pil

rhein verlag

```
from __future__ import print_function
```

```python
class Image:
    """
    This class represents an image object.  To create
    :py:class:`~PIL.Image.Image` objects, use the appropriate factory
    functions.  There’s hardly ever any reason to call the Image constructor
    directly.

    * :py:func:`~PIL.Image.open`
    * :py:func:`~PIL.Image.new`
    * :py:func:`~PIL.Image.frombytes`
    """
    format = None
    format_description = None
```

```python
def load(self):
    """
    Allocates storage for the image and loads the pixel data.  In
    normal cases, you don’t need to call this method, since the
    Image class automatically loads an opened image when it is
    accessed for the first time. This method will close the file
    associated with the image.

    :returns: An image access object.
    """
    if self.im and self.palette and self.palette.dirty:
        # realize palette
        self.im.putpalette(*self.palette.getdata())
        self.palette.dirty = 0
        self.palette.mode = "RGB"
        self.palette.rawmode = None
        if "transparency" in self.info:
            if isinstance(self.info["transparency"], int):
                self.im.putpalettealpha(self.info["transparency"], 0)
            else:
                self.im.putpalettealphas(self.info["transparency"])
            self.palette.mode = "RGBA"

    if self.im:
        if HAS_CFFI and USE_CFFI_ACCESS:
            if self.pyaccess:
                return self.pyaccess
            from PIL import PyAccess
            self.pyaccess = PyAccess.new(self, self.readonly)
            if self.pyaccess:
                return self.pyaccess
        return self.im.pixel_access(self.readonly)
```

```
def close(self):
    """
    Closes the file pointer, if possible.

    This operation will destroy the image core and release its memory.
    The image data will be unusable afterward.

    This function is only required to close images that have not
    had their file read and closed by the
    :py:meth:`~PIL.Image.Image.load` method.
    """
    try:
        self.fp.close()
    except Exception as msg:
        if Image.DEBUG:
            print (“Error closing: %s” % msg)

    # Instead of simply setting to None, we’re setting up a
    # deferred error that will better explain that the core image
    # object is gone.
    self.im = deferred_error(ValueError(“Operation on closed image”))
```

```
def verify(self):
    “Check file integrity”

    # raise exception if something’s wrong.  must be called
    # directly after open, and closes file when finished.
    self.fp = None
```

```python
def copy(self):
    """
    Copies this image. Use this method if you wish to paste things
    into an image, but still retain the original.

    :rtype: :py:class:`~PIL.Image.Image`
    :returns: An :py:class:`~PIL.Image.Image` object.
    """
    self.load()
    im = self.im.copy()
    return self._new(im)
```

```python
def crop(self, box=None):
    """
    Returns a rectangular region from this image. The box is a
    4-tuple defining the left, upper, right, and lower pixel
    coordinate.

    This is a lazy operation.  Changes to the source image may or
    may not be reflected in the cropped image.  To break the
    connection, call the :py:meth:`~PIL.Image.Image.load` method on
    the cropped copy.

    :param box: The crop rectangle, as a (left, upper, right, lower)-tuple.
    :rtype: :py:class:`~PIL.Image.Image`
    :returns: An :py:class:`~PIL.Image.Image` object.

    self.load()
    if box is None:
        return self.copy()

    # lazy operation
    return _ImageCrop(self, box)
```

```
def draft(self, mode, size):
    """
    NYI

    Configures the image file loader so it returns a version of the
    image that as closely as possible matches the given mode and
    size.  For example, you can use this method to convert a color
    JPEG to greyscale while loading it, or to extract a 128x192
    version from a PCD file.

    Note that this method modifies the :py:class:`~PIL.Image.Image` object
    in place.  If the image has already been loaded, this method has no
    effect.

    :param mode: The requested mode.
    :param size: The requested size.
    """
    pass
```

```python
def getcolors(self, maxcolors=256):
    """
    Returns a list of colors used in this image.

    :param maxcolors: Maximum number of colors.  If this number is
       exceeded, this method returns None.  The default limit is
       256 colors.
    :returns: An unsorted list of (count, pixel) values.
    """

    self.load()
    if self.mode in ("1", "L", "P"):
        h = self.im.histogram()
        out = []
        for i in range(256):
            if h[i]:
                out.append((h[i], i))
        if len(out) > maxcolors:
            return None
        return out
    return self.im.getcolors(maxcolors)
```

```
def getdata(self, band=None):
    """
    Returns the contents of this image as a sequence object
    containing pixel values.  The sequence object is flattened, so
    that values for line one follow directly after the values of
    line zero, and so on.

