder where you put your Python projects. My project folder contains all the files related to your poject.

To set up virtualenv you need to grab its latest version from https://github.com/pypa/virtualenv/tags. At the time of writing version 1.6.4 was the most up to date one. Download it to Workspace folder, unpack, delete original ZIP file and rename unpacked folder to virtualenv.

We're going to create your virtual environment which contains all the packages that your project needs to install in a way that they are separated from system-wide packages. Open console program, *cd* into Workspace directory and type following command:

```
$ python virtualenv/virtualenv.py --no-site-packages my-project/my-project-env
```

---

**Note:** On Windows change forward slashes to backslashes. If you use *PyPy* instead of *cPython* substitute *python* for *pypy* binary name.

---

This will create my-project/my-project-env with virtual environment in it. You can put your virtual environment whereever you want but typically you do it under a subfolder inside your project folder to keep things together.

Now the virtualenv is created you need to activate it so all the feature commands and installations will be performed in a virtualenv instead of being applied system-wide. To do so on Linux and Mac perform following command:

```
$ cd my-project
$ source my-project-env/bin/activate
```

and on Windows:

```
$ cd my-project
$ my-project-env/Scripts/activate.bat
```

After completing your shell prompt should include my-project-env environment name in it. You can now install Pystacia inside your virtual environment with *pip*.

```
$ pip install pystacia
```

You can test your installation by performing following action:

```
$ python
Python 2.7.1 (dcae7aed462b, Aug 17 2011, 09:46:15)
[PyPy 1.6.0 with GCC 4.0.1] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pystacia import lena
>>> lena().show()
```

It should display nice standard test image depicting Lena Söderberg, the first Lady of the Internet.

### 3.2.3 Installing tinying with IronPython on Windows and .NET

Installing Pystacia on IronPython is manual since at the time of writing IronPython couldn't handle *virtualenv*, *pip* or :term: *setuptools* properly. Though it's completely functional with a little bit of effort.

First obtain latest version of IronPython from http://ironpython.net/. At the time of writing it was 2.7.1. Perform standard installation procedure.

Create your sandbox folder where you install all the needed packages and perform testing. Let's say it's C:sandbox.

> C:sandbox

We need to manually satisfy all the dependencies. First grab :term:six library from http://pypi.python.org/pypi/six. Download, unpack and copy *six.py* file into your sandbox folder. Then go to http://pypi.python.org/pypi/decorator grab source distribution unpack it and grab file *decorator.py* from src subfolder directly into your sandbox folder. Now your sandbox folder should look like this:

> **C:sandbox** decorator.py six.py

Now it's time to install Pystacia itself. Go to https://bitbucket.org/liquibits/pystacia/downloads and grab Pystacia source distribution, unpack it and put folder Pystacia under your sandbox folder. You also need a binary image distribution for your Windows. It can be grabbed from the same URL. Remember to choose correct version for your architecture (32 bit or 64 bit). Unpack it and move all the files into *cdll* subdirectory under *pystacia* folder. Your installation should look like this now:

```
C:\sandbox
    decorator.py
    six.py
    pystacia\
        *some files here*
        cdll\
            *ImageMagick dlls here*
```

You are almost done. Open your console program and type:

```
cd c:\sandbox
ipy.exe -X:Frames
>>> from pystacia import *
```

If it succeeds everything is configured propertly. Note that we assmued that *ipy.exe* is on your system path - otherwise you need to type full path to it Also -X:Frames switch is mandatory since *IronPython* otherwise doesn't provide `sys._getframe()` which is referenced by *decorator* and *six* libraries.

### 3.2.4 What gets installed

Pystacia relies on *six* library to ship one source code both for Python 2.x and Python 3.x. It also heavily uses *decorator* library to make decorators easily documented and accessible with help in Python console. For testing on Python 2.6 and lower it pulls in *unittest2* library which is a backport of Python 3.x testing library. On Python 2.5 it also need *StringFormat* library as a polyfill for missing `str.format()` method.

### 3.2.5 I want to run a test suite

Test suite can be run by entering the *pystacia* folder inside site-packages folder and running *nosetests* after install *nose* package.

## 3.3 Working with images

The `pystacia.image.Image` is a central concept to the whole library. Though you typically don't use its constructor directly and rely on factories. All the functionality is implemented in `pystacia.image` module but for convenience reasons attributes are imported to main `pystacia` module from where you would import it.

### 3.3.1 Reading and writing

#### Files, streams and byte strings

To read a file from the disk you would use `pystacia.image.read()` factory:

```
from pystacia import read

image = read('example.jpg')
```

This reads file *example.jpg* from current working directory and returns `pystacia.image.Image` instance. To write it back to disk under different name and format you could write:

```
image.write('output.png')
```

This would save it back to *output.png* file with format *PNG*. Format is determined from the file extension.

Sometimes instead of having your data stored in a file you already have it in byte string or stream (file-like object with `read()` method). In such case you use `pystacia.image.read_blob()` instead:

```
from pystacia import read_blob

# data is byte string or stream in e.g. PNG format
image = read_blob(data)
```

Pystacia can also deal with *RAW* uncompressed data. You use `pystacia.image.read_raw()` method in such cases. Note that you need to explicitly specify *RAW* format, width, height and depth per channel as it's not carried along data itself.

```
from pystacia import read_raw

# create 2x2 pixel RGB image with red, green, blue and black pixel
# from RGB triplets in 8 bit depth
data = [255, 0, 0, 0, 255, 0, 0, 0, 255, 0, 0, 0]
data = bytes(data) # python 3k, ''.join(chr(x) for x in data) in 2.x

image = read_raw(data, 'rgb', 2, 2, 8)
```

Sometimes you may also want to dump image object contents to byte string instead of saving in to this. You use get_blob method to do so. To dump image contents as PNG blob you would call:

```
image.get_blob('png')
```

To get *RAW* data along its format, dimensions, and depth as dictionary analogical to parameters you would pass to `pystacia.image.read_raw()` you can call `pystacia.image.get_raw()` passing it color space:

```
image.get_raw('ycbcr')
```

### Generic images

Instead of reading an image from a file or stream you sometimes may want to start from blank image or well defined pattern.

Use `pystacia.image.blank()` to create empty blank image of given dimensions. By default it fills it with transparent pixels, but a third parameter specifying color can be used.

```
>>> from pystacia import blank, color
>>> blank(100, 100)
>>> blank(100, 100, color.from_string('red'))
>>> blank(100, 100, color.from_rgba(0, 1, 0, 0.5))
```

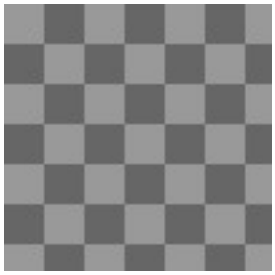


Figure 3.1: Transparent (default)

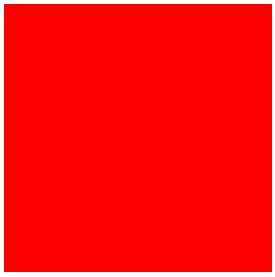


Figure 3.2: Red

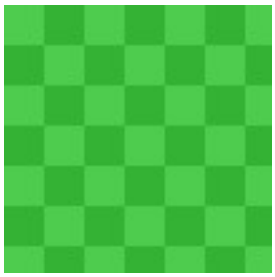


Figure 3.3: Translucent green

You can also generate a checkerboard pattern which is used in this documentation to mark transparent pixels with `pystacia.image.checkerboard()` which accepts width and height.

```
>>> from pystacia import checkerboard
>>> checkerboard(200, 200)
```

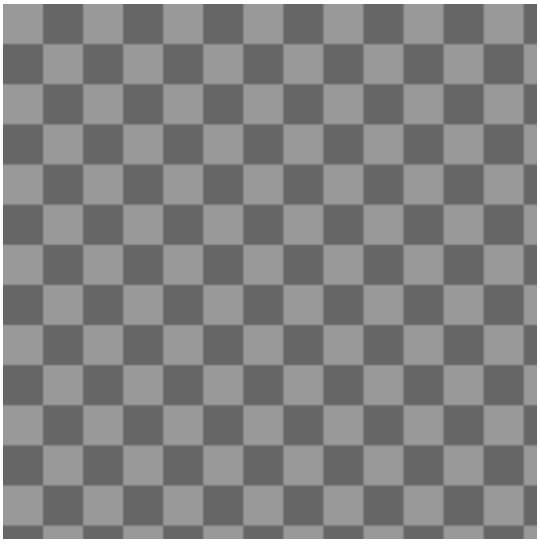


Figure 3.4: Checkerboard pattern

### 3.3.2 Common properties

#### Dimensions

All images have some common properties like dimensions, color space, type and depth. To get dimensions of the image you can access `pystacia.Image.size` property to get `(width, height)` tuple or `pystacia.Image.width` and `pystacia.Image.height` separately:

```
>>> image.size
(640, 480)
>>> image.width
640
>>> image.height
480
```

#### Color space

Color space represents combination of channels that image is internally stored in. You can query it with `pystacia.Image.colorspace` property. It yields *pystacia.image.colorspaces.rgb* for most images but other values are also possible.

```
>>> image.colorspace
pystacia.lazyenum.enum('colorspace').rgb
```

You can also assign to this property. It results in reinterpretation of stored color space i.e. if the original image was in *RGB* color space assigning it *YCbCr* would result in treating *Red channel* as *Luma* and *Green* and *Blue* information as *Chroma components* which yields strange visual effects.

Note that *RGB* image reinterpreted as *CMY* is simply negative since *CMY* is subtractive model complementing *RGB* i.e. each channel value is inversion of its counterpart.

If you want to change (convert) underlying color space without affecting visual representation use `pystacia.image.Image.convert_colorspace()` method instead.

Figure 3.5: Original *RGB* image

Figure 3.6: Reinterpreted as *YCbCr*

Figure 3.7: Reinterpreted as *CMY*

### Depth

Depth represents number of bits used to store channel information. It's typically 8 bit for *TrueColor* images but can be as well 16 bit for some *TIFF* images. You query the image depth with `pystacia.image.Image.depth` property.

```
>>> image.depth
8
```

### Storage type

Another aspect of image storing is a type. Type of image relates to how the values stored in memory are mapped into color values on the screen. Sometimes it's a direct mapping like *TrueColor* where values stored in :term: *RGB* triplets directly encode their visual representation. Another popular type are paletted (indexed) images where image consist of abstract values (typically 0 to 255) that are translated into final color value through a lookup table (*palette*). This has been popularized with *GIF* format. *Grayscale* image is an image storing only luminosity information (also typically in one byte). It can be also taught as a indexed image with implied palette which maps each luminosity (l) value into an *RGB* triplet (l, l, l). Finally bilevel image is an image consisting of two colors - typically black and white stored in one bit per pixel.

Each of types mentioned above also has its *matte* counterpart i.e. one that is accompanied by alpha channel. These have additional `_matte` suffix.

You can read and set types with `pystacia.Image.type` property. Setting a type which loses color information relative to original results in automatic *dithering*:

```
>>> image.type
pystacia.lazyenum.enum('type').truecolor
```

```
>>> image.type = types.palette

>>> image.type = types.grayscale

>>> image.type = types.bilevel
```

Here are close-ups of resulting images:

Figure 3.8: *TrueColor* image

Figure 3.9: Converted to *pallette*

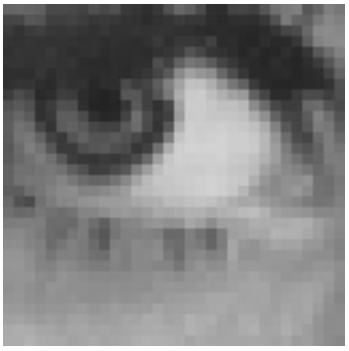


Figure 3.10: Coverted to *grayscale*

Figure 3.11: Coverted to *bilevel*

### 3.3.3 Geometry transformation

#### Rescaling

Rescaling is an operation of changing size of original image that preserves all the original visual characteristics in the new viewport. Rescaling can be both proportional and not proportional. You typically perform this operation by suppling width and height into `pystacia.image.Image.rescale()`:

```
>>> image.size
(256, 256)
>>> image.rescale(300, 200)
>>> image.size
(300, 200)
>>> image.rescale(128, 128)
>>> image.size
(128, 128)
```

Figure 3.12: Original

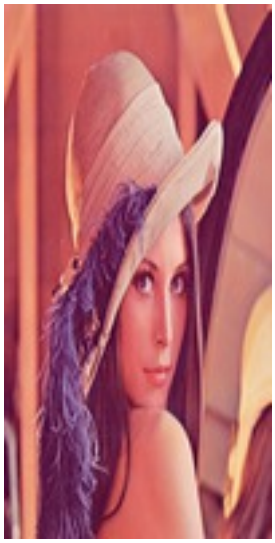


Figure 3.13: (100, 200)

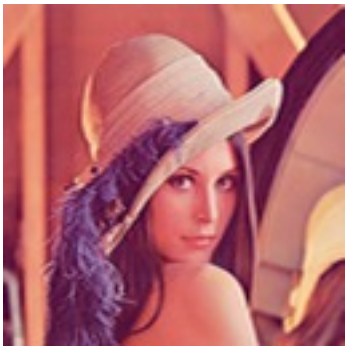


Figure 3.14: (128, 128)

Alternatively you can pass factor into method. This specifies how many times the original sizes are multiplied. If you pass single number the scaling will be proportional in both dimensions. You can also pass a two-element `tuple`.

```
>>> image.rescale(factor=0.75)
>>> image.rescale(factor=(0.6, 0.5))
>>> image.rescale(factor=(1.3, 1))
```

Figure 3.15: Factor 0.75

Note that this way resulting size is calculated relatively to previous size.

