`Image` Module

The `Image` module provides a class with the same name which is used to represent a PIL image. The module also provides a number of factory functions, including functions to load images from files, and to create new images.

Examples

Open, rotate, and display an image (using the default viewer)

The following script loads an image, rotates it 45 degrees, and displays it using an external viewer (usually xv on Unix, and the Paint program on Windows).

```
from PIL import Image
with Image.open("hopper.jpg") as im:
    im.rotate(45).show()
```

Create thumbnails

The following script creates nice thumbnails of all JPEG images in the current directory preserving aspect ratios with 128x128 max resolution.

```
from PIL import Image
import glob, os

size = 128, 128

for infile in glob.glob("*.jpg"):
    file, ext = os.path.splitext(infile)
    with Image.open(infile) as im:
        im.thumbnail(size)
        im.save(file + ".thumbnail", "JPEG")
```

Functions

```
PIL.Image.open(fp: StrOrBytesPath | IO[bytes], mode: Literal['r'] = 'r', formats:
    list[str] | tuple[str, ...] | None = None) → ImageFile.ImageFile          [source]
```

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 `load()` method). See `new()`. See File Handling in Pillow.

PARAMETERS:

- **fp** – A filename (string), os.PathLike object or a file object. The file object must implement `file.read`, `file.seek`, and `file.tell` methods, and be opened in binary mode. The file object will also seek to zero before reading.
- **mode** – The mode. If given, this argument must be “r”.
- **formats** – A list or tuple of formats to attempt to load the file in. This can be used to restrict the set of formats checked. Pass `None` to try all supported formats. You can print the set of available formats by running `python3 -m PIL` or using the `PIL.features.pilinfo()` function.

RETURNS:

An `Image` object.

RAISES:

- **FileNotFoundError** – If the file cannot be found.
- **PIL.UnidentifiedImageError** – If the image cannot be opened and identified.
- **ValueError** – If the `mode` is not “r”, or if a `StringIO` instance is used for `fp`.
- **TypeError** – If `formats` is not `None`, a list or a tuple.

Warning

To protect against potential DOS attacks caused by “decompression bombs” (i.e. malicious files which decompress into a huge amount of data and are designed to crash or cause disruption by using up a lot of memory), Pillow will issue a `DecompressionBombWarning` if the number of pixels in an image is over a certain limit, `MAX_IMAGE_PIXELS`.

This threshold can be changed by setting `MAX_IMAGE_PIXELS`. It can be disabled by setting `Image.MAX_IMAGE_PIXELS = None`.

If desired, the warning can be turned into an error with `warnings.simplefilter('error', Image.DecompressionBombWarning)` or suppressed entirely with `warnings.simplefilter('ignore', Image.DecompressionBombWarning)`. See also the logging documentation to have warnings output to the logging facility instead of stderr.

If the number of pixels is greater than twice `MAX_IMAGE_PIXELS`, then a `DecompressionBombError` will be raised instead.

Image processing

`PIL.Image.alpha_composite(im1: Image, im2: Image) → Image` [source]

Alpha composite im2 over im1.

PARAMETERS:

- **im1** – The first image. Must have mode RGBA.
- **im2** – The second image. Must have mode RGBA, and the same size as the first image.

RETURNS:

An `Image` object.

`PIL.Image.blend(im1: Image, im2: Image, alpha: float) → Image` [source]

Creates a new image by interpolating between two input images, using a constant alpha:

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

PARAMETERS:

- **im1** – The first image.
- **im2** – The second image. Must have the same mode and size as the first image.
- **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 `Image` object.

`PIL.Image.composite(image1: Image, image2: Image, mask: Image) → Image` [source]

Create composite image by blending images using a transparency mask.

PARAMETERS:

- **image1** – The first image.
- **image2** – The second image. Must have the same mode and size as the first image.
- **mask** – A mask image. This image can have mode “1”, “L”, or “RGBA”, and must have the same size as the other two images.

```
PIL.Image.eval(image: Image, *args: Callable[[int], float]) → Image          [source]
```

Applies the function (which should take one argument) to each pixel in the given image. If the image has more than one band, the same function is applied to each band. Note that the function is evaluated once for each possible pixel value, so you cannot use random components or other generators.

PARAMETERS:

- **image** – The input image.
- **function** – A function object, taking one integer argument.

RETURNS:

An `Image` object.

```
PIL.Image.merge(mode: str, bands: Sequence[Image]) → Image                   [source]
```

Merge a set of single band images into a new multiband image.

PARAMETERS:

- **mode** – The mode to use for the output image. See: Modes.
- **bands** – A sequence containing one single-band image for each band in the output image. All bands must have the same size.

RETURNS:

An `Image` object.

Constructing images

```
PIL.Image.new(mode: str, size: tuple[int, int] | list[int], color: float |
    tuple[float, ...] | str | None = 0) → Image                    [source]
```

Creates a new image with the given mode and size.

PARAMETERS:

- **mode** – The mode to use for the new image. See: Modes.
- **size** – A 2-tuple, containing (width, height) in pixels.
- **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 or HSV 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 `Image` object.

```
PIL.Image.fromarray(obj: SupportsArrayInterface, mode: str | None = None) → Image
```

[source]

Creates an image memory from an object exporting the array interface (using the buffer protocol):

```
from PIL import Image
import numpy as np
a = np.zeros((5, 5))
im = Image.fromarray(a)
```

If `obj` is not contiguous, then the `tobytes` method is called and `frombuffer()` is used.

In the case of NumPy, be aware that Pillow modes do not always correspond to NumPy dtypes. Pillow modes only offer 1-bit pixels, 8-bit pixels, 32-bit signed integer pixels, and 32-bit floating point pixels.

Pillow images can also be converted to arrays:

```
from PIL import Image
import numpy as np
im = Image.open("hopper.jpg")
a = np.asarray(im)
```

When converting Pillow images to arrays however, only pixel values are transferred. This means that P and PA mode images will lose their palette.

PARAMETERS:

- **obj** – Object with array interface
- **mode** –

  Optional mode to use when reading `obj`. Will be determined from type if `None`.