    Note that the sequence object returned by this method is an
    internal PIL data type, which only supports certain sequence
    operations.  To convert it to an ordinary sequence (e.g. for
    printing), use **list(im.getdata())**.

    :param band: What band to return.  The default is to return
       all bands.  To return a single band, pass in the index
       value (e.g. 0 to get the "R" band from an "RGB" image).
    :returns: A sequence-like object.

    self.load()
    if band is not None:
        return self.im.getband(band)
    return self.im  # could be abused
```

```python
def getpalette(self):
    """
    Returns the image palette as a list.

    :returns: A list of color values [r, g, b, ...], or None if the
       image has no palette.
    """

    self.load()
    try:
        if bytes is str:
            return [i8(c) for c in self.im.getpalette()]
        else:
            return list(self.im.getpalette())
    except ValueError:
        return None  # no palette
```

```python
def histogram(self, mask=None, extrema=None):
    """
    Returns a histogram for the image. The histogram is returned as
    a list of pixel counts, one for each pixel value in the source
    image. If the image has more than one band, the histograms for
    all bands are concatenated (for example, the histogram for an
    "RGB" image contains 768 values).

    A bilevel image (mode "1") is treated as a greyscale ("L") image
    by this method.

    If a mask is provided, the method returns a histogram for those
    parts of the image where the mask image is non-zero. The mask
    image must have the same size as the image, and be either a
    bi-level image (mode "1") or a greyscale image ("L").

    :param mask: An optional mask.
    :returns: A list containing pixel counts.
    """
    self.load()
    if mask:
        mask.load()
        return self.im.histogram((0, 0), mask.im)
    if self.mode in ("I", "F"):
        if extrema is None:
            extrema = self.getextrema()
        return self.im.histogram(extrema)
    return self.im.histogram()
```

```
def point(self, lut, mode=None):
    """
    Maps this image through a lookup table or function.

    :param lut: A lookup table, containing 256 (or 65336 if
       self.mode=="I" and mode == "L") values per band in the
       image.  A function can be used instead, it should take a
       single argument. The function is called once for each
       possible pixel value, and the resulting table is applied to
       all bands of the image.
    :param mode: Output mode (default is same as input).  In the
       current version, this can only be used if the source image
       has mode "L" or "P", and the output has mode "1" or the
       source image mode is "I" and the output mode is "L".
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    self.load()

    if isinstance(lut, ImagePointHandler):
        return lut.point(self)

    if callable(lut):
        # if it isn’t a list, it should be a function
        if self.mode in ("I", "I;16", "F"):
            # check if the function can be used with point_transform
            # UNDONE wiredfool -- I think this prevents us from ever doing
            # a gamma function point transform on > 8bit images.
            scale, offset = _getscaleoffset(lut)
            return self._new(self.im.point_transform(scale, offset))
        # for other modes, convert the function to a table
        lut = [lut(i) for i in range(256)] * self.im.bands

    if self.mode == "F":
        # FIXME: _imaging returns a confusing error message for this case
        raise ValueError("point operation not supported for this mode")

    return self._new(self.im.point(lut, mode))
```

```python
def putdata(self, data, scale=1.0, offset=0.0):
    """
    Copies pixel data to this image.  This method copies data from a
    sequence object into the image, starting at the upper left
    corner (0, 0), and continuing until either the image or the
    sequence ends.  The scale and offset values are used to adjust
    the sequence values: **pixel = value*scale + offset**.

    :param data: A sequence object.
    :param scale: An optional scale value.  The default is 1.0.
    :param offset: An optional offset value.  The default is 0.0.
    """

    self.load()
    if self.readonly:
        self._copy()

    self.im.putdata(data, scale, offset)
```

```python
def putpalette(self, data, rawmode="RGB"):
    """
    Attaches a palette to this image.  The image must be a "P" or
    "L" image, and the palette sequence must contain 768 integer
    values, where each group of three values represent the red,
    green, and blue values for the corresponding pixel
    index. Instead of an integer sequence, you can use an 8-bit
    string.

    :param data: A palette sequence (either a list or a string).
    """
    from PIL import ImagePalette

    if self.mode not in ("L", "P"):
        raise ValueError("illegal image mode")
    self.load()
    if isinstance(data, ImagePalette.ImagePalette):
        palette = ImagePalette.raw(data.rawmode, data.palette)
    else:
        if not isinstance(data, bytes):
            if bytes is str:
                data = "".join(chr(x) for x in data)
            else:
                data = bytes(data)
        palette = ImagePalette.raw(rawmode, data)
    self.mode = "P"
    self.palette = palette
    self.palette.mode = "RGB"
    self.load()  # install new palette
```

```
def resize(self, size, resample=NEAREST):
    """
    Returns a resized copy of this image.