Another interesting aspect of resizing is resize filter. This affects the sharpness or smoothness and quality of rescaled image. Typically used filters include *point* (also known as nearest neighbor), *cubic*, *sinc* or *lanczos*.

```
>>> image.rescale(factor=2, filter=filters.point)

>>> image.rescale(factor=2, filter=filters.cubic)

>>> image.rescale(factor=2, filter=filters.sinc)
```

Figure 3.16: Factor (0.6, 0.5)

Figure 3.17: Factor (1.3, 1)

```
>>> image.rescale(factor=2, filter=filters.lanczos)
```

Upscaling close-ups with differnt filters:

Figure 3.18: Point

```
>>> image.rescale(factor=0.5, filter=filters.point)

>>> image.rescale(factor=0.5, filter=filters.cubic)

>>> image.rescale(factor=0.5, filter=filters.sinc)

>>> image.rescale(factor=0.5, filter=filters.lanczos)
```

Downscaling close-ups with different filters:

### Resizing

If you wanna crop out a portion of an image you can use `pystacia.image.Image.resize()`. it accepts four parameters describing cropped out region: width, height, x and y in this order. The latter two default to 0:

```
>>> image.size
(256, 256)

>>> image.resize(128, 128)

>>> image.resize(64, 128, 128, 128)
```

### Rotating

You can rotate an image with `pystacia.image.Image.rotate()` method. Angle is mesaured in degrees. Positive angles yield clockwise rotation while negative ones counter-clockwise. The resulting empty spaces are filled with transparent pixels.

Figure 3.19: Cubic

Figure 3.20: Sinc

Figure 3.21: Lanczos

Figure 3.22: Point

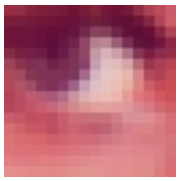


Figure 3.23: Cubic

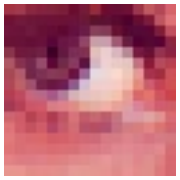


Figure 3.24: Sinc

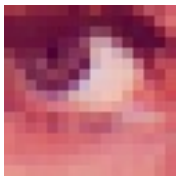


Figure 3.25: Lanczos

Figure 3.26: Original

Figure 3.27: (128, 128)

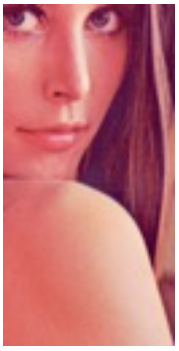


Figure 3.28: (64, 128, 128, 128)

```
>>> image.rotate(30)

>>> image.rotate(90)

>>> image.rotate(-45)
```

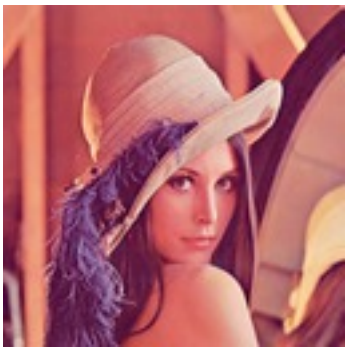


Figure 3.29: Original

### Flipping

Use `pystacia.image.Image.flip()` to flip (mirror) image around X or Y axis. Use `pystacia.image.Image.axes` enumeration to specify axis.

Figure 3.30: 30°

Figure 3.31: 90°

Figure 3.32: -45°

```
>>> image.flip(axes.x)

>>> image.flip(axes.y)
```

Figure 3.33: Original

### Transposing and transversing

Use `pystacia.image.Image.transpose()` and `pystacia.image.Image.transverse()` to transpose or transverse an image. Transposing creates a vertical mirror image by reflecting the pixels around the central x-axis while rotating them 90-degrees. Transversing creates a horizontal mirror image by reflecting the pixels around the central y-axis while rotating them 270-degrees.

#### Skewing

Skewing is the action of pushing one of the edges of an image along X or Y axis. You can perform it with `pystacia.image.Image.skew()` passing offset in pixels and desired axis.

```
>>> image.skew(10, axes.x)

>>> image.skew(-5, axes.x)

>>> image.skew(20, axes.y)
```

### Rolling

Rolling in an action of offsetting an image and filling empty space with pixels that overflew on the edge. It can be performed with `pystacia.image.Image.roll()` method. It accepts offses in X and Y directions as arguments.

Figure 3.34: Mirror X

Figure 3.35: Mirror Y

Figure 3.36: Original

Figure 3.37: Transposed

Figure 3.38: Transversed

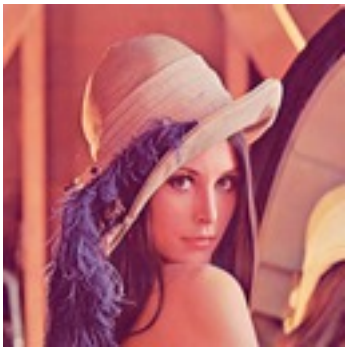


Figure 3.39: Original

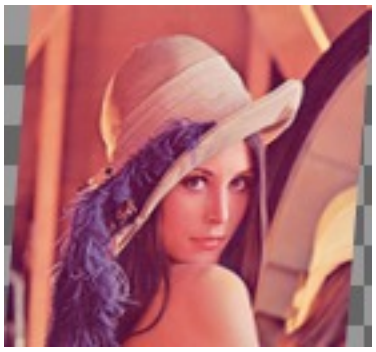


Figure 3.40: 10 pixels along X

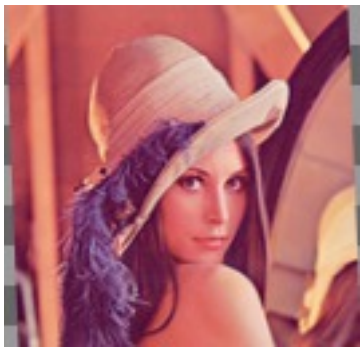


Figure 3.41: -5 pixels along X

Figure 3.42: 20 pixels along Y

```
>>> image.roll(100, 0)

>>> image.roll(-30, 40)
```

Figure 3.43: Original

#### Straightening image

Sometimes you have an image that is not straightened since It could be scanned so. You can use `pystacia.image.Image.straighten()` to correct that. It accepts single parameter - threshold which tells Pystacia what is the difference between baackground and subject.

```
>>> image.straighten(20)
```

#### Trimming extra background

If your image has extra background around it you can trim it off with `pystacia.image.Image.trim()` method. It accepts two optional parameters similarity and background color of space to discard (defaults to transparent).

```
>>> image.trim()
```

### 3.3.4 Color transformation

Color transformations are operations that affect color channel information without changing pixel location in any way.

Figure 3.44: Rolled by (100, 0)

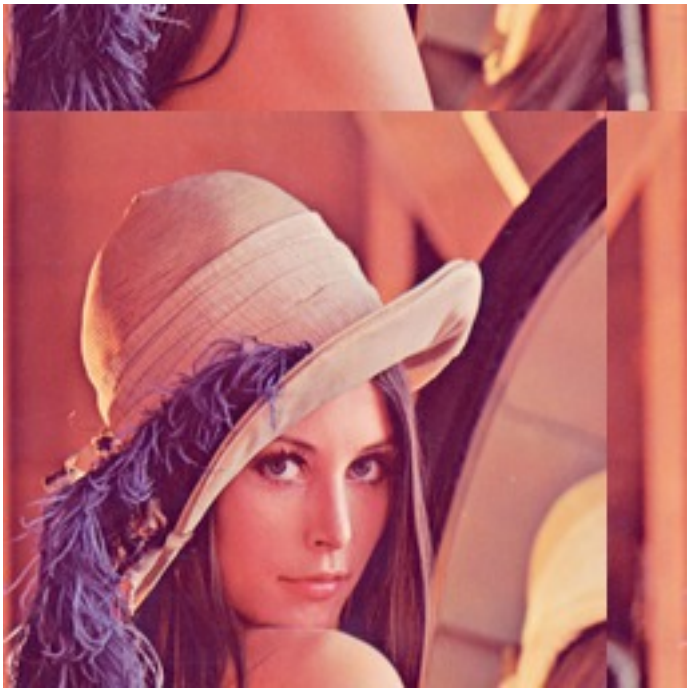


Figure 3.45: Rolled by (-30, 40)


Figure 3.46: Rotated image


Figure 3.47: Straightened up

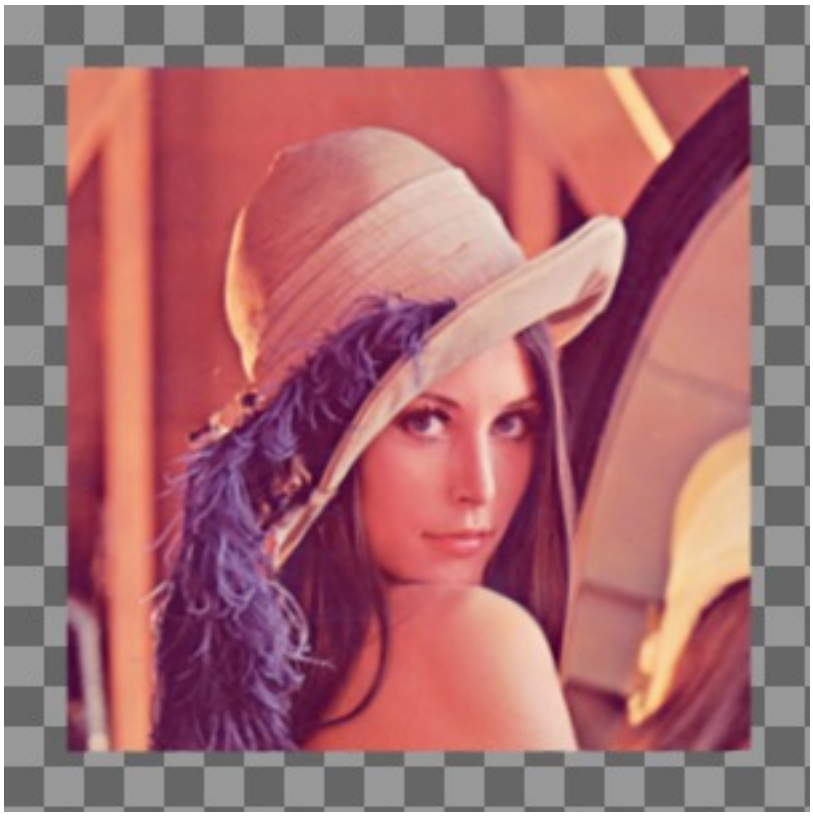


Figure 3.48: Image with empty space

Figure 3.49: Trimmed off

### Adjsuting contrast

`pystacia.image.Image.contrast()` increases or decreases contrast of an image. Passing *0* is no change operation. Values towards *-1* decrease constract whilst values towards *1* increase it.

```
>>> image.contrast(-1)

>>> image.contrast(-0.6)

>>> image.contrast(-0.25)

>>> image.contrast(0)

>>> image.contrast(0.25)

>>> image.contrast(0.75)

>>> image.contrast(1)
```

### Adjusting brightness

`pystacia.image.Image.brightness()` adjusts the brightness of an image. Value *0* is no-change operation. Values towards *-1* make image darker whilst calues towards *1* increase brightness.

```
>>> image.brightness(-1)

>>> image.brightness(-0.6)
```

Figure 3.50: -1

Figure 3.51: -0.6

Figure 3.52: -0.25

Figure 3.53: 0 (original)

Figure 3.54: +0.25

Figure 3.55: +1

```
>>> image.brightness(-0.25)

>>> image.brightness(0)

>>> image.brightness(0.25)

>>> image.brightness(0.75)

>>> image.brightness(1)
```

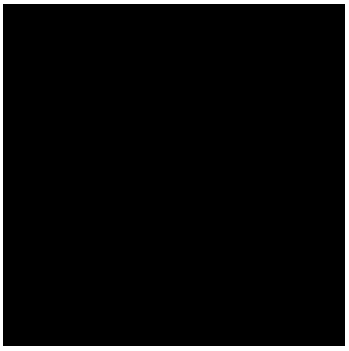


Figure 3.56: -1

Figure 3.57: -0.6

Figure 3.58: -0.25

### Gamma correction

You can use `pystacia.image.Image.gamma()` to apply gamma correction. Value of *1* is no-change operation. Values towards *0* make image darker. Values towards infinity make image lighter.

```
>>> image.gamma(0.3)

>>> image.gamma(0.6)

>>> image.gamma(1)

>>> image.gamma(1.5)

>>> image.gamma(2)
```

### Modulation

Modulation is an operation of adjusting hue, saturation and luminance of an image. It can be accomplished with `pystacia.image.Image.modulate()`. It accepts parameters in hue, saturation and luminance order. They all default to 0 meaning no change. Usable hue values start from -1 meaning rotation of hue by -180 degrees to 1 meaning +180 degrees. Saturation values towards *-1* desaturte image whilst values towards infinity saturate it. Setting luminosity to *-1* yields completely black image whilst values towards infinity make it brighter.