  This will not be used to convert the data after reading, but will be used to change how the data is read:

  ```
  from PIL import Image
  import numpy as np
  a = np.full((1, 1), 300)
  im = Image.fromarray(a, mode="L")
  im.getpixel((0, 0))  # 44
  im = Image.fromarray(a, mode="RGB")
  im.getpixel((0, 0))  # (44, 1, 0)
  ```

  See: Modes for general information about modes.

RETURNS:

An image object.

Added in version 1.1.6.

PIL.Image.**frombytes**(mode: str, size: tuple[int, int], data: bytes | bytearray |

```
SupportsArrayInterface, decoder_name: str = 'raw', *args: Any) → Image    [source]
```

Creates a copy of an image memory from pixel data in a buffer.

In its simplest form, this function takes three arguments (mode, size, and unpacked pixel data).

You can also use any pixel decoder supported by PIL. For more information on available decoders, see the section Writing Your Own File Codec.

Note that this function decodes pixel data only, not entire images. If you have an entire image in a string, wrap it in a `BytesIO` object, and use `open()` to load it.

PARAMETERS:

- **mode** – The image mode. See: Modes.
- **size** – The image size.
- **data** – A byte buffer containing raw data for the given mode.
- **decoder_name** – What decoder to use.
- **args** – Additional parameters for the given decoder.

RETURNS:

An `Image` object.

```
PIL.Image.frombuffer(mode: str, size: tuple[int, int], data: bytes |
    SupportsArrayInterface, decoder_name: str = 'raw', *args: Any) → Image    [source]
```

Creates an image memory referencing pixel data in a byte buffer.

This function is similar to `frombytes()`, but uses data in the byte buffer, where possible. This means that changes to the original buffer object are reflected in this image). Not all modes can share memory; supported modes include “L”, “RGBX”, “RGBA”, and “CMYK”.

Note that this function decodes pixel data only, not entire images. If you have an entire image file in a string, wrap it in a `BytesIO` object, and use `open()` to load it.

The default parameters used for the “raw” decoder differs from that used for `frombytes()`. This is a bug, and will probably be fixed in a future release. The current release issues a warning if you do this; to disable the warning, you should provide the full set of parameters. See below for details.

PARAMETERS:

- **mode** – The image mode. See: Modes.
- **size** – The image size.
- **data** – A bytes or other buffer object containing raw data for the given mode.
- **decoder_name** – What decoder to use.
- **args** –

  Additional parameters for the given decoder. For the default encoder (“raw”), it’s recommended that you provide the full set of parameters:

  ```
  frombuffer(mode, size, data, "raw", mode, 0, 1)
  ```

RETURNS:

An `Image` object.

Added in version 1.1.4.

Generating images

```
PIL.Image.effect_mandelbrot(size: tuple[int, int], extent: tuple[float, float, float,
    float], quality: int) → Image                                              [source]
```

Generate a Mandelbrot set covering the given extent.

PARAMETERS:

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **extent** – The extent to cover, as a 4-tuple: (x0, y0, x1, y1).
- **quality** – Quality.

```
PIL.Image.effect_noise(size: tuple[int, int], sigma: float) → Image            [source]
```

Generate Gaussian noise centered around 128.

PARAMETERS:

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **sigma** – Standard deviation of noise.

```
PIL.Image.linear_gradient(mode: str) → Image                                   [source]
```

Generate 256x256 linear gradient from black to white, top to bottom.

PARAMETERS:

mode – Input mode.

```
PIL.Image.radial_gradient(mode: str) → Image                                   [source]
```

Generate 256x256 radial gradient from black to white, centre to edge.

PARAMETERS:

mode – Input mode.

Registering plugins

`PIL.Image.preinit() → None` [source]

Explicitly loads BMP, GIF, JPEG, PPM and PPM file format drivers.

It is called when opening or saving images.

`PIL.Image.init() → bool` [source]

Explicitly initializes the Python Imaging Library. This function loads all available file format drivers.

It is called when opening or saving images if `preinit()` is insufficient, and by `pilinfo()`.

> **Note**
>
> These functions are for use by plugin authors. They are called when a plugin is loaded as part of `preinit()` or `init()`. Application authors can ignore them.

```
PIL.Image.register_open(id: str, factory: Callable[[IO[bytes], str | bytes],
    ImageFile.ImageFile] | type[ImageFile.ImageFile], accept: Callable[[bytes], bool |
    str] | None = None) → None                                          [source]
```

Register an image file plugin. This function should not be used in application code.

PARAMETERS:

- **id** – An image format identifier.
- **factory** – An image file factory method.
- **accept** – An optional function that can be used to quickly reject images having another format.

`PIL.Image.register_mime(id: str, mimetype: str) → None` [source]

Registers an image MIME type by populating `Image.MIME`. This function should not be used in application code.

`Image.MIME` provides a mapping from image format identifiers to mime formats, but `get_format_mimetype()` can provide a different result for specific images.

PARAMETERS:

- **id** – An image format identifier.
- **mimetype** – The image MIME type for this format.

```
PIL.Image.register_save(id: str, driver: Callable[[Image, IO[bytes], str | bytes],
    None]) → None                                                    [source]
```

Registers an image save function. This function should not be used in application code.

PARAMETERS:

- **id** – An image format identifier.
- **driver** – A function to save images in this format.

```
PIL.Image.register_save_all(id: str, driver: Callable[[Image, IO[bytes], str | bytes],
    None]) → None                                                    [source]
```

Registers an image function to save all the frames of a multiframe format. This function should not be used in application code.

PARAMETERS:

- **id** – An image format identifier.
- **driver** – A function to save images in this format.

```
PIL.Image.register_extension(id: str, extension: str) → None         [source]
```

Registers an image extension. This function should not be used in application code.

PARAMETERS:

- **id** – An image format identifier.
- **extension** – An extension used for this format.

PIL.Image.**register_extensions**(id: str, extensions: list[str]) → None [source]

Registers image extensions. This function should not be used in application code.

PARAMETERS:

- **id** – An image format identifier.
- **extensions** – A list of extensions used for this format.