    :param size: The requested size in pixels, as a 2-tuple:
       (width, height).
    :param resample: An optional resampling filter.  This can be
       one of :py:attr:`PIL.Image.NEAREST` (use nearest neighbour),
       :py:attr:`PIL.Image.BILINEAR` (linear interpolation in a 2x2
       environment), :py:attr:`PIL.Image.BICUBIC` (cubic spline
       interpolation in a 4x4 environment), or
       :py:attr:`PIL.Image.ANTIALIAS` (a high-quality downsampling filter).
       If omitted, or if the image has mode "1" or "P", it is
       set :py:attr:`PIL.Image.NEAREST`.
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    if resample not in (NEAREST, BILINEAR, BICUBIC, ANTIALIAS):
        raise ValueError("unknown resampling filter")

    self.load()

    if self.mode in ("1", "P"):
        resample = NEAREST

    if self.mode == ‘RGBA’:
        return self.convert(‘RGBa’).resize(size, resample).convert(‘RGBA’)

    if resample == ANTIALIAS:
        # requires stretch support (imToolkit & PIL 1.1.3)
        try:
            im = self.im.stretch(size, resample)
        except AttributeError:
            raise ValueError("unsupported resampling filter")
    else:
        im = self.im.resize(size, resample)

    return self._new(im)
```

```
def rotate(self, angle, resample=NEAREST, expand=0):
    """
    Returns a rotated copy of this image.  This method returns a
    copy of this image, rotated the given number of degrees counter
    clockwise around its centre.

    :param angle: In degrees counter clockwise.
    :param resample: An optional resampling filter.  This can be
       one of :py:attr:`PIL.Image.NEAREST` (use nearest neighbour),
       :py:attr:`PIL.Image.BILINEAR` (linear interpolation in a 2x2
       environment), or :py:attr:`PIL.Image.BICUBIC`
       (cubic spline interpolation in a 4x4 environment).
       If omitted, or if the image has mode "1" or "P", it is
       set :py:attr:`PIL.Image.NEAREST`.
    :param expand: Optional expansion flag.  If true, expands the output
       image to make it large enough to hold the entire rotated image.
       If false or omitted, make the output image the same size as the
       input image.
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    if expand:
        import math
        angle = -angle * math.pi / 180
        matrix = [
            math.cos(angle), math.sin(angle), 0.0,
            -math.sin(angle), math.cos(angle), 0.0
            ]

        def transform(x, y, matrix=matrix):
            (a, b, c, d, e, f) = matrix
            return a*x + b*y + c, d*x + e*y + f

        # calculate output size
        w, h = self.size
        xx = []
        yy = []
        for x, y in ((0, 0), (w, 0), (w, h), (0, h)):
            x, y = transform(x, y)
            xx.append(x)
            yy.append(y)
        w = int(math.ceil(max(xx)) - math.floor(min(xx)))
        h = int(math.ceil(max(yy)) - math.floor(min(yy)))

        # adjust center
        x, y = transform(w / 2.0, h / 2.0)
        matrix[2] = self.size[0] / 2.0 - x
        matrix[5] = self.size[1] / 2.0 - y

        return self.transform((w, h), AFFINE, matrix, resample)

    if resample not in (NEAREST, BILINEAR, BICUBIC):
        raise ValueError("unknown resampling filter")

    self.load()

    if self.mode in ("1", "P"):
        resample = NEAREST

    return self._new(self.im.rotate(angle, resample))
```

```
def show(self, title=None, command=None):
    """
    Displays this image. This method is mainly intended for
    debugging purposes.

    On Unix platforms, this method saves the image to a temporary
    PPM file, and calls the **xv** utility.

    On Windows, it saves the image to a temporary BMP file, and uses
    the standard BMP display utility to show it (usually Paint).

    :param title: Optional title to use for the image window,
       where possible.
    :param command: command used to show the image
    """

    _show(self, title=title, command=command)
```

```
def split(self):
    """
    Split this image into individual bands. This method returns a
    tuple of individual image bands from an image. For example,
    splitting an "RGB" image creates three new images each
    containing a copy of one of the original bands (red, green,
    blue).

    :returns: A tuple containing bands.