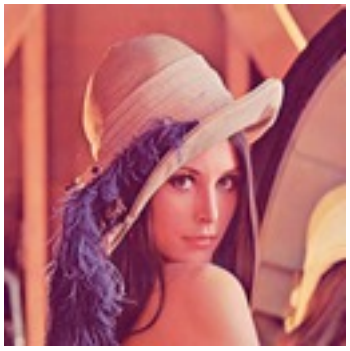


Figure 3.59: 0 (original)

Figure 3.60: +0.25

Figure 3.61: +0.75

Figure 3.62: 0.1

Figure 3.63: 0.3

Figure 3.64: 0.6

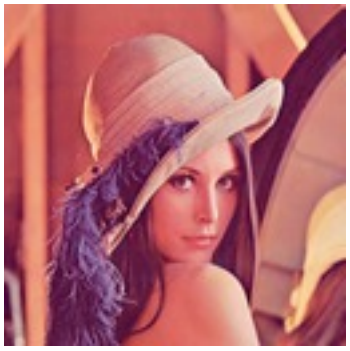


Figure 3.65: 1 (Original)

Figure 3.66: 1.5

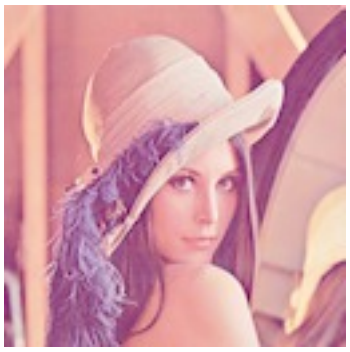


Figure 3.67: 2

```
>>> image.modulate(-1, -0.25, 0.1)

>>> image.modulate(-0.5, 0.25, 0)

>>> image.modulate(-0.2, 0.5, -0.25)

>>> image.modulate(0, 0, 0)

>>> image.modulate(0.4, -0.5, 0)

>>> image.modulate(0.8, 0, 0)
```

Figure 3.68: (-1, -0.25, 0.1)

Figure 3.69: (-0.5, 0.25, 0)

Figure 3.70: (-0.2, 0.5, -0.25)

### Desaturation

You can peform desaturation with `pystacia.image.Image.desaturate()`. It is a shortcut to `pystacia.image.Image.modulate()` passing *-1* as saturation.

```
>>> image.desaturate()
```

### Colorization

Colorization in an action of replacing all hue values in an image with a hue from a given color. `pystacia.image.Image.colorize()` accepting single color parameter performs it.

```
>>> image.colorize(color.from_string('red'))

>>> image.colorize(color.from_string('yellow'))

>>> image.colorize(color.from_string('blue'))

>>> image.colorize(color.from_string('violet'))

>>> image.colorize(color.from_string('green'))
```

### Sepia tone

`pystacia.image.Image.sepia()` performs effect similar to old-fashined sepia image. You can adjust hue and saturation parameters but the default values are a good starting point.

```
>>> image.sepia()
```

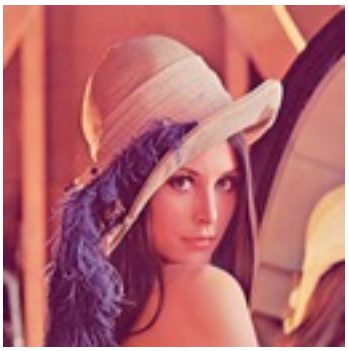


Figure 3.71: (0, 0, 0) Original

Figure 3.72: (0.4, -0.5, 0)

Figure 3.73: (0.8, 0, 0)

Figure 3.74: Original

Figure 3.75: Desatured

Figure 3.76: Original

Figure 3.77: red

Figure 3.78: yellow

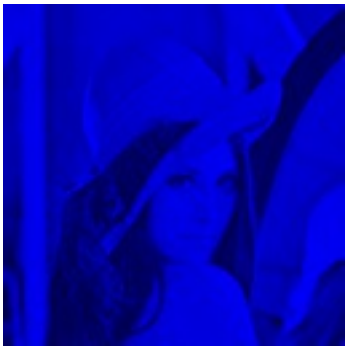


Figure 3.79: blue

Figure 3.80: violet

Figure 3.81: Original

Figure 3.82: Sepia tonning

### Equalization

`pystacia.image.Image.equalize()` is a method of streching channel information to fill full avaiable spectrum. It can result in drastic color quality improvement on low contrast, tainted images.

```
>>> image.equalize()
```

Figure 3.83: Original

### Invertion

Invertion is a process of substractin original channel value from it's maximum value. It results in a negative and can be performed with `pystacia.image.Image.invert()`.

```
>>> image.negative()
```

### Solarization

Solarization leads to effect similar of partly exposing an image in a darkroom. It can be performed with `pystacia.image.Image.solarize()`. It accepts single parameter - factor. Factor *0* is no change operation, Factor *1* is exactly the same as negative of original. Value of *0.5* yields particulary interesting effects.

```
>>> image.solarize(0)

>>> image.solarize(0.5)

>>> image.solarize(1)
```

Figure 3.84: Equalized image

Figure 3.85: Original

Figure 3.86: Inverted image

Figure 3.87: 0 (Original)

Figure 3.88: 0.5

Figure 3.89: 1 (Inverted original)

### Posterization

`pystacia.image.Image.posterize()` accepts single level parameter and reduces number of colors in the image to `levels ** 3` colors. Each channel has level final values distributed equally along its spectrum. So 1 level yields 1 color, 2 levels yield 8 color and so on.

```
>>> image.posterize(2)

>>> image.posterize(3)

>>> image.posterize(4)

>>> image.posterize(5)
```

Figure 3.90: Original

Figure 3.91: 2 levels

Figure 3.92: 3 levels

Figure 3.93: 4 levels

Figure 3.94: 5 levels

### 3.3.5 Bluring, denoising and enhancing

#### Blur

You can blur image with `pystacia.image.Image.blur()`. Method accepts mandatory radius and optional strength parameter.

```
>>> img.blur(3)
```

```
>>> img.blur(10)
```

Figure 3.95: Original

#### Radial blur

To perform radial blur use `pystacia.image.Image.radial_blur()`. Pass in single parameter - blur angle in degrees.

```
>>> img.blur(10)
```

```
>>> img.blur(45)
```

#### Removing noise

If you want to perform noise removal you can use `pystacia.image.Image.denoise()` method.

```
>>> img.denoise()
```

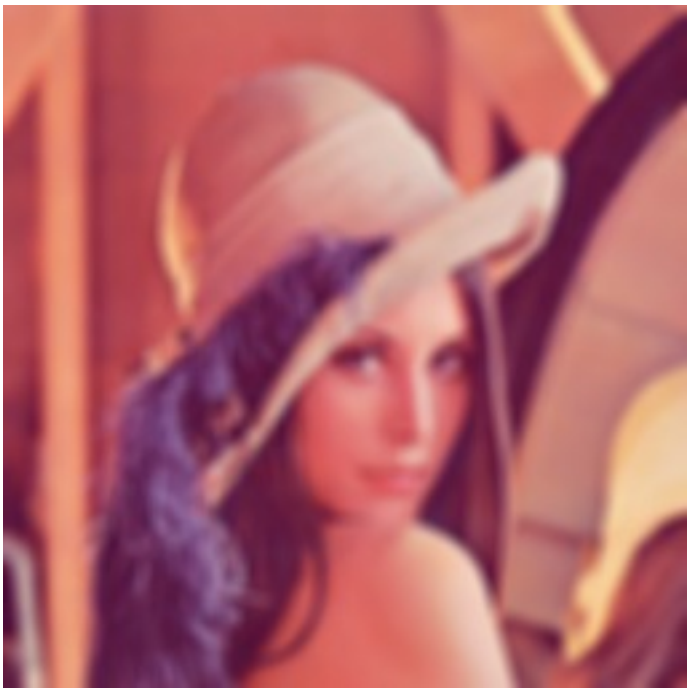


Figure 3.96: radius 3

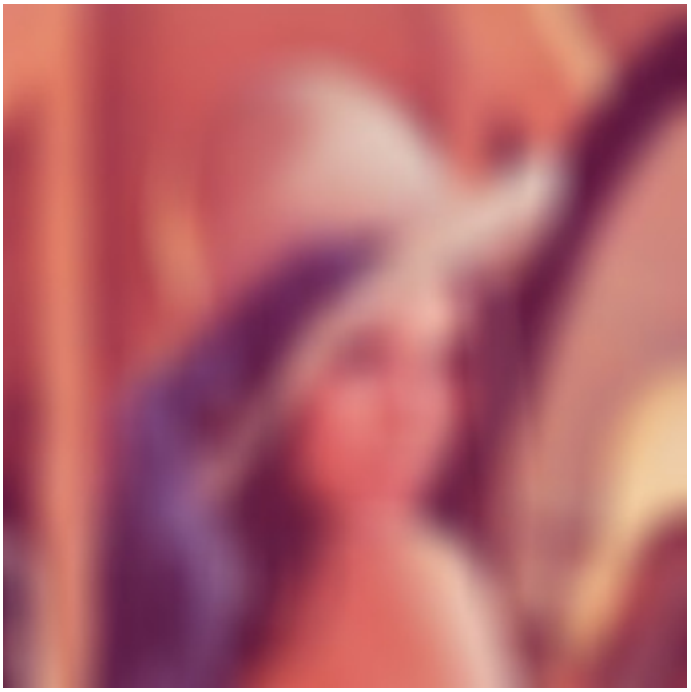


Figure 3.97: radius 10

Figure 3.98: Original

Figure 3.99: 10 degrees

Figure 3.100: 45 degrees

Figure 3.101: Original

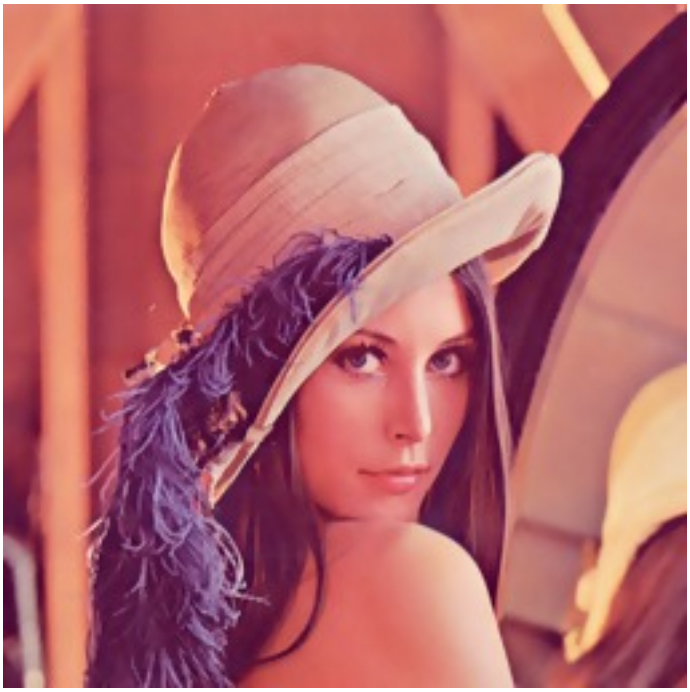


Figure 3.102: Denoised image

#### Removing speckles

`pystacia.image.Image.despeckle()` on the other hand removes speckles - larger grain defects than noise.

```
>>> img.despeckle()
```

#### Embossing

Emboss raises detected edges in image creating 3D effect sharpening it at the same time. Call `pystacia.image.Image.emboss()` to use it.

```
>>> img.emboss()
```

### 3.3.6 Deforming

#### Swirling

To apply whirlpool like effect use `pystacia.image.Image.swirl()`. Positive angles result in clockwise whirling, negative in counter-clockwise.

```
>>> img.swirl(60)
```

```
>>> img.swirl(-30)
```