PIL.Image.**registered_extensions**() → dict[str, str] [source]

Returns a dictionary containing all file extensions belonging to registered plugins

PIL.Image.**register_decoder**(name: str, decoder: type[ImageFile.PyDecoder]) → None [source]

Registers an image decoder. This function should not be used in application code.

PARAMETERS:

- **name** – The name of the decoder
- **decoder** – An ImageFile.PyDecoder object

> Added in version 4.1.0.

PIL.Image.**register_encoder**(name: str, encoder: type[ImageFile.PyEncoder]) → None [source]

Registers an image encoder. This function should not be used in application code.

PARAMETERS:

- **name** – The name of the encoder
- **encoder** – An ImageFile.PyEncoder object

> Added in version 4.1.0.

The Image Class

```
class PIL.Image.Image                                                    [source]
```

This class represents an image object. To create `Image` objects, use the appropriate factory functions. There's hardly ever any reason to call the Image constructor directly.

- `open()`
- `new()`
- `frombytes()`

An instance of the `Image` class has the following methods. Unless otherwise stated, all methods return a new instance of the `Image` class, holding the resulting image.

```
Image.alpha_composite(im: Image, dest: Sequence[int] = (0, 0), source: Sequence[int] =
    (0, 0)) → None                                                       [source]
```

'In-place' analog of Image.alpha_composite. Composites an image onto this image.

PARAMETERS:

- **im** – image to composite over this one
- **dest** – Optional 2 tuple (left, top) specifying the upper left corner in this (destination) image.
- **source** – Optional 2 (left, top) tuple for the upper left corner in the overlay source image, or 4 tuple (left, top, right, bottom) for the bounds of the source rectangle

Performance Note: Not currently implemented in-place in the core layer.

```
Image.apply_transparency() → None                                        [source]
```

If a P mode image has a "transparency" key in the info dictionary, remove the key and instead apply the transparency to the palette. Otherwise, the image is unchanged.


```
Image.convert(mode: str | None = None, matrix: tuple[float, ...] | None = None,
    dither: Dither | None = None, palette: Palette = Palette.WEB, colors: int = 256) →
```

Image [source]

Returns a converted copy of this image. For the “P” mode, this method translates pixels through the palette. If mode is omitted, a mode is chosen so that all information in the image and the palette can be represented without a palette.

This supports all possible conversions between “L”, “RGB” and “CMYK”. The `matrix` argument only supports “L” and “RGB”.

When translating a color image to grayscale (mode “L”), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

The default method of converting a grayscale (“L”) or “RGB” image into a bilevel (mode “1”) image uses Floyd-Steinberg dither to approximate the original image luminosity levels. If dither is `None`, all values larger than 127 are set to 255 (white), all other values to 0 (black). To use other thresholds, use the `point()` method.

When converting from “RGBA” to “P” without a `matrix` argument, this passes the operation to `quantize()`, and `dither` and `palette` are ignored.

When converting from “PA”, if an “RGBA” palette is present, the alpha channel from the image will be used instead of the values from the palette.

PARAMETERS:

- **mode** – The requested mode. See: Modes.
- **matrix** – An optional conversion matrix. If given, this should be 4- or 12-tuple containing floating point values.
- **dither** – Dithering method, used when converting from mode “RGB” to “P” or from “RGB” or “L” to “1”. Available methods are `Dither.NONE` or `Dither.FLOYDSTEINBERG` (default). Note that this is not used when `matrix` is supplied.
- **palette** – Palette to use when converting from mode “RGB” to “P”. Available palettes are `Palette.WEB` or `Palette.ADAPTIVE`.
- **colors** – Number of colors to use for the `Palette.ADAPTIVE` palette. Defaults to 256.

RETURN TYPE:

`Image`

RETURNS:

An `Image` object.

The following example converts an RGB image (linearly calibrated according to ITU-R 709, using the D65 luminant) to the CIE XYZ color space:

```
rgb2xyz = (
    0.412453, 0.357580, 0.180423, 0,
    0.212671, 0.715160, 0.072169, 0,
    0.019334, 0.119193, 0.950227, 0)
out = im.convert("RGB", rgb2xyz)
```

`Image.copy() → Image` [source]

Copies this image. Use this method if you wish to paste things into an image, but still retain the original.

RETURN TYPE:

`Image`

RETURNS:

An `Image` object.

`Image.crop(box: tuple[float, float, float, float] | None = None) → Image` [source]

Returns a rectangular region from this image. The box is a 4-tuple defining the left, upper, right, and lower pixel coordinate. See Coordinate System.

Note: Prior to Pillow 3.4.0, this was a lazy operation.

PARAMETERS:

box – The crop rectangle, as a (left, upper, right, lower)-tuple.

RETURN TYPE:

`Image`

RETURNS:

An `Image` object.

s crops the input image with the provided coordinates:

```
from PIL import Image

with Image.open("hopper.jpg") as im:

    # The crop method from the Image module takes four coordinates as input.
    # The right can also be represented as (left+width)
    # and lower can be represented as (upper+height).
    (left, upper, right, lower) = (20, 20, 100, 100)

    # Here the image "im" is cropped and assigned to new variable im_crop
    im_crop = im.crop((left, upper, right, lower))
```

Image.**draft**(mode: str | None, size: tuple[int, int] | None) → tuple[str, tuple[int, int, float, float]] | None [source]

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 grayscale while loading it.

If any changes are made, returns a tuple with the chosen `mode` and `box` with coordinates of the original image within the altered one.

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

Note: This method is not implemented for most images. It is currently implemented only for JPEG and MPO images.

PARAMETERS:

- **mode** – The requested mode.
- **size** – The requested size in pixels, as a 2-tuple: (width, height).

ıge.**effect_spread**(distance: int) → Image [sc

Randomly spread pixels in an image.

PARAMETERS:

distance – Distance to spread pixels.

```
Image.entropy(mask: Image | None = None, extrema: tuple[float, float] | None = None) →
    float                                                                    [source]
```

Calculates and returns the entropy for the image.

A bilevel image (mode “1”) is treated as a grayscale (“L”) image by this method.

If a mask is provided, the method employs the 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 grayscale image (“L”).