    """

    self.load()
    if self.im.bands == 1:
        ims = [self.copy()]
    else:
        ims = []
        for i in range(self.im.bands):
            ims.append(self._new(self.im.getband(i)))
    return tuple(ims)
```

```
# FIXME: the different tranform methods need further explanation
# instead of bloating the method docs, add a separate chapter.
def transform(self, size, method, data=None, resample=NEAREST, fill=1):
    """
    Transforms this image.  This method creates a new image with the
    given size, and the same mode as the original, and copies data
    to the new image using the given transform.

    :param size: The output size.
    :param method: The transformation method.  This is one of
      :py:attr:`PIL.Image.EXTENT` (cut out a rectangular subregion),
      :py:attr:`PIL.Image.AFFINE` (affine transform),
      :py:attr:`PIL.Image.PERSPECTIVE` (perspective transform),
      :py:attr:`PIL.Image.QUAD` (map a quadrilateral to a rectangle), or
      :py:attr:`PIL.Image.MESH` (map a number of source quadrilaterals
      in one operation).
    :param data: Extra data to the transformation method.
    :param resample: Optional resampling filter.  It can be one of
       :py:attr:`PIL.Image.NEAREST` (use nearest neighbour),
       :py:attr:`PIL.Image.BILINEAR` (linear interpolation in a 2x2
       environment), or :py:attr:`PIL.Image.BICUBIC` (cubic spline
       interpolation in a 4x4 environment). If omitted, or if the image
       has mode "1" or "P", it is set to :py:attr:`PIL.Image.NEAREST`.
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    if self.mode == ‘RGBA’:
        return self.convert(‘RGBa’).transform(
            size, method, data, resample, fill).convert(‘RGBA’)

    if isinstance(method, ImageTransformHandler):
        return method.transform(size, self, resample=resample, fill=fill)
    if hasattr(method, "getdata"):
        # compatibility w. old-style transform objects
        method, data = method.getdata()
    if data is None:
        raise ValueError("missing method data")

    im = new(self.mode, size, None)
    if method == MESH:
        # list of quads
        for box, quad in data:
            im.__transformer(box, self, QUAD, quad, resample, fill)
    else:
        im.__transformer((0, 0)+size, self, method, data, resample, fill)

    return im
```

```
def _wedge():
    "Create greyscale wedge (for debugging only)"

    return Image()._new(core.wedge("L"))
```

```
def new(mode, size, color=0):
    """
    Creates a new image with the given mode and size.

    :param mode: The mode to use for the new image.
    :param size: A 2-tuple, containing (width, height) in pixels.
    :param color: What color to use for the image.  Default is black.
       If given, this should be a single integer or floating point value
       for single-band modes, and a tuple for multi-band modes (one value
       per band).  When creating RGB images, you can also use color
       strings as supported by the ImageColor module.  If the color is
       None, the image is not initialised.
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    if color is None:
        # don't initialize
        return Image()._new(core.new(mode, size))

    if isStringType(color):
        # css3-style specifier

        from PIL import ImageColor
        color = ImageColor.getcolor(color, mode)

    return Image()._new(core.fill(mode, size, color))
```

```
def open(fp, mode="r"):
    """
    Opens and identifies the given image file.

    This is a lazy operation; this function identifies the file, but
    the file remains open and the actual image data is not read from
    the file until you try to process the data (or call the
    :py:meth:`~PIL.Image.Image.load` method).  See
    :py:func:`~PIL.Image.new`.

    :param file: A filename (string) or a file object.  The file object
       must implement :py:meth:`~file.read`, :py:meth:`~file.seek`, and
       :py:meth:`~file.tell` methods, and be opened in binary mode.
    :param mode: The mode.  If given, this argument must be "r".
    :returns: An :py:class:`~PIL.Image.Image` object.
    :exception IOError: If the file cannot be found, or the image cannot be
       opened and identified.
    """

    if mode != "r":
        raise ValueError("bad mode %r" % mode)

    if isPath(fp):
        filename = fp
        fp = builtins.open(fp, "rb")
    else:
        filename = ""

    prefix = fp.read(16)

    preinit()

    for i in ID:
        try:
            factory, accept = OPEN[i]
            if not accept or accept(prefix):
                fp.seek(0)
                return factory(fp, filename)
        except (SyntaxError, IndexError, TypeError):
            # import traceback
            # traceback.print_exc()
            pass

    if init():

        for i in ID:
            try:
                factory, accept = OPEN[i]
                if not accept or accept(prefix):
                    fp.seek(0)
                    return factory(fp, filename)
            except (SyntaxError, IndexError, TypeError):
                # import traceback
                # traceback.print_exc()
                pass

    raise IOError("cannot identify image file %r"
                  % (filename if filename else fp))
```

```python
def blend(im1, im2, alpha):
    """
    Creates a new image by interpolating between two input images, using
    a constant alpha.::

        out = image1 * (1.0 - alpha) + image2 * alpha

    :param im1: The first image.