Figure 3.103: Original

Figure 3.104: Despeckled image

Figure 3.105: Original

Figure 3.106: Embossed image

Figure 3.107: Original

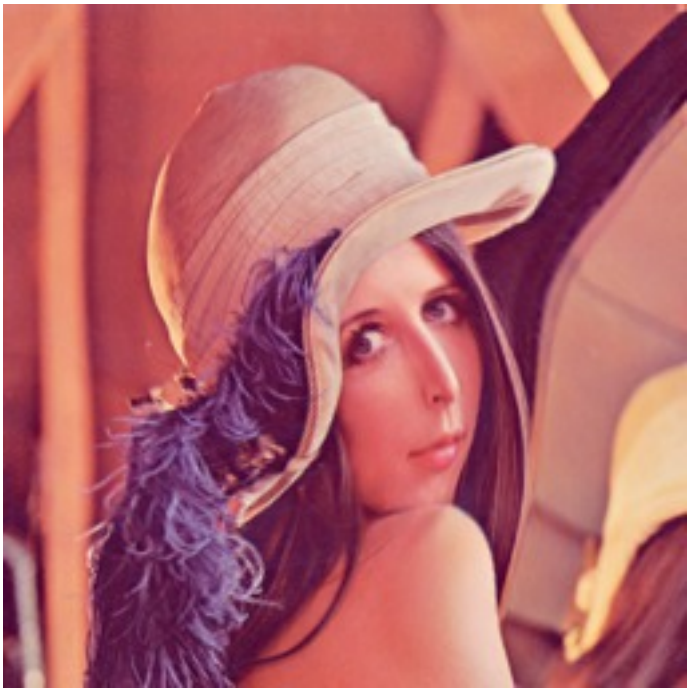


Figure 3.108: 60 degrees

Figure 3.109: -30 degrees

### Waving

`pystacia.image.Image.wave()` applies sinusoidal deformation along give axis (defaults to x). You can control amplitude and length of the wave. Resulting extra pixels are transparent.

```
>>> img.wave(20, 100)

>>> img.wave(-10, 50)

>>> img.wave(50, 200 axis=axes.y)

>>> img.wave(10, 30, axis=axes.y)
```

Figure 3.110: Original

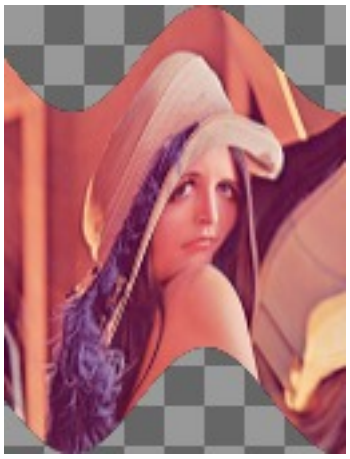


Figure 3.111: (100, 20, x)

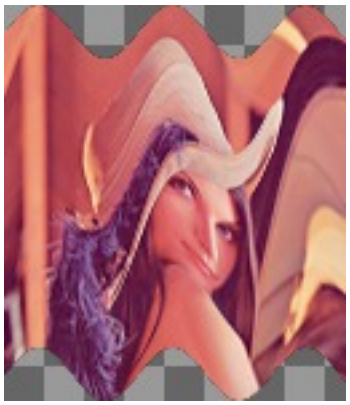


Figure 3.112: (50, -10, x)

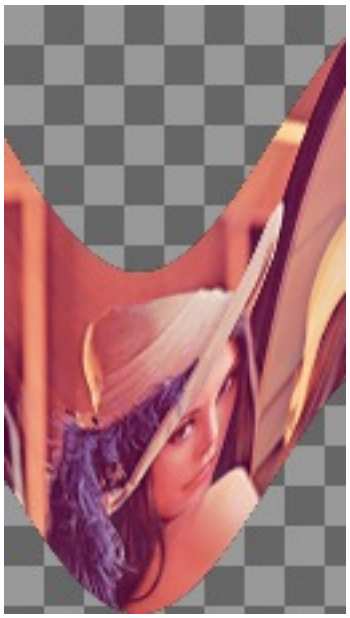


Figure 3.113: (200, 50, y)

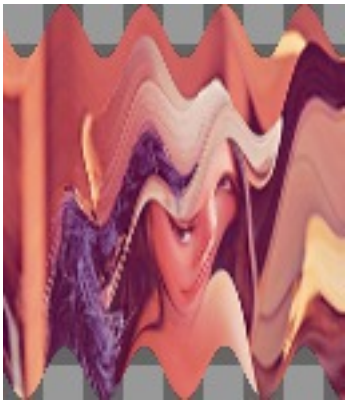


Figure 3.114: (30, 10, y)

### 3.3.7 Special effects

#### Sketch effect

You can use `pystacia.image.Image.sketch()` to simulate sketch effect. You can control the effect with two parameters radius of strokes and angle of pencils (defaults to 45 degrees).

```
>>> image.sketch(3)
```

```
>>> image.sketch(6, 0)
```

Figure 3.115: Original

Figure 3.116: radius 3, angle 45

Figure 3.117: radius 6, angle 0

### Oil paint effect

`pystacia.image.Image.oil_paint()` simulates oil painting by covering image with circles filled with mean color value. It accepts single paramater - radius in pixels.

```
>>> image.oil_paint(3)
```

```
>>> image.oil_paint(8)
```

Figure 3.118: Original

### Spreading

`pystacia.image.Image.spread()` fuzzes and image with pixel displacement within given radius.

```
>>> image.spread(3)
```

```
>>> image.spread(6)
```

### Fx method

With `pystacia.image.Image.fx()` you can perform custom operations using *ImageMagick* tiny scripting language. Beware that this can be very slow on large images as it's directly interpreted and not compiled in any way. http://www.imagemagick.org/script/fx.php has infomration on syntax.

```
>>> image.fx('u * 1/2')
```

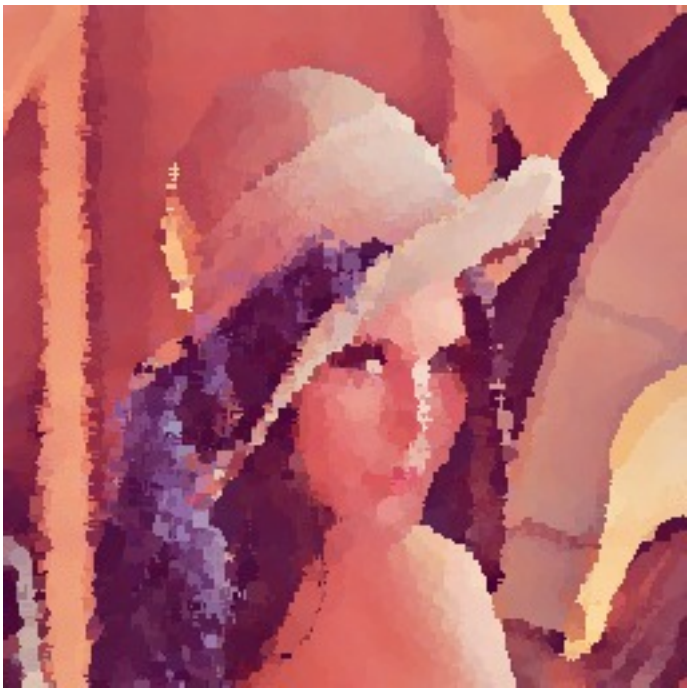


Figure 3.119: radius 2

Figure 3.120: radius 8

Figure 3.121: Original

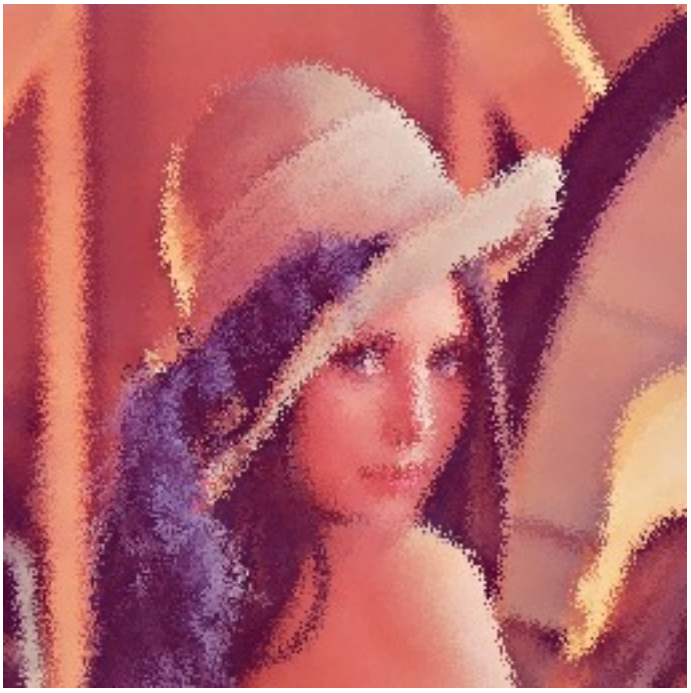


Figure 3.122: radius 2

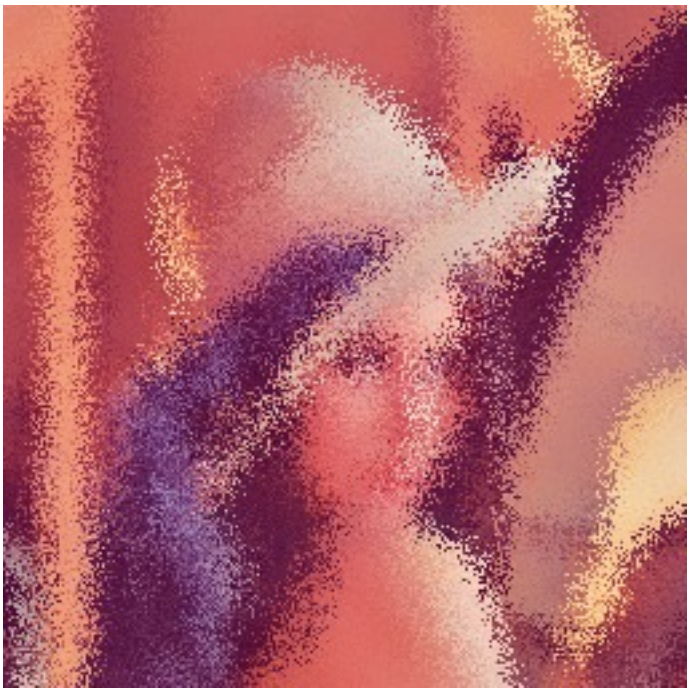


Figure 3.123: radius 6

Figure 3.124: Original

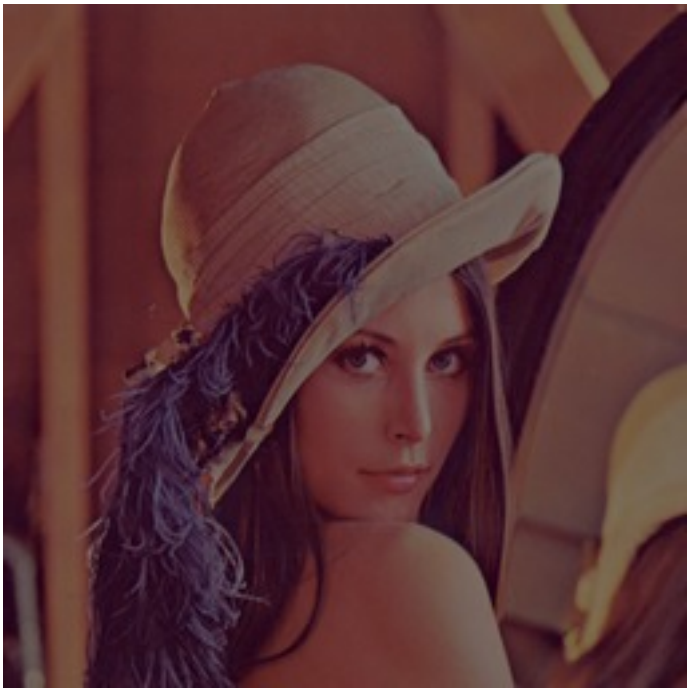


Figure 3.125: After processing

### 3.3.8 Pixel manipulation

#### Reading single pixels

To access pixel data anywhere in the image you cen use `pystacia.image.Image.get_pixel()` passing it x and y coordinates.

```
>>> image.get_pixel(128, 128)
<Color(r=0.9396,g=0.5933,b=0.4317,a=1) object at 0x108002200L>
```

#### Filling

If you want to fill image with solid color you use `pystacia.image.Image.fill()` passing it color. You can optionally pass also blend parameter specyfing opacity with *1* meaning opaque.

```
>>> image.fill(color.from_string('red'))

>>> image.fill(color.from_string('green'), 0.5)

>>> image.fill(color.from_string('blue'), 0.25)

>>> image.fill(color.from_string('orange'), 0.2)
```

#### Setting color

Another way to paint over whole image is using `pystacia.image.Image.set_color()`. Unlinke `pystacia.image.fill()` it always discards backgorund information replacing pixels. You can use alpha component of color to gain translucency.