PARAMETERS:

- **mask** – An optional mask.
- **extrema** – An optional tuple of manually-specified extrema.

RETURNS:

A float value representing the image entropy

```
Image.filter(filter: ImageFilter.Filter | type[ImageFilter.Filter]) → Image    [source]
```

Filters this image using the given filter. For a list of available filters, see the `ImageFilter` module.

PARAMETERS:

filter – Filter kernel.

RETURNS:

An `Image` object.

s blurs the input image using a filter from the `ImageFilter` module:

```
from PIL import Image, ImageFilter

with Image.open("hopper.jpg") as im:

    # Blur the input image using the filter ImageFilter.BLUR
    im_blurred = im.filter(filter=ImageFilter.BLUR)
```

```
Image.frombytes(data: bytes | bytearray | SupportsArrayInterface, decoder_name: str =
    'raw', *args: Any) → None                                                    [source]
```

Loads this image with pixel data from a bytes object.

This method is similar to the `frombytes()` function, but loads data into this image instead of creating a new image object.

```
Image.getbands() → tuple[str, ...]                                               [source]
```

Returns a tuple containing the name of each band in this image. For example, `getbands` on an RGB image returns (“R”, “G”, “B”).

RETURNS:

A tuple containing band names.

RETURN TYPE:

tuple

This helps to get the bands of the input image:

```
from PIL import Image

with Image.open("hopper.jpg") as im:
    print(im.getbands())  # Returns ('R', 'G', 'B')
```

Image.**getbbox**(*, alpha_only: bool = True) → tuple[int, int, int, int] | None [source]

Calculates the bounding box of the non-zero regions in the image.

PARAMETERS:

alpha_only – Optional flag, defaulting to `True`. If `True` and the image has an alpha channel, trim transparent pixels. Otherwise, trim pixels when all channels are zero. Keyword-only argument.

RETURNS:

The bounding box is returned as a 4-tuple defining the left, upper, right, and lower pixel coordinate. See Coordinate System. If the image is completely empty, this method returns None.

This helps to get the bounding box coordinates of the input image:

```
from PIL import Image

with Image.open("hopper.jpg") as im:
    print(im.getbbox())
    # Returns four coordinates in the format (left, upper, right, lower)
```

Image.**getchannel**(channel: int | str) → Image [source]

Returns an image containing a single channel of the source image.

PARAMETERS:

channel – What channel to return. Could be index (0 for “R” channel of “RGB”) or channel name (“A” for alpha channel of “RGBA”).

RETURNS:

An image in “L” mode.

Added in version 4.3.0.

Image.**getcolors**(maxcolors: int = 256) → list[tuple[int, tuple[int, ...]]] | list[tuple[int, float]] | None [source]

Returns a list of colors used in this image.

The colors will be in the image’s mode. For example, an RGB image will return a tuple of (red, green, blue) color values, and a P image will return the index of the color in the palette.

PARAMETERS:

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.

Image.**getdata**(band: int | None = None) → core.ImagingCore [source]

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())`.

PARAMETERS:

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.

```
Image.getexif() → Exif                                                    [source]
```

Gets EXIF data from the image.

RETURNS:

an `Exif` object.

```
Image.getextrema() → tuple[float, float] | tuple[tuple[int, int], ...]    [source]
```

Gets the minimum and maximum pixel values for each band in the image.

RETURNS:

For a single-band image, a 2-tuple containing the minimum and maximum pixel value. For a multi-band image, a tuple containing one 2-tuple for each band.

```
Image.getpalette(rawmode: str | None = 'RGB') → list[int] | None          [source]
```

Returns the image palette as a list.

PARAMETERS:

rawmode –

The mode in which to return the palette. `None` will return the palette in its current mode.

> Added in version 9.1.0.

RETURNS:

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

```
Image.getpixel(xy: tuple[int, int] | list[int]) → float | tuple[int, ...] | None
```

[source]

Returns the pixel value at a given position.

PARAMETERS:

xy – The coordinate, given as (x, y). See Coordinate System.

RETURNS:

The pixel value. If the image is a multi-layer image, this method returns a tuple.

```
Image.getprojection() → tuple[list[int], list[int]]
```

[source]

Get projection to x and y axes

RETURNS:

Two sequences, indicating where there are non-zero pixels along the X-axis and the Y-axis, respectively.

```
Image.getxmp() → dict[str, Any]
```

[source]

Returns a dictionary containing the XMP tags. Requires defusedxml to be installed.

RETURNS:

XMP tags in a dictionary.

```
Image.histogram(mask: Image | None = None, extrema: tuple[float, float] | None = None)
    → list[int]                                                                [source]
```

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. Counts are grouped into 256 bins for each band, even if the image has more than 8 bits per band. 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 grayscale (“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 grayscale image (“L”).

PARAMETERS:

- **mask** – An optional mask.
- **extrema** – An optional tuple of manually-specified extrema.

RETURNS:

A list containing pixel counts.

```
Image.paste(im: Image | str | float | tuple[float, ...], box: Image | tuple[int, int,
    int, int] | tuple[int, int] | None = None, mask: Image | None = None) → None
                                                                               [source]
```

Pastes another image into this image. The box argument is either a 2-tuple giving the upper left corner, a 4-tuple defining the left, upper, right, and lower pixel coordinate, or None (same as (0, 0)). See Coordinate System. If a 4-tuple is given, the size of the pasted image must match the size of the region.

If the modes don’t match, the pasted image is converted to the mode of this image (see the `convert()` method for details).

Instead of an image, the source can be a integer or tuple containing pixel values. The meth then fills the region with the given color. When creating RGB images, you can also use color strings as supported by the ImageColor module.

If a mask is given, this method updates only the regions indicated by the mask. You can use either “1”, “L”, “LA”, “RGBA” or “RGBa” images (if present, the alpha band is used as mask). Where the mask is 255, the given image is copied as is. Where the mask is 0, the current value is preserved. Intermediate values will mix the two images together, including their alpha channels if they have them.

See `alpha_composite()` if you want to combine images with respect to their alpha channels.