    :param im2: The second image.  Must have the same mode and size as
       the first image.
    :param alpha: The interpolation alpha factor.  If alpha is 0.0, a
       copy of the first image is returned. If alpha is 1.0, a copy of
       the second image is returned. There are no restrictions on the
       alpha value. If necessary, the result is clipped to fit into
       the allowed output range.
    :returns: An :py:class:`~PIL.Image.Image` object.
    """

    im1.load()
    im2.load()
    return im1._new(core.blend(im1.im, im2.im, alpha))
```

```
def composite(image1, image2, mask):
    """
    Create composite image by blending images using a transparency mask.

    :param image1: The first image.
    :param image2: The second image.  Must have the same mode and
       size as the first image.
    :param mask: A mask image.  This image can can have mode
       "1", "L", or "RGBA", and must have the same size as the
       other two images.
    """

    image = image2.copy()
    image.paste(image1, None, mask)
    return image
```

```
def text(self, xy, text):
    """
    Draws text at the given position. You must use
    :py:meth:`~PIL.PSDraw.PSDraw.setfont` before calling this method.
    """
    text = "\\(".join(text.split("("))
    text = "\\)".join(text.split(")"))
    xy = xy + (text,)
    self.fp.write("%d %d M (%s) S\n" % xy)
```

```python
def image(self, box, im, dpi = None):
    """Draw a PIL image, centered in the given box."""
    # default resolution depends on mode
    if not dpi:
        if im.mode == "1":
            dpi = 200 # fax
        else:
            dpi = 100 # greyscale
    # image size (on paper)
    x = float(im.size[0] * 72) / dpi
    y = float(im.size[1] * 72) / dpi
    # max allowed size
    xmax = float(box[2] - box[0])
    ymax = float(box[3] - box[1])
    if x > xmax:
        y = y * xmax / x; x = xmax
    if y > ymax:
        x = x * ymax / y; y = ymax
    dx = (xmax - x) / 2 + box[0]
    dy = (ymax - y) / 2 + box[1]
    self.fp.write("gsave\n%f %f translate\n" % (dx, dy))
    if (x, y) != im.size:
        # EpsImagePlugin._save prints the image at (0,0,xsize,ysize)
        sx = x / im.size[0]
        sy = y / im.size[1]
        self.fp.write("%f %f scale\n" % (sx, sy))
    EpsImagePlugin._save(im, self.fp, None, 0)
    self.fp.write("\ngrestore\n")
```

```
import numbers
```

```python
def setink(self, ink):
    # compatibility
    if warnings:
        warnings.warn(
            "’setink’ is deprecated; use keyword arguments instead",
            DeprecationWarning, stacklevel=2
            )
    if isStringType(ink):
        ink = ImageColor.getcolor(ink, self.mode)
    if self.palette and not isinstance(ink, numbers.Number):
        ink = self.palette.getcolor(ink)
    self.ink = self.draw.draw_ink(ink, self.mode)
```

```
def setfill(self, onoff):
    # compatibility
    if warnings:
        warnings.warn(
            "’setfill’ is deprecated; use keyword arguments instead",
            DeprecationWarning, stacklevel=2
            )
    self.fill = onoff
```

```
##
# Draw an arc.

def arc(self, xy, start, end, fill=None):
    ink, fill = self._getink(fill)
    if ink is not None:
        self.draw.draw_arc(xy, start, end, ink)
```

```
##
# Draw a bitmap.

def bitmap(self, xy, bitmap, fill=None):
    bitmap.load()
    ink, fill = self._getink(fill)
    if ink is None:
        ink = fill
    if ink is not None:
        self.draw.draw_bitmap(xy, bitmap.im, ink)
```

```
##
# Draw a chord.

def chord(self, xy, start, end, fill=None, outline=None):
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_chord(xy, start, end, fill, 1)
    if ink is not None:
        self.draw.draw_chord(xy, start, end, ink, 0)
```

```
##
# Draw an ellipse.

def ellipse(self, xy, fill=None, outline=None):
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_ellipse(xy, fill, 1)
    if ink is not None:
        self.draw.draw_ellipse(xy, ink, 0)
```