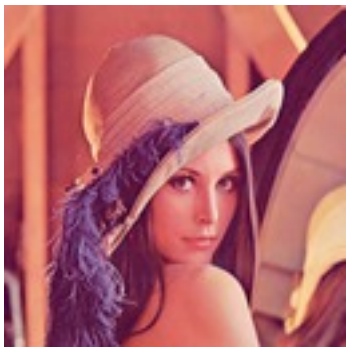


Figure 3.126: Original

Figure 3.127: red

Figure 3.128: green 0.5 blend

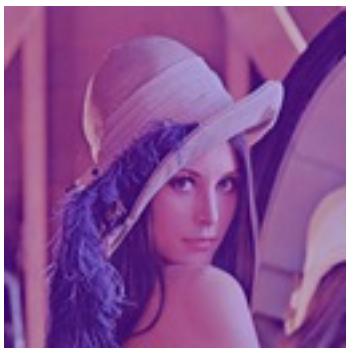


Figure 3.129: blue 0.25 blend

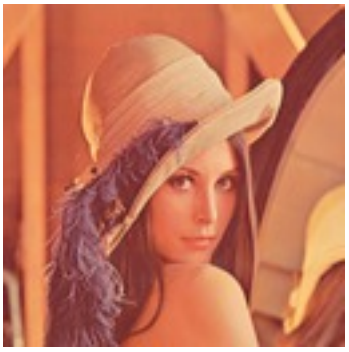


Figure 3.130: orange 0.2 blend

```
>>> image.set_color(color.from_string('red'))

>>> image.set_color(color.from_rgba(0, 1, 0, 0.5))

>>> image.set_color(color.from_rgba(0, 0, 0, 0.2))

>>> image.set_color(color.from_rgba(1, 0, 1, 0.5))
```

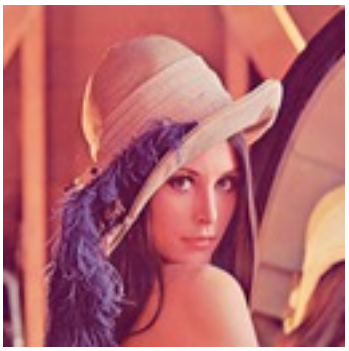


Figure 3.131: Original

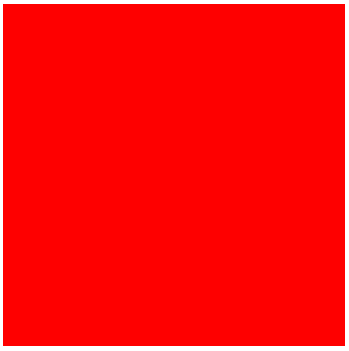


Figure 3.132: red

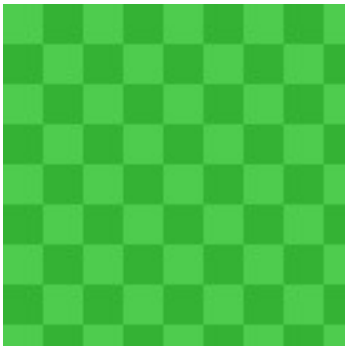


Figure 3.133: green 0.5 alpha

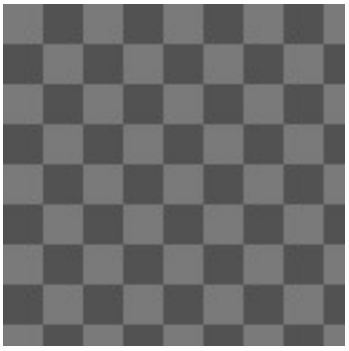


Figure 3.134: black 0.2 alpha

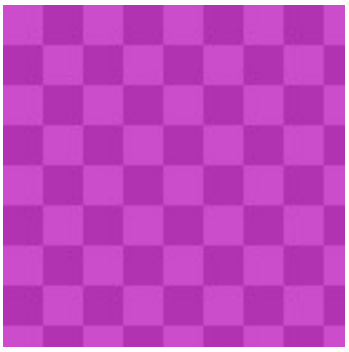


Figure 3.135: violet 0.5 alpha

### Setting alpha

Sometimes you may want to override alpha level for all pixels at once. You can do that with `pystacia.image.Image.set_alpha()`.

```
>>> image.set_alpha(0.75)

>>> image.set_color(0.5)

>>> image.set_color(0.25)

>>> image.set_color(0)
```

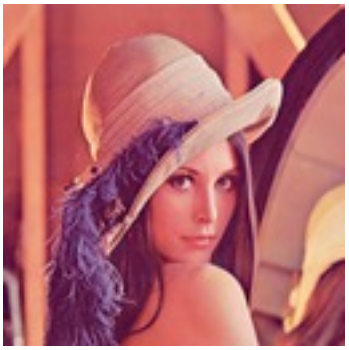


Figure 3.136: Original

Figure 3.137: 0.75

Figure 3.138: 0.5

### Overlaying

With `pystacia.image.Image.overlay()` you can overlay images on top of image the method is called from. It accepts the image that is going to be overlaid as first parameter, x and y coordinates and composite mode. There are many composite modes available. *over* is the default one, other popular ones include *colorize*, *multiply*, *overlay*, *pin_light*.

```
>>> image.overlay(other, 32, 32)

>>> image.overlay(other, 32, 32, composites.colorize)

>>> image.overlay(other, 32, 32, composites.multiply)

>>> image.overlay(other, 32, 32, composites.overlay)

>>> image.overlay(other, 32, 32, composites.pin_light)

>>> image.overlay(pther, 32, 32, composites.saturate)
```

Figure 3.139: 0.25

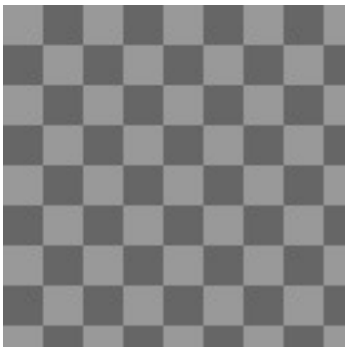


Figure 3.140: 0

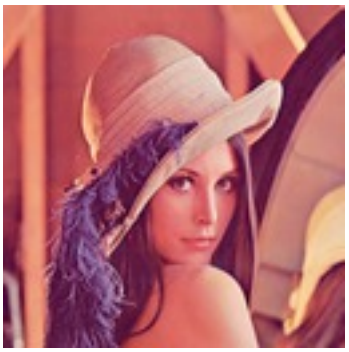


Figure 3.141: Original

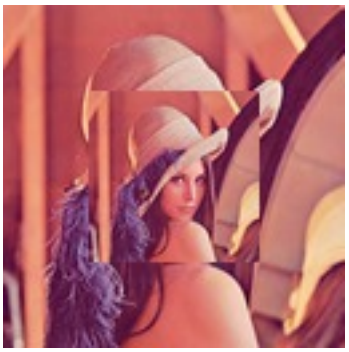


Figure 3.142: Over (default)

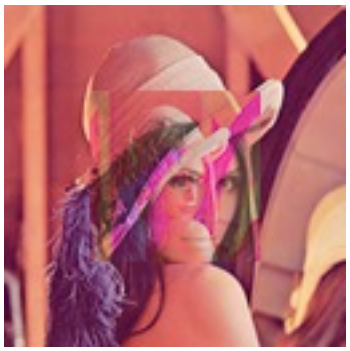


Figure 3.143: colorize

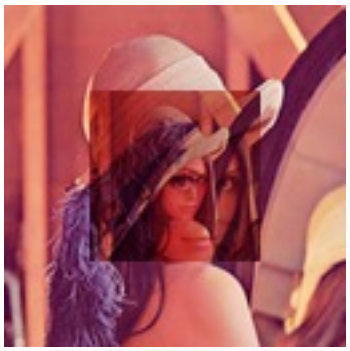


Figure 3.144: multiply

Figure 3.145: overlay

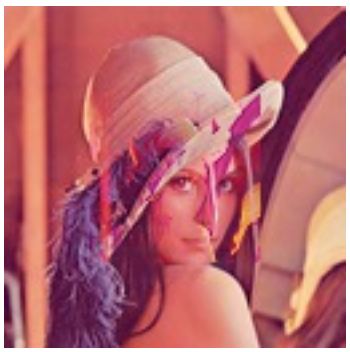


Figure 3.146: pin_light

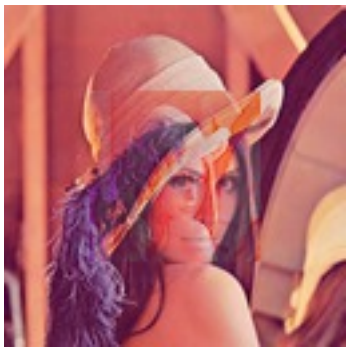


Figure 3.147: saturate

Figure 3.148: soft_light

Figure 3.149: modulus_add

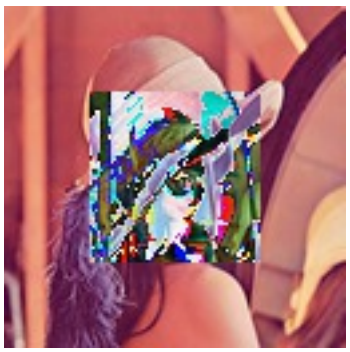


Figure 3.150: modulus_substract

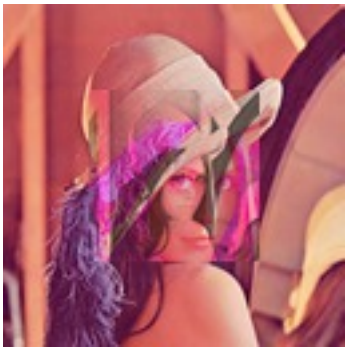


Figure 3.151: luminize

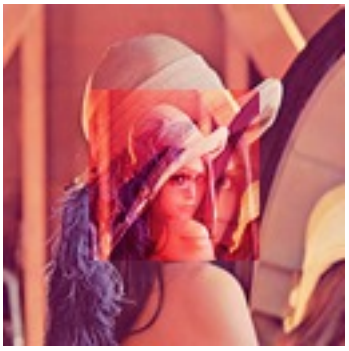


Figure 3.152: hard_light

### 3.3.9 Utilities

#### Displaying on the screen

When in GUI session you can display image in available image preview program with `pystacia.image.Image.show()`. The call is non-blocking meaning that the control is immediately returned to your program

```
>>> image.show()
```

#### Marking transparent pixels

Sometimes it might be not clear from the context which pixels are translucent and which are opaque. You can use `pystacia.image.Image.checkerboard()` to overlay your image on top of checkerboard pattern in the same manner that *Photoshop* does.

```
>>> image.checkerboard()
```

### 3.3.10 Bundled images

pystacia comes with standard test images which can be used for testing purposes. Most of them are embedded in *ImageMagick* library.

`pystacia.lena()` optionally accepting size parameter:

`pystacia.magick_logo():`

`pystacia.rose():`

`pystacia.wizard()`

`pystacia.granite()`

`pystacia.netscape()`

## 3.4 Working with color

Couple of methods in Pystacia use color as argument. There are many ways to factory a color in Pystacia. All the machinery is defined in `pystcia.color` As a convention all the channel information for red, blue, green, alpha and so on is specified as `float` numbers between *0* and *1*. It can be misleading for people used to thinking in 8-bit 0-255 mode. Pystacia uses `float` because of internal 16-bit precision that *ImageMagick* works in and also its *C* interface uses *float*.

You normally don't instantiate `pystacia.color.Color` directly. You should rely on factory methods presented below instead.