PARAMETERS:

- **im** – Source image or pixel value (integer, float or tuple).
- **box** –

  An optional 4-tuple giving the region to paste into. If a 2-tuple is used instead, it’s treated as the upper left corner. If omitted or None, the source is pasted into the upper left corner.

  If an image is given as the second argument and there is no third, the box defaults to (0, 0), and the second argument is interpreted as a mask image.

- **mask** – An optional mask image.

```
Image.point(lut: Sequence[float] | NumpyArray | Callable[[int], float] |
    Callable[[ImagePointTransform], ImagePointTransform | float] | ImagePointHandler,
    mode: str | None = None) → Image                                    [source]
```

Maps this image through a lookup table or function.

PARAMETERS:

- **lut** –

  A lookup table, containing 256 (or 65536 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.

  It may also be an `ImagePointHandler` object:

  ```
  class Example(Image.ImagePointHandler):
    def point(self, im: Image) -> Image:
      # Return result
  ```

- **mode** – Output mode (default is same as input). 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 `Image` object.

```
Image.putalpha(alpha: Image | int) → None                    [source]
```

Adds or replaces the alpha layer in this image. If the image does not have an alpha layer, it’s converted to “LA” or “RGBA”. The new layer must be either “L” or “1”.

PARAMETERS:

alpha – The new alpha layer. This can either be an “L” or “1” image having the same size as this image, or an integer.


```
age.putdata(data: Sequence[float] | Sequence[Sequence[int]] | core.ImagingCore
  NumpyArray, scale: float = 1.0, offset: float = 0.0) → None    [source]
```

Copies pixel data from a flattened sequence object into the image. The values should start at the upper left corner (0, 0), continue to the end of the line, followed directly by the first value of the second line, and so on. Data will be read until either the image or the sequence ends. The scale and offset values are used to adjust the sequence values: **pixel = value*scale + offset**.

PARAMETERS:

- **data** – A flattened sequence object.
- **scale** – An optional scale value. The default is 1.0.
- **offset** – An optional offset value. The default is 0.0.

```
Image.putpalette(data: ImagePalette.ImagePalette | bytes | Sequence[int], rawmode: str
    = 'RGB') → None                                                          [source]
```

Attaches a palette to this image. The image must be a “P”, “PA”, “L” or “LA” image.

The palette sequence must contain at most 256 colors, made up of one integer value for each channel in the raw mode. For example, if the raw mode is “RGB”, then it can contain at most 768 values, made up of red, green and blue values for the corresponding pixel index in the 256 colors. If the raw mode is “RGBA”, then it can contain at most 1024 values, containing red, green, blue and alpha values.

Alternatively, an 8-bit string may be used instead of an integer sequence.

PARAMETERS:

- **data** – A palette sequence (either a list or a string).
- **rawmode** – The raw mode of the palette. Either “RGB”, “RGBA”, or a mode that can be transformed to “RGB” or “RGBA” (e.g. “R”, “BGR;15”, “RGBA;L”).

```
Image.putpixel(xy: tuple[int, int], value: float | tuple[int, ...] | list[int]) → None
                                                                             [source]
```

Modifies the pixel at the given position. The color is given as a single numerical value for single-band images, and a tuple for multi-band images. In addition to this, RGB and RG tuples are accepted for P and PA images.

Note that this method is relatively slow. For more extensive changes, use `paste()` or the `ImageDraw` module instead.

See:

- `paste()`
- `putdata()`
- `ImageDraw`

PARAMETERS:

- **xy** – The pixel coordinate, given as (x, y). See Coordinate System.
- **value** – The pixel value.

```
Image.quantize(colors: int = 256, method: int | None = None, kmeans: int = 0, palette:
    Image | None = None, dither: Dither = Dither.FLOYDSTEINBERG) → Image      [source]
```

Convert the image to ‘P’ mode with the specified number of colors.

PARAMETERS:

- **colors** – The desired number of colors, <= 256
- **method** –

  `Quantize.MEDIANCUT` (median cut), `Quantize.MAXCOVERAGE` (maximum coverage), `Quantize.FASTOCTREE` (fast octree), `Quantize.LIBIMAGEQUANT` (libimagequant; check support using `PIL.features.check_feature()` with `feature="libimagequant"`).

  By default, `Quantize.MEDIANCUT` will be used.

  The exception to this is RGBA images. `Quantize.MEDIANCUT` and `Quantize.MAXCOVERAGE` do not support RGBA images, so `Quantize.FASTOCTREE` is used by default instead.

- **kmeans** – Integer greater than or equal to zero.
- **palette** – Quantize to the palette of given `PIL.Image.Image`.
- **dither** – Dithering method, used when converting from mode “RGB” to “P” or from “RGB” or “L” to “1”. Available methods are `Dither.NONE` or `Dither.FLOYDSTEINBERG` (default).

RETURNS:

A new image

```
Image.reduce(factor: int | tuple[int, int], box: tuple[int, int, int, int] | None =
    None) → Image                                                          [source]
```

Returns a copy of the image reduced `factor` times. If the size of the image is not dividable by `factor`, the resulting size will be rounded up.

PARAMETERS:

- **factor** – A greater than 0 integer or tuple of two integers for width and height separately.
- **box** – An optional 4-tuple of ints providing the source image region to be reduced. The values must be within `(0, 0, width, height)` rectangle. If omitted or `None`, the entire source is used.

```
Image.remap_palette(dest_map: list[int], source_palette: bytes | bytearray | None =
    None) → Image                                                          [source]
```

Rewrites the image to reorder the palette.

PARAMETERS:

- **dest_map** – A list of indexes into the original palette. e.g. `[1,0]` would swap a two item palette, and `list(range(256))` is the identity transform.
- **source_palette** – Bytes or None.

RETURNS:

An `Image` object.

```
Image.resize(size: tuple[int, int] | list[int] | NumpyArray, resample: int | None =
    None, box: tuple[float, float, float, float] | None = None, reducing_gap: float |
    None = None) → Image                                                   [source]
```

Returns a resized copy of this image.