```
##
# Draw a line, or a connected sequence of line segments.

def line(self, xy, fill=None, width=0):
    ink, fill = self._getink(fill)
    if ink is not None:
        self.draw.draw_lines(xy, ink, width)
```

```
##
# (Experimental) Draw a shape.

def shape(self, shape, fill=None, outline=None):
    # experimental
    shape.close()
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_outline(shape, fill, 1)
    if ink is not None:
        self.draw.draw_outline(shape, ink, 0)
```

```
##
# Draw a pieslice.

def pieslice(self, xy, start, end, fill=None, outline=None):
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_pieslice(xy, start, end, fill, 1)
    if ink is not None:
        self.draw.draw_pieslice(xy, start, end, ink, 0)
```

```
##
# Draw one or more individual pixels.

def point(self, xy, fill=None):
    ink, fill = self._getink(fill)
    if ink is not None:
        self.draw.draw_points(xy, ink)
```

```
##
# Draw a polygon.

def polygon(self, xy, fill=None, outline=None):
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_polygon(xy, fill, 1)
    if ink is not None:
        self.draw.draw_polygon(xy, ink, 0)
```

```
##
# Draw a rectangle.

def rectangle(self, xy, fill=None, outline=None):
    ink, fill = self._getink(outline, fill)
    if fill is not None:
        self.draw.draw_rectangle(xy, fill, 1)
    if ink is not None:
        self.draw.draw_rectangle(xy, ink, 0)
```

```python
class Pen:
    def __init__(self, color, width=1, opacity=255):
        self.color = ImageColor.getrgb(color)
        self.width = width
```

```python
class Brush:
    def __init__(self, color, opacity=255):
        self.color = ImageColor.getrgb(color)
```

```
def flush(self):
    return self.image
```

```python
def constant(image, value):
    """Fill a channel with a given grey level.

    :rtype: :py:class:`~PIL.Image.Image`
    """

    return Image.new("L", image.size, value)
```

```python
def duplicate(image):
    """Copy a channel. Alias for :py:meth:`PIL.Image.Image.copy`.

    :rtype: :py:class:`~PIL.Image.Image`
    """

    return image.copy()
```

```python
def invert(image):
    """
    Invert an image (channel).

    .. code-block:: python

        out = MAX - image

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image.load()
    return image._new(image.im.chop_invert())
```

```python
def lighter(image1, image2):
    """
    Compares the two images, pixel by pixel, and returns a new image containing
    the lighter values.

    .. code-block:: python

        out = max(image1, image2)

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_lighter(image2.im))
```

```python
def darker(image1, image2):
    """
    Compares the two images, pixel by pixel, and returns a new image
    containing the darker values.

    .. code-block:: python

        out = min(image1, image2)

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_darker(image2.im))
```

```python
def difference(image1, image2):
    """
    Returns the absolute value of the pixel-by-pixel difference between the two
    images.

    .. code-block:: python

        out = abs(image1 - image2)

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_difference(image2.im))
```

```python
def multiply(image1, image2):
    """
    Superimposes two images on top of each other.

    If you multiply an image with a solid black image, the result is black. If
    you multiply with a solid white image, the image is unaffected.

    .. code-block:: python

        out = image1 * image2 / MAX

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_multiply(image2.im))
```

```python
def add(image1, image2, scale=1.0, offset=0):
    """
    Adds two images, dividing the result by scale and adding the
    offset. If omitted, scale defaults to 1.0, and offset to 0.0.

    .. code-block:: python

        out = ((image1 + image2) / scale + offset)

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_add(image2.im, scale, offset))
```

```python
def subtract(image1, image2, scale=1.0, offset=0):
    """
    Subtracts two images, dividing the result by scale and adding the
    offset. If omitted, scale defaults to 1.0, and offset to 0.0.

    .. code-block:: python

        out = ((image1 - image2) / scale + offset)

    :rtype: :py:class:`~PIL.Image.Image`
    """

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_subtract(image2.im, scale, offset))
```

```
def logical_and(image1, image2):
    “””Logical AND between two images.

    .. code-block:: python

        out = ((image1 and image2) % MAX)

    :rtype: :py:class:`~PIL.Image.Image`
    ”””

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_and(image2.im))
```

```python
def logical_or(image1, image2):
    “””Logical OR between two images.

    .. code-block:: python

        out = ((image1 or image2) % MAX)

    :rtype: :py:class:`~PIL.Image.Image`
    ”””

    image1.load()
    image2.load()
    return image1._new(image1.im.chop_or(image2.im))
```

```
class Color(_Enhance):
    """Adjust image color balance.