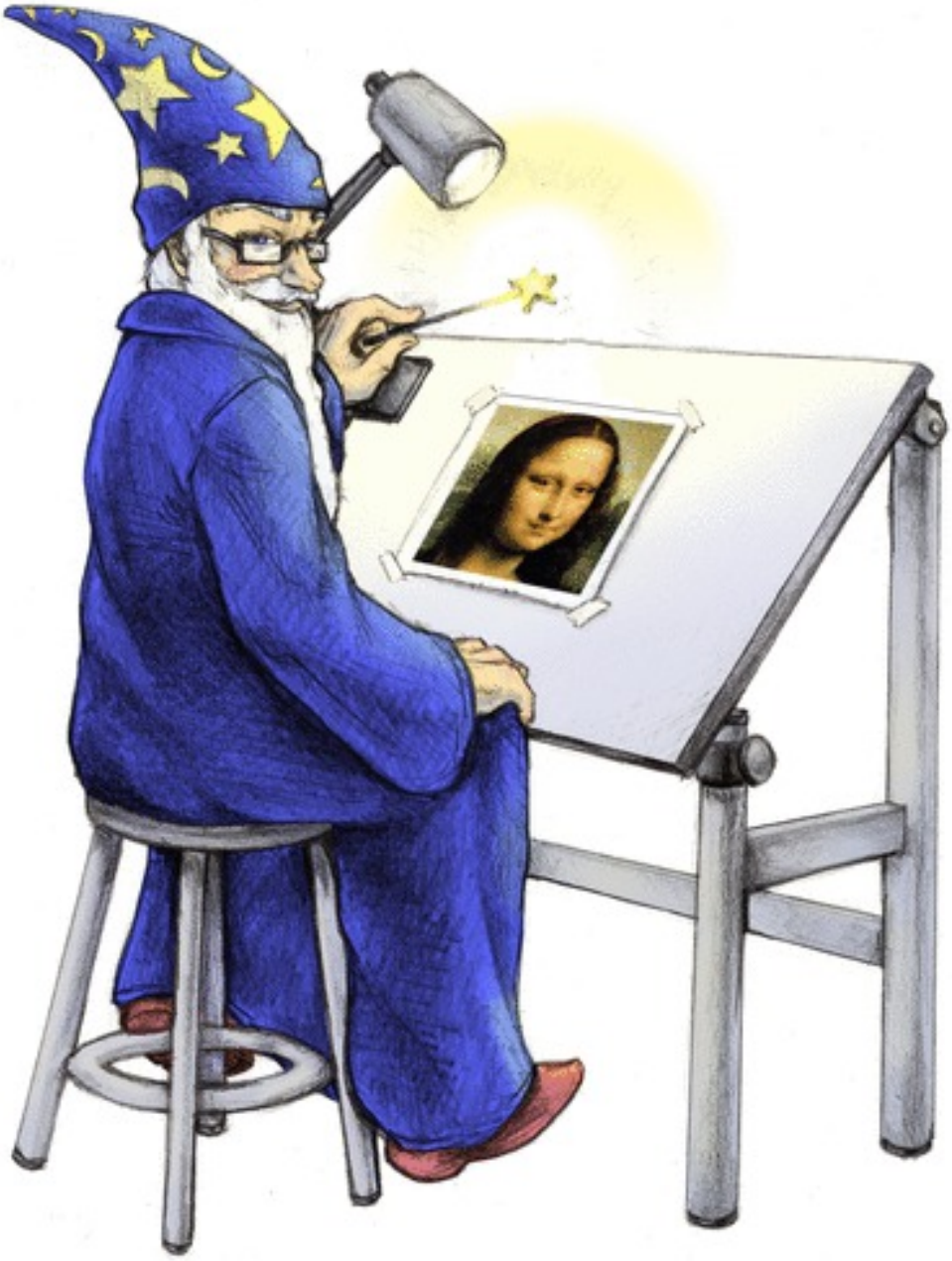
### 3.4.1 Creating colors from string

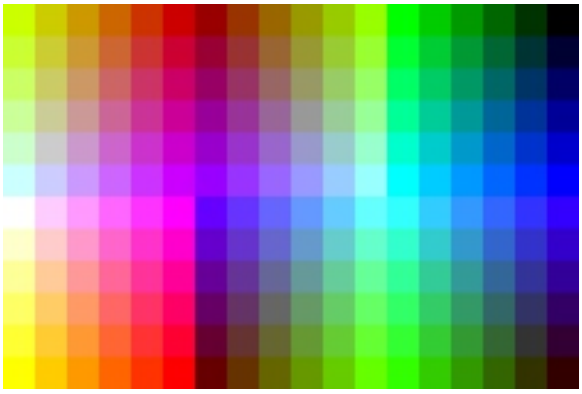
`pystacia.color.from_string()` can be used to synthesize colors from their string representations. It accepts broad variety of input formats as defined in *CSS3* color module.

Well known colors:

```
>>> color.from_string('transparent')
<Color(r=0,g=0,b=0,a=0) object at 0x10a858c00L>
>>> color.from_string('red')
<Color(r=1,g=0,b=0,a=1) object at 0x103716c00L>
```


Image Magick





```
>>> color.from_string('teal')
<Color(r=0,g=0.502,b=0.502,a=1) object at 0x10a856a00L>
```

Function syntax:

```
>>> color.from_string('rgb(255, 255, 255)')  # rgb
<Color(r=1,g=1,b=1,a=1) object at 0x108047e00L>
>>> color.from_string('rgb(100%,100%,50%)')  # rgb with percentage
<Color(r=1,g=1,b=0.5,a=1) object at 0x10a817e00L>
>>> color.from_string('rgba(255, 255, 255, 0.5)')  # rgba
<Color(r=1,g=1,b=1,a=0.5) object at 0x10a819a00L>
>>> color.from_string('hsl(120, 100%, 75%)')  # hsl
<Color(r=0.9,g=1,b=0.5,a=1) object at 0x10804a400L>
>>> color.from_string('hsla(240, 100%, 50%, 0.5)')  # hsla
<Color(r=0,g=1,b=0.4,a=0.5) object at 0x1080a0c00L>
```

### 3.4.2 Creating colors from floats

It's not always convenient to use string syntax. You can use `pystacia.color.from_rgb()` and `pystacia.color.from_rgba()` to create colors from numerical values.

```
>>> color.from_rgb(0, 1, 1)
<Color(r=0,g=1,b=1,a=1) object at 0x1037a2800L>
>>>> color.from_rgba(0.5, 0.5, 0.5, 0.5)
<Color(r=0.5,g=0.5,b=0.5,a=0.5) object at 0x10b058a00L>
```

You can also create colors by probing them from images with `pystacia.image.Image.get_pixel()`.

### 3.4.3 Color class

Once instantiated a pystacia.color.Color instance can be queried and modified.

#### Channel information

Red, blue, green and alpha information can be accessed and modified with pystacia.color.Color.red, pystacia.color.Color.green, pystacia.color.Color.blue, pystacia.color.Color.alpha properties that also have convenience one letter abbreviations: pystacia.color.Color.r, pystacia.color.Color.g, pystacia.color.Color.b, pystacia.color.Color.a.

```
>>> red = color.from_string('red')
>>> red.red
1
>>> red.red == red.r
True
>>> red.green
0
>>> red.green = 1
>>> red.g
1
>>> red.a = 0.5
>>> red
<Color(r=1,g=1,b=0,a=0.5) object at 0x108036200L>
```

You can also set several channels at once with pystacia.color.Color.set_rgb() and pystacia.color.Color.set_rgba() methods:

```
>>> red.set_rgb(0, 0.5, 1)
>>> red
<Color(r=0,g=0.5,b=1,a=0.5) object at 0x108036200L>
>>> red.set_rgba(1, 1, 1, 0.1)
>>> red
<Color(r=1,g=1,b=1,a=0.1) object at 0x108036200L>
```

Also access all channels at once as tuples with pystacia.color.Color.get_rgb() and pystacia.color.Color.get_rgba():

```
>>> red.get_rgb()
(1, 1, 1)
>>> red.get_rgba()
(1, 1, 1, 0.1)
```

To return value *CSS3* string representation of color use pystacia.color.Color.get_string() or cast instance with str():

```
>>> red.get_string()
'rgba(255, 255, 255, 0.1)'
>>> str(red)
'rgba(255, 255, 255, 0.1)'
```

#### Testing for transparency

You can query if color is fully transparent with pystacia.color.Color.transparent property whilst you can use pystacia.color.Color.opaque to test if color is fully opaque.

```
>>> red = color.from_string('red')
>>> red.opaque
True
>>> red.transparent
False
>>> transparent = color.from_string('transparent')
>>> transparent.opaque
False
>>> transparent.transparent
True
```

## 3.5 Miscellaneous

### 3.5.1 Skipping binary install

Pystacia to work requires *ImageMagick* shared libraries. Specifically *MagickWand* DLL. Pystacia by default does install prebuilt binaries. Sometimes you may want to compile *ImageMagick* yourself and not use packaged one. You can set environment variable `PYSTACIA_SKIP_BINARIES` to non-false value while performing installation to skip copying of DLLs.

....$ PYSTACIA_SKIP_BINARIES=1 pip install pystacia

### 3.5.2 Library search path

When performing :term:'MagickWand' search theses directories are inspected in following order:

- `PYSTACIA_LIBRARY_PATH` if set
- *cdll* subdirectory under package install location if `PYSTACIA_SKIP_PACKAGE` is not set
- if inside *virtualenv* subdirectories *lib* and *cdll* under `VIRTUAL_ENV` are also inspected
- system-wide locations

When loading a library SONAMEs in file names are preferred in this order: 5, 4, 3 and no SONAME.

### 3.5.3 Subclassing Image class

Any factory function inside Pystcia can accept optional factory parameter specifying class or function to be used when instantiating objects.

```
>>> from pystacia import Image
>>> class MyImage(Image):
>>>     def cool_effect(self):
>>>         self.swirl(45)
>>> from pystacia import wizard
>>> img = wizard(factory=MyImage)
>>> img
<MyImage(w=480,h=640,8bit,rgb,palette) object at 0x103297200L>
>>> img.cool_effect()
```

### 3.5.4 Environment variables influencing pystacia runtime

- `PYSTACIA_SKIP_BINARIES` – if set no binary *ImageMagick* build is copy in *setup.py*
- `PYSTACIA_LIBRARY_PATH` – prepend this path to *MagickWand* search path
- `PYSTACIA_SKIP_PACKAGE` – skip inspecting package *cdll* subdirectory on DLL search effectively discarding pre-build binaries even if they are installed
- `PYSTACIA_SKIP_VIRTUALENV` – skip inspecting *lib* and *dll* subdirectories if package was installed under *virtualenv*.
- `PYSTACIA_NO_CHAINS` – disable chaining

CHAPTER 4

# Reference Material

Reference material includes documentation for Pystacia *API*.

## 4.1 API Documentation

### 4.1.1 pystacia

pystacia is a raster graphics library utilizing ImageMagick.

### 4.1.2 pystacia.image

pystacia.image.**read**(*filename*, *factory=None*)

Read `Image` from filename.

**Parameters**

- **filename** (`str`) – file name to read
- **factory** – Image subclass to use when instantiating objects

**Return type** `Image`

Reads file, determines its format and returns an `Image` representing it. You can optionally pass factory parameter to use alternative `Image` subclass.

```
>>> read('example.jpg')
<Image(w=512,h=512,8bit,rgb,truecolor) object at 0x10302ee00L>
```

pystacia.image.**read_blob**(*blob*, *format=None*, *length=None*, *factory=None*)

Read `Image` from a blob string or stream with a header.

**Parameters**

- **blob** (`str` (Python 2.x) / `bytes` (Python 3.x) or file-like object) – blob data string or stream
- **format** (`str`) – container format such as *JPEG* or *BMP*
- **length** (`int`) – read maximum this bytes from stream
- **factory** – Image subclass to use when instantiating objects

**Return type** `Image`

Reads image from string or data stream that contains a valid file header i.e. it carries information on image dimensions, bit depth and compression. Data format is equivalent to e.g. *JPEG* file read into string(Python 2.x)/bytes(Python 3.x) or file-like object. It is useful in cases when you have open file-like object but not the file itself in the file system. That often happens in web applications which map *POST* data with file-like objects. Format and length are typically not used but can be a hint when the information cannot be guessed from the data itself. You can optionally pass factory parameter to use alternative `Image` subclass.

```
>>> with file('example.jpg') as f:
...     img = read_blob(f)
>>> img
<Image(w=512,h=512,8bit,rgb,truecolor) object at 0x10302ee00L>
```

pystacia.image.**read_raw**(*raw*, *format*, *width*, *height*, *depth*, *factory=None*)

Read `Image` from raw string or stream.

**Parameters**

- **raw** (`str` (Python 2.x) / `bytes` (Python 3.x) or file-like object) – raw data string or stream
- **format** (`str`) – raw pixel format eg. `'RGB'`
- **width** (`int`) – width of image in raw data
- **height** (`int`) – height of image in raw data
- **depth** (`int`) – depth of a single channel in bits
- **factory** – Image subclass to use when instantiating objects

**Return type** `Image`

Reads image data from a raw string or stream containing data in format such as *RGB* or *YCbCr*. The contained image has dimensions of width and height pixels. Each channel is of depth bits. You can optionally pass factory parameter to use alternative `Image` subclass.

```
>>> img = read_raw(b'raw triplets', 'rgb', 1, 1, 8)
```

pystacia.image.**blank**(*width*, *height*, *background=None*, *factory=None*)

Create `Image` with monolithic background

**Parameters**

- **width** (`int`) – Width in pixels
- **height** (`int`) – Height in pixels
- **background** (`pystacia.Color`) – background color, defaults to fully transparent

Creates blank image of given dimensions filled with background color.

```
>>> from pystacia.image import color
>>> blank(32, 32, color.from_string('red'))
<Image(w=32,h=32,16bit,rgb,palette) object at 0x108006000L>
```

### The `Image` class

**class** pystacia.image.**Image**(*resource=None*)

**adaptive_blur**(*radius*, *strength=None*, *bias=None*)

Adaptively blur an image

**Parameters** **radius** (`float`) – radius in pixels

Applies adaptive blur of given radius to an image.

This method can be chained.

**adaptive_sharpen**(*radius*, *strength=None*, *bias=None*)
Adaptive sharpen an image

**Parameters** **radius** (`float`) – radius in pixels

Applies adaptive sharpening to an image.

This method can be chained.

**add_noise**(*attenuate=0*, *noise_type=None*)
Add noise to an image.

**Parameters**

- **attenuate** (`float`) – Attenuation factor
- **noise_type** – values representing

`pystacia.image.enum.noises`

Adds noise of given type to an image. Noise type defaults to `"gaussian"`.

This method can be chained.

**auto_gamma**()
Auto-gamma image.

Extracts the 'mean' from the image and adjust the image to try make set its gamma appropriately.

This method can be chained.

**auto_level**()
Auto-level image.

Adjusts the levels of an image by scaling the minimum and maximum values to the full quantum range.