PARAMETERS:

- **size** – The requested size in pixels, as a tuple or array: (width, height).
- **resample** – An optional resampling filter. This can be one of `Resampling.NEAREST`, `Resampling.BOX`, `Resampling.BILINEAR`, `Resampling.HAMMING`, `Resampling.BICUBIC` or `Resampling.LANCZOS`. If the image has mode “1” or “P”, it is always set to `Resampling.NEAREST`. If the image mode is “BGR;15”, “BGR;16” or “BGR;24”, then the default filter is `Resampling.NEAREST`. Otherwise, the default filter is `Resampling.BICUBIC`. See: Filters.
- **box** – An optional 4-tuple of floats providing the source image region to be scaled. The values must be within (0, 0, width, height) rectangle. If omitted or None, the entire source is used.
- **reducing_gap** – Apply optimization by resizing the image in two steps. First, reducing the image by integer times using `reduce()`. Second, resizing using regular resampling. The last step changes size no less than by `reducing_gap` times. `reducing_gap` may be None (no first step is performed) or should be greater than 1.0. The bigger `reducing_gap`, the closer the result to the fair resampling. The smaller `reducing_gap`, the faster resizing. With `reducing_gap` greater or equal to 3.0, the result is indistinguishable from fair resampling in most cases. The default value is None (no optimization).

RETURNS:

An `Image` object.

This resizes the given image from `(width, height)` to `(width/2, height/2)`:

```python
from PIL import Image

with Image.open("hopper.jpg") as im:

    # Provide the target width and height of the image
    (width, height) = (im.width // 2, im.height // 2)
    im_resized = im.resize((width, height))
```

Image.**rotate**(angle: float, resample: Resampling = Resampling.NEAREST, expand: int | bool = False, center: tuple[float, float] | None = None, translate: tuple[int, int] | None = None, fillcolor: float | tuple[float, ...] | str | None = None) → Image [source]

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.

PARAMETERS:

- **angle** – In degrees counter clockwise.
- **resample** – An optional resampling filter. This can be one of `Resampling.NEAREST` (use nearest neighbour), `Resampling.BILINEAR` (linear interpolation in a 2x2 environment), or `Resampling.BICUBIC` (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode “1” or “P”, it is set to `Resampling.NEAREST`. See Filters.
- **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. Note that the expand flag assumes rotation around the center and no translation.
- **center** – Optional center of rotation (a 2-tuple). Origin is the upper left corner. Default is the center of the image.
- **translate** – An optional post-rotate translation (a 2-tuple).
- **fillcolor** – An optional color for area outside the rotated image.

RETURNS:

An `Image` object.

This rotates the input image by `theta` degrees counter clockwise:

```python
from PIL import Image

with Image.open("hopper.jpg") as im:

    # Rotate the image by 60 degrees counter clockwise
    theta = 60
    # Angle is in degrees counter clockwise
    im_rotated = im.rotate(angle=theta)
```

Image.**save**(fp: StrOrBytesPath | IO[bytes], format: str | None = None, **params: Any) → None [source]

Saves this image under the given filename. If no format is specified, the format to use is determined from the filename extension, if possible.

Keyword options can be used to provide additional instructions to the writer. If a writer doesn't recognise an option, it is silently ignored. The available options are described in the image format documentation for each writer.

You can use a file object instead of a filename. In this case, you must always specify the format. The file object must implement the `seek`, `tell`, and `write` methods, and be opened in binary mode.

PARAMETERS:

- **fp** – A filename (string), os.PathLike object or file object.
- **format** – Optional format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter should always be used.
- **params** – Extra parameters to the image writer.

RETURNS:

None

RAISES:

- **ValueError** – If the output format could not be determined from the file name. Use the format option to solve this.
- **OSError** – If the file could not be written. The file may have been created, and may contain partial data.

```
Image.seek(frame: int) → None                                    [source]
```

Seeks to the given frame in this sequence file. If you seek beyond the end of the sequence, the method raises an `EOFError` exception. When a sequence file is opened, the library automatically seeks to frame 0.

See `tell()`.

If defined, `n_frames` refers to the number of available frames.

PARAMETERS:

frame – Frame number, starting at 0.

RAISES:

EOFError – If the call attempts to seek beyond the end of the sequence.

```
Image.show(title: str | None = None) → None
```
[source]

Displays this image. This method is mainly intended for debugging purposes.

This method calls `PIL.ImageShow.show()` internally. You can use `PIL.ImageShow.register()` to override its default behaviour.

The image is first saved to a temporary file. By default, it will be in PNG format.

On Unix, the image is then opened using the **xdg-open**, **display**, **gm**, **eog** or **xv** utility, depending on which one can be found.

On macOS, the image is opened with the native Preview application.

On Windows, the image is opened with the standard PNG display utility.

PARAMETERS:

title – Optional title to use for the image window, where possible.

```
Image.split() → tuple[Image, ...]
```
[source]

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).

If you need only one band, `getchannel()` method can be more convenient and faster.

RETURNS:

A tuple containing bands.

```
Image.tell() → int                                                    [source]
```

Returns the current frame number. See `seek()`.

If defined, `n_frames` refers to the number of available frames.

RETURNS:

Frame number, starting with 0.

```
Image.thumbnail(size: tuple[float, float], resample: Resampling = Resampling.BICUBIC,
    reducing_gap: float | None = 2.0) → None                          [source]
```

Make this image into a thumbnail. This method modifies the image to contain a thumbnail version of itself, no larger than the given size. This method calculates an appropriate thumbnail size to preserve the aspect of the image, calls the `draft()` method to configure the file reader (where applicable), and finally resizes the image.

Note that this function modifies the `Image` object in place. If you need to use the full resolution image as well, apply this method to a `copy()` of the original image.

PARAMETERS:

- **size** – The requested size in pixels, as a 2-tuple: (width, height).
- **resample** – Optional resampling filter. This can be one of `Resampling.NEAREST`, `Resampling.BOX`, `Resampling.BILINEAR`, `Resampling.HAMMING`, `Resampling.BICUBIC` or `Resampling.LANCZOS`. If omitted, it defaults to `Resampling.BICUBIC`. (was `Resampling.NEAREST` prior to version 2.5.0). See: Filters.
- **reducing_gap** – Apply optimization by resizing the image in two steps. First, reducing the image by integer times using `reduce()` or `draft()` for JPEG images. Second, resizing using regular resampling. The last step changes size no less than by `reducing_gap` times. `reducing_gap` may be None (no first step is performed) or should be greater than 1.0. The bigger `reducing_gap`, the closer the result to the fair resampling. The smaller `reducing_gap`, the faster resizing. With `reducing_gap` greater or equal to 3.0, the result is indistinguishable from fair resampling in most cases. The default value is 2.0 (very close to fair resampling while still being faster in many cases).