    This class can be used to adjust the colour balance of an image, in
    a manner similar to the controls on a colour TV set. An enhancement
    factor of 0.0 gives a black and white image. A factor of 1.0 gives
    the original image.
    """
    def __init__(self, image):
        self.image = image
        self.degenerate = image.convert("L").convert(image.mode)
```

```python
class Contrast(_Enhance):
    """Adjust image contrast.

    This class can be used to control the contrast of an image, similar
    to the contrast control on a TV set. An enhancement factor of 0.0
    gives a solid grey image. A factor of 1.0 gives the original image.
    """
    def __init__(self, image):
        self.image = image
        mean = int(ImageStat.Stat(image.convert("L")).mean[0] + 0.5)
        self.degenerate = Image.new("L", image.size, mean).convert(image.mode)
```

```python
class Brightness(_Enhance):
    """Adjust image brightness.

    This class can be used to control the brighntess of an image.  An
    enhancement factor of 0.0 gives a black image. A factor of 1.0 gives the
    original image.
    """
    def __init__(self, image):
        self.image = image
        self.degenerate = Image.new(image.mode, image.size, 0)
```

```
def __init__(self):
    """
    Constructor for OleMetadata
    All attributes are set to None by default
    """
    # properties from SummaryInformation stream
    self.codepage = None
    self.title = None
    self.subject = None
    self.author = None
    self.keywords = None
    self.comments = None
    self.template = None
    self.last_saved_by = None
    self.revision_number = None
    self.total_edit_time = None
    self.last_printed = None
    self.create_time = None
    self.last_saved_time = None
    self.num_pages = None
    self.num_words = None
    self.num_chars = None
    self.thumbnail = None
    self.creating_application = None
    self.security = None
    # properties from DocumentSummaryInformation stream
    self.codepage_doc = None
    self.category = None
    self.presentation_target = None
    self.bytes = None
    self.lines = None
    self.paragraphs = None
    self.slides = None
    self.notes = None
    self.hidden_slides = None
    self.mm_clips = None
    self.scale_crop = None
    self.heading_pairs = None
    self.titles_of_parts = None
    self.manager = None
    self.company = None
    self.links_dirty = None
    self.chars_with_spaces = None
    self.unused = None
    self.shared_doc = None
    self.link_base = None
    self.hlinks = None
    self.hlinks_changed = None
    self.version = None
    self.dig_sig = None
    self.content_type = None
    self.content_status = None
    self.language = None
    self.doc_version = None
```

```python
def colorize(image, black, white):
    """
    Colorize grayscale image.  The **black** and **white**
    arguments should be RGB tuples; this function calculates a color
    wedge mapping all black pixels in the source image to the first
    color, and all white pixels to the second color.

    :param image: The image to colorize.
    :param black: The color to use for black input pixels.
    :param white: The color to use for white input pixels.
    :return: An image.
    """
    assert image.mode == "L"
    black = _color(black, "RGB")
    white = _color(white, "RGB")
    red = []; green = []; blue = []
    for i in range(256):
        red.append(black[0]+i*(white[0]-black[0])//255)
        green.append(black[1]+i*(white[1]-black[1])//255)
        blue.append(black[2]+i*(white[2]-black[2])//255)
    image = image.convert("RGB")
    return _lut(image, red + green + blue)
```

```
def crop(image, border=0):
    """
    Remove border from image.  The same amount of pixels are removed
    from all four sides.  This function works on all image modes.

    .. seealso:: :py:meth:`~PIL.Image.Image.crop`

    :param image: The image to crop.
    :param border: The number of pixels to remove.
    :return: An image.
    """
    left, top, right, bottom = _border(border)
    return image.crop(
        (left, top, image.size[0]-right, image.size[1]-bottom)
        )
```

```python
def deform(image, deformer, resample=Image.BILINEAR):
    """
    Deform the image.

    :param image: The image to deform.
    :param deformer: A deformer object.  Any object that implements a
                    **getmesh** method can be used.
    :param resample: What resampling filter to use.
    :return: An image.
    """
    return image.transform(
        image.size, Image.MESH, deformer.getmesh(image), resample
        )
```

```python
def equalize(image, mask=None):
    """
    Equalize the image histogram. This function applies a non-linear
    mapping to the input image, in order to create a uniform
    distribution of grayscale values in the output image.

    :param image: The image to equalize.
    :param mask: An optional mask.  If given, only the pixels selected by
                 the mask are included in the analysis.
    :return: An image.