This method can be chained.

**blur**(*radius*, *strength=None*)
Blur image.

**Parameters**

- **radius** (*float*) – Gaussian operator radius
- **strength** (*float*) – Standard deviation (sigma)

Convolves the image with a Gaussian operator of the given radius and standard deviation (strength).

This method can be chained.

**brightness**(*factor*)
Brightens an image.

**Parameters** **factor** (`float`) – Brightness factor between -1 and 1

Brightens an image with specified factor. Factor of `0` is no-change operation. Values towards `-1` make image darker. `-1` makes image completely black. Values towards 1 make image brigther. `1` makes image completely white.

This method can be chained.

**charcoal**(*radius*, *strength=None*, *bias=None*)
Simulate a charcoal.

**Parameters** **radius** (`float`) – Charcoal radius

This method can be chained.

**checkerboard**()
Fills transparent pixels with checkerboard.

Useful for presentation when you want to explicitly mark transparent pixels when otherwise it might be unclear where they are.

**colorize**(*color*)
Colorize image.

**Parameters** **color** (`pystacia.color.Color`) – color from which hue value is used

Colorizes image resulting in image containing only one hue value.

This method can be chained.

**colorspace**
Return or set colorspace associated with image.

Sets or gets colorspace. When you set this property there's no colorspace conversion performed and the original channel values are just left as is. If you actually want to perform a conversion use `convert_colorspace` instead. Popular colorspace include RGB, YCbCr, grayscale and so on.

**Return type** `pystacia.lazyenum.EnumValue`

**compare**(*image*, *metric=None*, *factory=None*)
Compare image to another image

**Parameters**

- **image** (`pystacia.image.Image`) – reference image
- **metric** (`pystacia.lazyenum.EnumValue`) – distortion metric
- **factory** – factory to be used to create difference image

**Return type**

`tuple` of `Image` and `float` distortion in given metric

Compares two images of the same sizes. Returns a tuple of an image marking different parts with red color and a distortion metric. By default it uses `pystacia.image.metrics.absolute_error` metric. If images are of different sizes returns `False` instead.

**contrast**(*factor*)
Change image contrast.

**Parameters** **factor** (`float`) – Contrast factor between -1 and 1

Change image contrast with specified factor. Factor of `0` is no-change operation. Values towards `-1` make image less contrasting. `-1` makes image completely gray. Values towards `1` increase image contrast. `1` pulls channel values towards 0 and 1 resulting in a highly contrasting posterized image.

This method can be chained.

**contrast_stretch**(*black=0*, *white=1*)
Stretch image contrast

**convert_colorspace**(*colorspace*)
Convert to given colorspace.

**Parameters** **colorspace** (`pystacia.color.Color`) – destination colorspace

Converts an image to a given colorspace.

```
>>> img = read('example.jpg')
>>> img.convert_colorspace(colorspaces.ycbcr)
>>> img.colorspace == colorspaces.ycbcr
True
```

This method can be chained.

**denoise**()
Attempt to remove noise preserving edges.

Applies a digital filter that improves the quality of a noisy image.

This method can be chained.

**depth**
Set or get image depth per channel.

**Return type** int

Set or get depth per channel in bits. Either 8 or 16.

**desaturate**()
Desatures an image.

Reduces saturation level of all pixels to minimum yielding grayscale image.

This method can be chained.

**despeckle**()
Attempt to remove speckle preserving edges.

Resulting image almost solid color areas are smoothed preserving edges.

This method can be chained.

**detect_edges**(*radius*, *strength=None*)
Detect edges in image.

**Parameters** **radius** (float) – radius of detected edges

Analyzes image and marks contrasting edges. It operates on all channels by default so it might be a good idea to grayscale an image before performing it.

This method can be chained.

**emboss**(*radius=0*, *strength=None*)
Apply edge detecting algorithm.

**Parameters**

- **radius** (int) – filter radius
- **strength** (int) – filter strength (sigma)

On a typical photo creates effect of raised edges.

This method can be chained.

**equalize**()
Equalize image histogram.

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. See also: http://en.wikipedia.org/wiki/Histogram_equalization.

This method can be chained.

**evaluate**(*operation*, *value*)
Apply operation to image color information

**Parameters**

- **operation** (Enum `operations` representation) – Operation like "multiply" or "subtract"
- **value** (`float`) – A constant value used as operand

Evaluate each pixel in image with operation and provided value. E.g. using `'divide'` operation and `2` as value would result in all pixels divided by two.

**fill**(*fill*, *blend=1*)
Overlay color over whole image.

**Parameters**

- **fill** (`pystacia.color.Color`) – color to overlay
- **blend** (`float`) – overlay blending

Overlay a color over whole image. Blend is blending factor of a color with *0* being completely transparent and `1` fully opaque.

This method can be chained.

**fit**(*width=None*, *height=None*, *mode=None*, *upscale=False*, *filter=None*, *blur=1*)
Fits an image into a rectangle preserving aspect ratio

**Parameters**

- **width** (`int`) – width in pixels
- **height** (`int`) – height in pixels

Fits an image into a region of width, and height pixels preserving aspect ratio.

This method can be chained.

**flip**(*axis*)
Flip an image along given axis.

**Parameters** **axis** (`pystacia.lazyenum.EnumValue`) – X or Y axis

Flips (mirrors) an image along `axes.x` or `axes.y`.

This method can be chained.

**format**
Get original image format.

**Return type** `str`

Return format image was originally stored in.

**fx**(*expression*)
Perform expression using ImageMagick mini-language.

**Parameters** **expression** – expression to evaluate

For more information on the available expressions visit: http://www.imagemagick.org/script/fx.php.

This method can be chained.

**gamma**(*gamma*)
Apply gamma correction to an image.

**Parameters** **gamma** (`float`) – Gamma value

Apply gamma correction to an image. Value `1` is an identity operation. Higher values yield brighter image and lower values darken image. More information on gamma correction can be found here: http://en.wikipedia.org/wiki/Gamma_correction.

This method can be chained

**gaussian_blur** (*radius*, *strength=None*, *bias=None*)
Gaussian blur an image

**Parameters** **radius** (`float`) – radius in pixels

Applies Gaussian blur to an image of given radius.

This method can be chained.

**get_blob** (*format*, *compression=None*, *quality=None*, *factory=None*)
Return a blob representing an image

**Parameters**

- **format** (`str`) – format of the output such as *JPEG*
- **compression** (`pystacia.lazyenum.EnumValue`) – compression supported by format
- **quality** – output quality

**Return type** `str` (Python 2.x) / `bytes` (Python 3.x)

Returns blob carrying data representing an image along its header in the given format. Compression is one of compression algorithms. Some formats like *TIFF* supports more then one compression algorithms but typically this parameter is not used. The interpretation of quality parameter depends on the chosen format. E.g. for *JPEG* it's integer number between 1 (worst) and 100 (best). The default value is to choose best available compression that preserves good quality image. The details are in the *ImageMagick documentation <http://www.imagemagick.org/script/ command-line-options.php#quality>*.

**get_pixel** (*x*, *y*, *factory=None*)
Get pixel color at given coordinates.

**Parameters**

- **x** (`int`) – x coordinate of pixel
- **y** (`int`) – y coordinate of pixel

**Return type** **:color:'pystacia.color.Color'**

Reads pixel color at point `(x,y)`.

**get_range** ()
Return range minimum and maximum range of channels

**Return type** `tuple`

Return a range of color information as a tuple of floats between 0 and 1. E.g. black and white image would have a range of 0 for minimum and 1 for maximum 1-color images will have maximum equal to minimum.

**get_raw** (*format*, *factory=None*)
Return `dict` representing raw image data.

**Parameters** **format** (`str`) – format of the output such as *RGB*

**Return type** `dict`

Returns raw data `dict` consisting of raw, format, width, height and depth keys along their values.

**height**
Get image height.

**Return type** int

Return image height in pixels.

**invert**(*only_gray=False*)
Invert image colors.

Inverts all image colors resulting in a negative image.

This method can be chained.

**is_same**(*image*)
Check if images are the same.

**Parameters** **image** – reference image

**Return type** bool

Returns True if images are exactly the same i.e. are of same dimensions and underlying pixel data is exactly the same.

**map**(*lookup*, *interpolation=None*)
Map image using intensities as keys and lookup image.

**Parameters**

- **lookup** (pystacia.image.Image) – Lookup table image
- **interpolation** – interpolation method

Maps an image using lightness as key and copying color values from lookup image.

**modulate**(*hue=0*, *saturation=0*, *lightness=0*)
Modulate hue, saturation and lightness of the image

**Parameters**

- **hue** (float) – Hue value from -1 to 1
- **saturation** (float) – Saturation value from -1 to infinity
- **lightness** (float) – Lightness value from -1 to infinity

Setting any of the parameters to 0 is no-change operation. Hue parameter represents hue rotation relatively to current position. *-1* means rotation by 180 degrees counter-clockwise and 1 is rotation by 180 degrees clockwise. Setting saturation to −1 completely desaturates image (makes it grayscale) while values from 0 towards infinity make it more saturated. Setting lightness to −1 makes image completely black and values from 0 towards infinity make it lighter eventually reaching pure white.

This method can be chained.

**motion_blur**(*radius*, *angle=0*, *strength=None*, *bias=None*)
Motion blur image

**Parameters**

- **radius** (float) – Blur radius in pixels
- **angle** – Angle measured clockwise to X-axis

:type angle:float

Applies motion blur of given radius along in given direction measured in degrees of X-axis clockwise.

This method can be chained.

**normalize**()
Normalize image histogram.

**oil_paint**(*radius*)
Simulates oil painting.

**Parameters** **radius** (`float`) – brush radius

Each pixel is replaced by the most frequent color occurring in a circular region defined by radius.

This method can be chained.

**overlay**(*image*, *x=0*, *y=0*, *composite=None*)
Overlay another image on this image.

**Parameters**

- **image** (`pystacia.image.Image`) – imaged to be overlayed
- **x** (`int`) – x coordinate of overlay
- **y** (`int`) – y coordinate of overlay
- **composite** (`pystacia.lazyenum.EnumValue`) – Composition operator

Overlays given image on this image at `(x, y)` using composite operator. There are many popular composite operators available like lighten, darken, colorize, saturate, overlay, burn or default - over.

```
>>> img = read('example.jpg')
>>> img2 = read('example2.jpg')
>>> img.overlay(img2, 10, 10)
```

This method can be chained.

**posterize**(*levels*, *dither=False*)
Reduces number of colors in the image.

**Parameters**

- **levels** (`int`) – Output number of levels per channel
- **dither** – Weather dithering should be performed

‘type dither: :bool:

Reduces the image to a limited number of color levels. Levels specify color levels allowed in each channel. The channel spectrum is divided equally by level. Very low values (2, 3 or 4) have the most visible effect. `1` produces `1**3` output colors, `2` produces `2**3` colors i.e. `8` and so on. Setting dither to `True` enables dithering.

This method can be chained.

**radial_blur**(*angle*)
Performs radial blur.

**Parameters** **angle** (`float`) – Blur angle in degrees

Radial blurs image within given angle.

This method can be chained.

**rescale**(*width=None*, *height=None*, *factor=None*, *filter=None*, *blur=1*)
Rescales an image to given dimensions.

**Parameters**

- **width** (`int`) – Width of resulting image

- **height** (`int`) – Height of resulting image
- **factor** (`float` or `tuple` of `float`) – Zoom factor
- **filter** (`pystacia.lazuenym.Enums`) – Scaling filter

Rescales an image to a given width and height pixels. If one of the dimensions is set to `None` it gets automatically computed using the other one so that the image aspect ratio is preserved. Instead of supplying width and height you can use factor parameter which is a tuple of two floats specifying scaling factor along x and y axes. You can also pass single float as factor which implies the same factor for both axes. Filter is one of possible scaling algorithms. You can choose from popular *Bilinear*, *Cubic*, *Sinc*, *Lanczos* and many more. By default it uses filter which is most adequate for the scaling you perform i.e. the one which preserves as much as possible detail and sharpness.

```
>>> img = read('example.jpg')
>>> img.size
(32L, 32L)
>>> img.rescale(640, 480)
>>> img.size
(640L, 480L)
>>> img.rescale(factor=.5)
>>> img.size
(320L, 240L)
```

This method can be chained.

**resize** (*width*, *height*, *x=0*, *y=0*)

Resize (crop) image to given dimensions.

**Parameters**

- **width** (`int`) – Width of resulting image
- **height** (`int`) – Height of resulting image
- **x** (`int`) – x origin of resized area
- **y** (`int`) – y origin of resized area

Crops out the given x, y, width, height area of image.

```
>>> img = read('example.jpg')
>>> img.size
(512L, 512L)
>>> img.resize(320, 240, 10, 20)
>>> img.size()
(320L, 240L)
```

This method can be chained.

**roll** (*x*, *y*)

Roll pixels in the image.