RETURNS:

None

```
Image.tobitmap(name: str = 'image') → bytes                    [source]
```

Returns the image converted to an X11 bitmap.

Note

This method only works for mode “1” images.

PARAMETERS:

name – The name prefix to use for the bitmap variables.

RETURNS:

A string containing an X11 bitmap.

RAISES:

ValueError – If the mode is not “1”

```
Image.tobytes(encoder_name: str = 'raw', *args: Any) → bytes
```
[source]

Return image as a bytes object.

Warning

This method returns the raw image data from the internal storage. For compressed image data (e.g. PNG, JPEG) use `save()`, with a BytesIO parameter for in-memory data.

PARAMETERS:

- **encoder_name** –

  What encoder to use. The default is to use the standard “raw” encoder.

  A list of C encoders can be seen under codecs section of the function array in `_imaging.c`. Python encoders are registered within the relevant plugins.

- **args** – Extra arguments to the encoder.

RETURNS:

A `bytes` object.


```
Image.transform(size: tuple[int, int], method: Transform | ImageTransformHandler |
    SupportsGetData, data: Sequence[Any] | None = None, resample: int =
```

```
Resampling.NEAREST, fill: int = 1, fillcolor: float | tuple[float, ...] | str |
None = None) → Image                                                    [source]
```

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.

PARAMETERS:

- **size** – The output size in pixels, as a 2-tuple: (width, height).
- **method** –

  The transformation method. This is one of `Transform.EXTENT` (cut out a rectangular subregion), `Transform.AFFINE` (affine transform), `Transform.PERSPECTIVE` (perspective transform), `Transform.QUAD` (map a quadrilateral to a rectangle), or `Transform.MESH` (map a number of source quadrilaterals in one operation).

  It may also be an `ImageTransformHandler` object:

  ```
  class Example(Image.ImageTransformHandler):
      def transform(self, size, data, resample, fill=1):
          # Return result
  ```

  Implementations of `ImageTransformHandler` for some of the `Transform` methods are provided in `ImageTransform`.

  It may also be an object with a `method.getdata` method that returns a tuple supplying new `method` and `data` values:

  ```
  class Example:
      def getdata(self):
          method = Image.Transform.EXTENT
          data = (0, 0, 100, 100)
          return method, data
  ```

- **data** – Extra data to the transformation method.

- **resample** – Optional resampling filter. It can be one of `Resampling.NEAREST` (use nearest neighbour), `Resampling.BILINEAR` (linear interpolation in a 2x2 environment), or `Resampling.BICUBIC` (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode “1” or “P”, it is set to `Resampling.NEAREST`. See: Filters.
- **fill** – If `method` is an `ImageTransformHandler` object, this is one of the arguments passed to it. Otherwise, it is unused.
- **fillcolor** – Optional fill color for the area outside the transform in the output image.

RETURNS:

An `Image` object.

```
Image.transpose(method: Transpose) → Image                    [source]
```

Transpose image (flip or rotate in 90 degree steps)

PARAMETERS:

method – One of `Transpose.FLIP_LEFT_RIGHT`, `Transpose.FLIP_TOP_BOTTOM`, `Transpose.ROTATE_90`, `Transpose.ROTATE_180`, `Transpose.ROTATE_270`, `Transpose.TRANSPOSE` or `Transpose.TRANSVERSE`.

RETURNS:

Returns a flipped or rotated copy of this image.

This flips the input image by using the `Transpose.FLIP_LEFT_RIGHT` method.

```python
from PIL import Image

with Image.open("hopper.jpg") as im:

    # Flip the image from left to right
    im_flipped = im.transpose(method=Image.Transpose.FLIP_LEFT_RIGHT)
    # To flip the image from top to bottom,
    # use the method "Image.Transpose.FLIP_TOP_BOTTOM"
```

Image.**verify**() → **None** [source]

Verifies the contents of a file. For data read from a file, this method attempts to determine if the file is broken, without actually decoding the image data. If this method finds any problems, it raises suitable exceptions. If you need to load the image after using this method, you must reopen the image file.

Image.**load**() → **core.PixelAccess** | **None** [source]

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.

If the file associated with the image was opened by Pillow, then this method will close it. The exception to this is if the image has multiple frames, in which case the file will be left open for seek operations. See File Handling in Pillow for more information.

RETURNS:

An image access object.

RETURN TYPE:

`PixelAccess`

age.**close**() → **None** [sc

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 required to close images that have multiple frames or have not had their file read and closed by the `load()` method. See File Handling in Pillow for more information.

Image Attributes

Instances of the `Image` class have the following attributes:

`Image.filename: str`

The filename or path of the source file. Only images created with the factory function `open` have a filename attribute. If the input is a file like object, the filename attribute is set to an empty string.

`Image.format: str | None`

The file format of the source file. For images created by the library itself (via a factory function, or by running a method on an existing image), this attribute is set to `None`.

`Image.mode: str`

Image mode. This is a string specifying the pixel format used by the image. Typical values are “1”, “L”, “RGB”, or “CMYK.” See Modes for a full list.

`Image.size: tuple[int]`

Image size, in pixels. The size is given as a 2-tuple (width, height).

`Image.width: int`

Image width, in pixels.

`Image.height: int`

Image height, in pixels.

`Image.palette: PIL.ImagePalette.ImagePalette | None`

Colour palette table, if any. If mode is “P” or “PA”, this should be an instance of the `ImagePa` class. Otherwise, it should be set to `None`.

`Image.info: dict`

A dictionary holding data associated with the image. This dictionary is used by file handlers to pass on various non-image information read from the file. See documentation for the various file handlers for details.