    """
    if image.mode == "P":
        image = image.convert("RGB")
    h = image.histogram(mask)
    lut = []
    for b in range(0, len(h), 256):
        histo = [_f for _f in h[b:b+256] if _f]
        if len(histo) <= 1:
            lut.extend(list(range(256)))
        else:
            step = (reduce(operator.add, histo) - histo[-1]) // 255
            if not step:
                lut.extend(list(range(256)))
            else:
                n = step // 2
                for i in range(256):
                    lut.append(n // step)
                    n = n + h[i+b]
    return _lut(image, lut)
```

```python
def expand(image, border=0, fill=0):
    """
    Add border to the image

    :param image: The image to expand.
    :param border: Border width, in pixels.
    :param fill: Pixel fill value (a color value).  Default is 0 (black).
    :return: An image.
    """
    "Add border to image"
    left, top, right, bottom = _border(border)
    width = left + image.size[0] + right
    height = top + image.size[1] + bottom
    out = Image.new(image.mode, (width, height), _color(fill, image.mode))
    out.paste(image, (left, top))
    return out
```

```python
def flip(image):
    """
    Flip the image vertically (top to bottom).

    :param image: The image to flip.
    :return: An image.
    """
    return image.transpose(Image.FLIP_TOP_BOTTOM)
```

```
def grayscale(image):
    """
    Convert the image to grayscale.

    :param image: The image to convert.
    :return: An image.
    """
    return image.convert("L")
```

```python
def invert(image):
    """
    Invert (negate) the image.

    :param image: The image to invert.
    :return: An image.
    """
    lut = []
    for i in range(256):
        lut.append(255-i)
    return _lut(image, lut)
```

```
def mirror(image):
    """
    Flip image horizontally (left to right).

    :param image: The image to mirror.
    :return: An image.
    """
    return image.transpose(Image.FLIP_LEFT_RIGHT)
```

```python
def posterize(image, bits):
    """
    Reduce the number of bits for each color channel.

    :param image: The image to posterize.
    :param bits: The number of bits to keep for each channel (1-8).
    :return: An image.
    """
    lut = []
    mask = ~(2**(8-bits)-1)
    for i in range(256):
        lut.append(i & mask)
    return _lut(image, lut)
```

```python
def getcolor(self, color):
    """Given an rgb tuple, allocate palette entry.

    .. warning:: This method is experimental.
    """
    if self.rawmode:
        raise ValueError("palette contains raw palette data")
    if isinstance(color, tuple):
        try:
            return self.colors[color]
        except KeyError:
            # allocate new color slot
            if isinstance(self.palette, bytes):
                self.palette = [int(x) for x in self.palette]
            index = len(self.colors)
            if index >= 256:
                raise ValueError("cannot allocate more than 256 colors")
            self.colors[color] = index
            self.palette[index] = color[0]
            self.palette[index+256] = color[1]
            self.palette[index+512] = color[2]
            self.dirty = 1
            return index
    else:
        raise ValueError("unknown color specifier: %r" % color)
```

```python
def save(self, fp):
    """Save palette to text file.

    .. warning:: This method is experimental.
    """
    if self.rawmode:
        raise ValueError("palette contains raw palette data")
    if isinstance(fp, str):
        fp = open(fp, "w")
    fp.write("# Palette\n")
    fp.write("# Mode: %s\n" % self.mode)
    for i in range(256):
        fp.write("%d" % i)
        for j in range(i, len(self.palette), 256):
            fp.write(" %d" % self.palette[j])
        fp.write("\n")
    fp.close()
```

```
def dump(self):
    ”””
    Dump all metadata, for debugging purposes.
    ”””
    print(‘Properties from SummaryInformation stream:’)
    for prop in self.SUMMARY_ATTRIBS:
        value = getattr(self, prop)
        print(‘- %s: %s’ % (prop, repr(value)))
    print(‘Properties from DocumentSummaryInformation stream:’)
    for prop in self.DOCSUM_ATTRIBS:
        value = getattr(self, prop)
        print(‘- %s: %s’ % (prop, repr(value)))
```

```
ERRORS = {
    -1: "image buffer overrun error",
    -2: "decoding error",
    -3: "unknown error",
    -8: "bad configuration",
    -9: "out of memory error"
}
```

pil

auflage 100 exemplare
isbn 978-3-944574-35-6

autor david kühne
cc by-nc-nd

druck & bindung rhein verlag
schrift bembo std
papier munken cream print 115g/m^2

anschrift rhein verlag, 2014
birkenstraße 99
hinterhof
40233 düsseldorf

website http://www.rhein-verlag.com

9 783944 574356

RV