**Parameters**

- **x** (`int`) – offset in the x-axis direction
- **y** (`int`) – offset in the y-axis direction

Rolls pixels in the image in the left-to-right direction along x-axis and top-to-bottom direction along y-axis. Offsets can be negative to roll in the opposite direction.

This method can be chained.

**rotate** (*angle*)
Rotate an image.

**Parameters** **angle** – angle of rotation in degrees clockwise

Rotates an image clockwise. Resulting image can be larger in size than the original. The resulting empty spaces are filled with transparent pixels.

This method can be chained

**sepia** (*threshold=0.8*, *saturation=-0.4*)
Sepia-tonne an image.

**Parameters**

- **threshold** (`float`) – Controls hue. Set to sepia tone by default.
- **saturation** – Saturation level

:type saturation:`float`

Perform sepia-toning of an image. You can control hue by adjusting threshold parameter.

This method can be chained.

**set_alpha** (*alpha*)
Set alpha channel of pixels in the image.

**Parameters** **alpha** (*float*) – target alpha value

Resets alpha channel of all pixels in the image to given value between 0 (transparent) and 1 (opaque).

This method can be chained.

**set_color** (*fill*)
Fill whole image with one color.

**Parameters** **fill** (`pystacia.color.Color`) – desired fill color

Fills whole image with a monolithic color.

```
>>> img = read('example.jpg')
>>> img.fill(color.from_string('yellow'))
>>> img.get_pixel(20, 20) == color.from_string('yellow')
True
```

This method can be chained.

**shade** (*azimuth=45*, *elevation=45*, *grayscale=True*)
Simulate 3D shading effect.

**Parameters**

- **azimuth** (`float`) – azimuth in degrees - light direction
- **elevation** (`float`) – elevation above image surface in degrees
- **grayscale** (`bool`) – Whether grayscale image

Simulates 3D effect by finding edges and rendering them as raised with light coming from azimuth direction in elevation degrees above image surface. Azimuth is rotation along Z axis. By default it grayscales an image before applying an effect.

This method can be chained

**sharpen** (*radius*, *strength=None*, *bias=None*)
Sharpen an image.

**Parameters** **radius** (`float`) – Radius of sharpening kernel

Performs sharpening of an image.

This method can be chained.

**show** (*checkerboard=True*, *zoom=1*, *no_gui=False*)
Display an image in GUI.

**Parameters** **no_gui** (`bool`) – Skip opening interactive viewer program

**Return type** `str`

Saves image to temporary lossless file format on a disk and sends it to default image handling program to display. Returns a path to the temporary file. You get no guarantees about life span of a file after process ended since it will be typically deleted when process ends.

**size**
Return a tuple of image width and height.

**Return type** `tuple` of two `int`

Returns a tuple storing image width on first position and image height on second position.

```
>>> img = read('example.jpg')
>> img.size
(640, 480)
```

**sketch** (*radius*, *angle=45*, *strength=None*)
Simulate sketched image.

**Parameters**

- **radius** (`float`) – stroke length.
- **angle** – angle of strokes clockwise relative to horizontal axis
- **strength** (`float`) – effect strength (sigma)

Simulates a sketch by adding strokes into an image.

This method can be chained.

**skew** (*offset*, *axis=None*)
Skews an image by given offsets.

**Parameters**

- **offset** (`int`) – offset in pixels along given axis
- **axis** (`pystacia.lazyenum.EnumValue`) – axis along which to perform skew

Skews an image along given axis. If no axis is given it defaults to X axis.

This method can be chained.

**solarize** (*factor*)
Solarizes an image.

**Parameters** **factor** (`float`) – solarize factor

Applies solarization which is a color value operation similar to what can be a result of partially exposing a photograph in a darkroom. The usable range of factor is from `0` to `1` inclusive. Value of `0` is no-change operation whilst `1` produces a negative. Typically factor `0.5` produces interesting effect.

This method can be chained.

**splice**(*x*, *y*, *width*, *height*)
Insert bands of transparent areas into an image.

**Parameters**

- **x** (`int`) – x coordinate of splice
- **y** (`int`) – y coordinate of splice
- **width** (`int`) – width of splice
- **height** (`int`) – height of splice

This method can be chained.

**spread**(*radius*)
Spread pixels in random direction.

**Parameters** **radius** (`int`) – Maximal distance from original position

Applies special effect method that randomly displaces each pixel in a block defined by the radius parameter.

This method can be chained.

**straighten**(*threshold*)
Attempt to straighten image.

**Parameters** **threshold** (`float`) – Separate background from foreground.

Removes skew from the image. Skew is an artifact that occurs in scanned images because of the camera being misaligned, imperfections in the scanning or surface, or simply because the paper was not placed completely flat when scanned.

This method can be chained.

**swirl**(*angle*)
Distort image with swirl effect.

**Parameters** **angle** (`float`) – Angle in degrees clockwise

Swirls an image by angle clockwise. Angle can be negative resulting in distortion in opposite direction.

This method can be chained.

**threshold**(*factor=0.5*, *mode=None*)
Threshold image forcing pixels into black & white.

**Parameters** **factor** (`float` or `tuple`) – Threshold factor

**Patam mode** One of 'default', 'white', 'black'

Threshold image resulting in black & white image. Factor specify lightness threshold. It's a float ranging from 0 to 1 and defaults to 0.5. Pixels below intensity factor are renderer black and pixels above it end up white. In white mode channel pixels above factor intensity are pushed into their maximum whilst one below are left untouched. In black mode channel pixels below factor are set to `0` whilst one above are left untouched. In random mode factor should be two-element tuple that specify minimum and maximum thresholds. Thresholds are then randomly chosen for each pixel individually from given range.

**total_colors**
Return total number of unique colors in image

**transpose**()
Transpose an image.

Creates a vertical mirror image by reflecting the pixels around the central x-axis while rotating them 90-degrees. In other words each row of source image from top to bottom becomes a column of new image from left to right.

This method can be chained.

**transverse**()
Transverse an image.

Creates a horizontal mirror image by reflecting the pixels around the central y-axis while rotating them 270-degrees.

This method can be chained.

**trim**(*similarity=0.1*, *background=None*)
Attempt to trim off extra background around image.

**Parameters**

- **similarity** (`float`) – Similarity factor
- **background** (`pystacia.color.Color`) – background color, transparent by default

Removes edges that are the background color from the image. The greater similarity the more distant hues are considered the same color. Similarity of *0* means only this exact color.

This method can be chained.

**type**
Set or get image type.

**Return type** `pystacia.lazyenum.EnumValue`

Popular image types include truecolor, pallete, bilevel and their matter counterparts.

```
>>> img = read('example.jpg')
>>> img.type == types.truecolor
```

**wave**(*amplitude*, *length*, *offset=0*, *axis=None*)
Apply wave like distortion to an image.

**Parameters**

- **amplitude** (`int`) – amplitude (A) of wave in pixels
- **length** (`int`) – length (lambda) of wave in pixels.
- **offset** – offset (phi) from initial position in pixels
- **axis** (`pystacia.enum.EnumValue`) – axis along which to apply deformation. Defaults to x.

Applies wave like distortion to an image along chosen axis. Axis defaults to :attr:`axes.x`. Offset parameter is not effective as for now. Will be implemented in the feature. Resulting empty areas are filled with transparent pixels.

This method can be chained.

**width**
Get image width.

**Return type** `int`

Return image width in pixels.

**write**(*filename*, *format=None*, *compression=None*, *quality=None*, *flatten=None*, *background=None*)
Write an image to file system.

**Parameters**

- **filename** (`str`) – file name to write to.
- **format** (`str`) – file format
- **compression** (`pystacia.lazyenum.EnumValue`) – compression algorithm
- **quality** (`int`) – output quality

Saves an image to disk under given filename, format is determined from filename unless specified explicitly. The interpretation of quality parameter depends on the chosen format. E.g. for *JPEG* it's a integer number between 1 (worst) and 100 (best). The default value is to choose best available compression that preserves good quality image. The details are in the *ImageMagick documentation <http://www.imagemagick.org/script/ command-line-options.php#quality>*.

```
>>> img = blank(10, 10)
>>> img.write('example.jpg')
>>> img.close()
```

This method can be chained.

### 4.1.3 pystacia.color

#### Color factories

These functions exist for easy creation of `Color` objects from different input formats.

`pystacia.color.`**`from_string`**(*value*, *factory=None*)

Create `Color` from string.

**Parameters**

- **value** (`str`) – CSS color specification
- **factory** – alternative `Color` subclass to use

**Return type** `Color` or factory instance

Creates new instance from a valid color specification string as in *CSS* 2.1. Supported formats include rgb, rgba, hsl, hsla, color identifiers, hexadecimal values, integer and percent values where applicable. When factory is specified this type is used instead of default `Color` type.

```
>>> from_string('red')
<Color(r=1,g=0,b=0,a=1) object at 0x10320e400L>
>>> from_string('rgb(1,1,0)')
<Color(r=1,g=1,b=0,a=1) object at 0x103270600L>
>>> from_string('#fff')
<Color(r=1,g=1,b=1,a=1) object at 0x103251800L>
>>> from_string('hsla(50%, 100%, 100%, 0.5)')
<Color(r=0,g=1,b=1,a=0.5) object at 0x103252a00L>
```

`pystacia.color.`**`from_rgb`**(*r*, *g*, *b*, *factory=None*)

Create opaque `Color` from red, green and blue components.

**Parameters**

- **r** – red component
- **g** – green component
- **b** – blue component

- **factory** – alternative `Color` subclass to use

**Return type** `Color` or factory instance

Red, green and blue components should be numbers between 0 and 1 inclusive. Resulting color is opaque (alpha channel equal to 1). When factory is specified this type is used instead of default `Color` type.

```
>>> from_rgb(0.5, 1, 0.5)
<Color(r=0.5,g=1,b=0.5,a=1) object at 0x103266200L>
```

pystacia.color.**from_rgba**(*r*, *g*, *b*, *a*, *factory=None*)

Create `Color` from red, green, blue and alpha components.

**Parameters**

- **r** – red component
- **g** – green component
- **b** – blue component
- **a** – alpha component
- **factory** – alternative `Color` subclass to use

**Return type** `Color` or factory instance

Red, green, blue and alpha components should be numbers between 0 and 1 inclusive.

```
>>> from_rgba(1, 1, 0, 0.5)
<Color(r=1,g=1,b=0,a=0.5) object at 0x103222600L>
```

### The `Color` class

**class** pystacia.color.**Color**(*resource=None*)

Object representing color information.

**a**

Convenience synonym for `alpha`.

**alpha**

Set or get alpha channel information.

The value ought to be a float between 0 and 1.

**Return type** `float` or `int`

**b**

Convenience synonym for `blue`.

**blue**

Set or get blue channel information.

The value ought to be a float between 0 and 1.

**Return type** `float` or `int`

**g**

Convenience synonym for `green`.

**get_hsl**()

Return hue, saturation and lightness components.

**Return type** tuple

**get_int24**()
Return RGB triplet as single 24 bit integer.

Returns an integer representing this color. Highest 8 bit represent red channel information.

**get_rgb**()
Return red, green and blue components.

**Return type** tuple

Returns tuple containing red, green and blue channel information as numbers between 0 and 1.

**get_rgb8**()
Return red, gren and blue components as 8bit integers

**Return type** `tuple`

Returns tuple containing red, green and blue in this order as 8bit integers randing from 0 to 255

**get_rgba**()
Return red, green, blue and alpha components.

**Return type** `tuple`

Returns tuple containing red, green, blue and alpha channel information as numbers between 0 and 1.

**get_string**()
Return string representation of color.

**Return type** `str`

Returns standard CSS string representation of color either `rgb(r, g, b)` or `rgba(r, g, b, a)` when color is not fully opaque.

**green**
Set or get green channel information.

The value ought to be a float between 0 and 1.

**Return type** `float` or `int`

**opaque**
Check if color is fully opaque.

**Return type** `bool`

Returns `True` if alpha component is exactly equal to 1, otherwise `False`.

```
>>> from_string('red').opaque
True
>>> from_string('transparent').opaque
False
```

**r**
Convenience synonym for `red`.

**red**
Set or get red channel information.

The value ought to be a float between 0 and 1.

**Return type** `float` or `int`

**set_rgb**(*r*, *g*, *b*)
Set red, green and blue components all at once.

**Parameters**

- **r** – red component
- **g** – green component
- **b** – blue component

Components should be numbers between 0 and 1. Alpha component remains unchanged.

**set_rgba**(*r*, *g*, *b*, *a*)
Set red, green, blue and alpha components all at once.

**Parameters**

- **r** – red component
- **g** – green component
- **b** – blue component
- **a** – alpha component

Components should be numbers between 0 and 1.

**set_string**(*value*)
Resets color value of this instance from string.

Usage and parameters identical to factory function `from_string()`.

**transparent**
Check if color is fully transparent.

**Return type** `bool`

Returns `True` if alpha component is exactly equal to 0, otherwise `False`.

```
>>> from_string('blue').transparent
False
>>> from_string('transparent').transparent
True
```

CHAPTER 5

# Indices and tables

- *genindex*
- *modindex*
- *search*

## p