Most methods ignore the dictionary when returning new images; since the keys are not standardized, it's not possible for a method to know if the operation affects the dictionary. If you need the information later on, keep a reference to the info dictionary returned from the open method.

Unless noted elsewhere, this dictionary does not affect saving files.

`Image.is_animated: bool`

`True` if this image has more than one frame, or `False` otherwise.

This attribute is only defined by image plugins that support animated images. Plugins may leave this attribute undefined if they don't support loading animated images, even if the given format supports animated images.

Given that this attribute is not present for all images use `getattr(image, "is_animated", False)` to check if Pillow is aware of multiple frames in an image regardless of its format.

> See also
>
> `n_frames`, `seek()` and `tell()`

`Image.n_frames: int`

The number of frames in this image.

This attribute is only defined by image plugins that support animated images. Plugins may leave this attribute undefined if they don't support loading animated images, even if the given format supports animated images.

Given that this attribute is not present for all images use `getattr(image, "n_frames", 1)` to check the number of frames that Pillow is aware of in an image regardless of its format.

See also

`is_animated`, `seek()` and `tell()`

Image.has_transparency_data

Determine if an image has transparency data, whether in the form of an alpha channel, a palette with an alpha channel, or a “transparency” key in the info dictionary.

Note the image might still appear solid, if all of the values shown within are opaque.

RETURNS:

A boolean.

Classes

class PIL.Image.**Exif** [source]

Bases: `MutableMapping`

This class provides read and write access to EXIF image data:

```
from PIL import Image
im = Image.open("exif.png")
exif = im.getexif()  # Returns an instance of this class
```

Information can be read and written, iterated over or deleted:

```
print(exif[274])  # 1
exif[274] = 2
for k, v in exif.items():
  print("Tag", k, "Value", v)  # Tag 274 Value 2
del exif[274]
```

To access information beyond IFD0, `get_ifd()` returns a dictionary:

```
from PIL import ExifTags
im = Image.open("exif_gps.jpg")
exif = im.getexif()
gps_ifd = exif.get_ifd(ExifTags.IFD.GPSInfo)
print(gps_ifd)
```

Other IFDs include `ExifTags.IFD.Exif`, `ExifTags.IFD.MakerNote`, `ExifTags.IFD.Interop` and `ExifTags.IFD.IFD1`.

`ExifTags` also has enum classes to provide names for data:

```
print(exif[ExifTags.Base.Software])  # PIL
print(gps_ifd[ExifTags.GPS.GPSDateStamp])  # 1999:99:99 99:99:99
```

bigtiff = False

endian: str | None = None

get_ifd(tag: int) → dict[int, Any] [source]

hide_offsets() → None [source]

load(data: bytes) → None [source]

load_from_fp(fp: IO[bytes], offset: int | None = None) → None [source]

tobytes(offset: int = 8) → bytes [source]

class PIL.Image.**ImagePointHandler** [source]

Used as a mixin by point transforms (for use with `point()`)

class PIL.Image.**ImagePointTransform**(scale: float, offset: float) [source]

Used with `point()` for single band images with more than 8 bits, this represents an affine transformation, where the value is multiplied by `scale` and `offset` is added.

class PIL.Image.**ImageTransformHandler** [source]

Used as a mixin by geometry transforms (for use with `transform()`)

Protocols

class PIL.Image.**SupportsArrayInterface**(*args, **kwargs) [source]

Bases: `Protocol`

An object that has an `__array_interface__` dictionary.

class PIL.Image.**SupportsGetData**(*args, **kwargs) [source]

Bases: `Protocol`

Constants

PIL.Image.**NONE**

PIL.Image.**MAX_IMAGE_PIXELS**

Set to 89,478,485, approximately 0.25GB for a 24-bit (3 bpp) image. See `open()` for more information about how this is used.

PIL.Image.**WARN_POSSIBLE_FORMATS**

Set to false. If true, when an image cannot be identified, warnings will be raised from formats that attempted to read the data.

Transpose methods

Used to specify the `Image.transpose()` method to use.

```
class PIL.Image.Transpose(value, names=<not given>, *values, module=None,
    qualname=None, type=None, start=1, boundary=None)                    [source]
```

```
    FLIP_LEFT_RIGHT = 0

    FLIP_TOP_BOTTOM = 1

    ROTATE_180 = 3

    ROTATE_270 = 4

    ROTATE_90 = 2

    TRANSPOSE = 5

    TRANSVERSE = 6
```

Transform methods

Used to specify the `Image.transform()` method to use.

class PIL.Image.**Transform** [source]

AFFINE

Affine transform

EXTENT

Cut out a rectangular subregion

PERSPECTIVE

Perspective transform

QUAD

Map a quadrilateral to a rectangle

MESH

Map a number of source quadrilaterals in one operation

Resampling filters

See Filters for details.

```
class PIL.Image.Resampling(value, names=<not given>, *values, module=None,
    qualname=None, type=None, start=1, boundary=None)                    [source]
```

```
    BICUBIC = 3

    BILINEAR = 2

    BOX = 4

    HAMMING = 5

    LANCZOS = 1

    NEAREST = 0
```

Dither modes

Used to specify the dithering method to use for the `convert()` and `quantize()` methods.

```
class PIL.Image.Dither                                                    [source]
```

NONE

No dither

ORDERED

Not implemented

RASTERIZE

Not implemented

FLOYDSTEINBERG

Floyd-Steinberg dither

Palettes

Used to specify the palette to use for the `convert()` method.

```
class PIL.Image.Palette(value, names=<not given>, *values, module=None, qualname=None,
    type=None, start=1, boundary=None)                                    [source]
```

ADAPTIVE = 1

WEB = 0

Quantization methods

Used to specify the quantization method to use for the `quantize()` method.

class PIL.Image.**Quantize** [source]

MEDIANCUT

Median cut. Default method, except for RGBA images. This method does not support RGBA images.

MAXCOVERAGE

Maximum coverage. This method does not support RGBA images.

FASTOCTREE

Fast octree. Default method for RGBA images.

LIBIMAGEQUANT

libimagequant

Check support using `PIL.features.check_feature()` with `feature="libimagequant"`.