Pillow v2.4.0 (PIL fork)

`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

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).

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

```
from PIL import Image
im = Image.open("bride.jpg")
im.rotate(45).show()
```

The following script creates nice 128x128 thumbnails of all JPEG images in the current directory.

Create thumbnails

```
from PIL import Image
import glob, os

size = 128, 128

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

Functions

`PIL.Image.`**open**(*fp*, *mode='r'*) [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()`.

Parameters:	• **file** – A filename (string) or a file object. The file object must implement `read()`, `seek()`, and `tell()` methods, and be opened in binary mode. • **mode** – The mode. If given, this argument must be “r”.
Returns:	An `Image` object.
Raises IOError:	If the file cannot be found, or the image cannot be opened and identified.

Image processing

PIL.Image.**alpha_composite**(*im1*, *im2*) [source]

Alpha composite im2 over im1.

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

Returns: An `Image` object.

PIL.Image.**blend**(*im1*, *im2*, *alpha*) [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*, *image2*, *mask*) [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 can have mode “1”, “L”, or “RGBA”, and must have the same size as the other two images.

PIL.Image.**eval**(*image*, **args*) [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*, *bands*) [source]

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

Parameters:
- **mode** – The mode to use for the output image.
- **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*, *size*, *color=0*) [source]

Creates a new image with the given mode and size.

Parameters:
- **mode** – The mode to use for the new image.
- **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 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*, *mode=None*) [source]

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

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

Parameters:
- **obj** – Object with array interface
- **mode** – Mode to use (will be determined from type if None)

Returns: An image memory.

New in version 1.1.6.

PIL.Image.**frombytes**(*mode*, *size*, *data*, *decoder_name='raw'*, **args*) [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 Decoder**.

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.
- **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.**fromstring**(**args*, ***kw*) [source]

Deprecated alias to frombytes.

Deprecated since version 2.0.

PIL.Image.**frombuffer**(*mode*, *size*, *data*, *decoder_name='raw'*, **args*) [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.

In the current version, the default parameters used for the “raw” decoder differs from that used for `fromstring()`. 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.
- **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.

New in version 1.1.4.

Registering plugins

Note: These functions are for use by plugin authors. Application authors can ignore them.

`PIL.Image.`**`register_open`**(*id*, *factory*, *accept=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*, *mimetype*) [source]

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

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

`PIL.Image.`**`register_save`**(*id*, *driver*) [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_extension**(*id*, *extension*) [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.

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.**convert**(*mode=None*, *matrix=None*, *dither=None*, *palette=0*, *colors=256*) [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.

The current version supports all possible conversions between "L", "RGB" and "CMYK." The **matrix** argument only supports "L" and "RGB".

When translating a color image to black and white (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 greyscale ("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 non-zero values are set to 255 (white). To use other thresholds, use the **point()** method.

Parameters:
- **mode** – The requested mode.
- **matrix** – An optional conversion matrix. If given, this should be 4- or 16-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 NONE or FLOYDSTEINBERG (default).
- **palette** – Palette to use when converting from mode "RGB" to "P". Available palettes are WEB or ADAPTIVE.
- **colors** – Number of colors to use for the 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() [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=None*) [source]

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 **load()** method on the cropped copy.

Parameters: **box** – The crop rectangle, as a (left, upper, right, lower)-tuple.
Return type: Image
Returns: An Image object.

Image.draft(*mode*, *size*) [source]

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

Parameters:
- **mode** – The requested mode.
- **size** – The requested size.

Image.filter(*filter*) [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.

Image.getbands() [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

Image.**getbbox**() [source]

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

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

Image.**getcolors**(*maxcolors=256*) [source]

Returns a list of colors used in this image.

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=None*) [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.**getextrema**() [source]

Gets the 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.**getpixel**(*xy*) [source]

Returns the pixel value at a given position.

Parameters: **xy** – The coordinate, given as (x, y).

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

Image.**histogram**(*mask=None*, *extrema=None*) [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. 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”).

Parameters: **mask** – An optional mask.
Returns: A list containing pixel counts.

Image.**offset**(*xoffset*, *yoffset=None*) [source]

Deprecated since version 2.0.

> **Note:** New code should use `PIL.ImageChops.offset()`.

Returns a copy of the image where the data has been offset by the given distances. Data wraps around the edges. If **yoffset** is omitted, it is assumed to be equal to **xoffset**.

Parameters:
- **xoffset** – The horizontal distance.
- **yoffset** – The vertical distance. If omitted, both distances are set to the same value.

Returns: An `Image` object.

Image.**paste**(*im*, *box=None*, *mask=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)). 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 method 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" or "RGBA" images (in the latter case, 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 can be used for transparency effects.

Note that if you paste an "RGBA" image, the alpha band is ignored. You can work around this by using the same image as both source image and mask.

Parameters:
- **im** – Source image or pixel value (integer 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*, *mode=None*) [source]

Maps this image through a lookup table or function.

Parameters:
- **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.
- **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 `Image` object.

`Image.`**`putalpha`**(*alpha*) [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 or other color value.

`Image.`**`putdata`**(*data*, *scale=1.0*, *offset=0.0*) [source]

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**.

Parameters:
- **data** – A 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*, *rawmode='RGB'*) [source]

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.

Parameters: **data** – A palette sequence (either a list or a string).

`Image.`**`putpixel`**(*xy*, *value*) [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.

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).
- **value** – The pixel value.

`Image.`**`quantize`**(*colors=256*, *method=None*, *kmeans=0*, *palette=None*) [source]

`Image.`**`resize`**(*size*, *resample=0*) [source]

Returns a resized copy of this image.

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

Returns: An `Image` object.

`Image.`**`rotate`**(*angle*, *resample=0*, *expand=0*) [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 `PIL.Image.NEAREST` (use nearest neighbour), `PIL.Image.BILINEAR` (linear interpolation in a 2x2 environment), or `PIL.Image.BICUBIC` (cubic spline interpolation in a 4x4 environment). If omitted, or if the image has mode “1” or “P”, it is set `PIL.Image.NEAREST`.
- **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 `Image` object.

`Image.`**`save`**(*fp*, *format=None*, ***params*) [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 later in this handbook.

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:
- **file** – File name 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.
- **options** – Extra parameters to the image writer.

Returns: None

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

`Image.`**`seek`**(*frame*) [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.

Note that in the current version of the library, most sequence formats only allows you to seek to the next frame.

See `tell()`.

Parameters: **frame** – Frame number, starting at 0.
Raises EOFError:
If the call attempts to seek beyond the end of the sequence.

`Image.`**`show`**(*title=None*, *command=None*) [source]

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).

Parameters:
- **title** – Optional title to use for the image window, where possible.
- **command** – command used to show the image

`Image.`**`split`**() [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).

Returns: A tuple containing bands.

`Image.`**`tell`**() [source]

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

Returns: Frame number, starting with 0.

`Image.`**`thumbnail`**(*size*, *resample=1*) [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 the bilinear and bicubic filters in the current version of PIL are not well-suited for thumbnail generation. You should use `PIL.Image.ANTIALIAS` unless speed is much more important than quality.

Also 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** – Requested size.

- **resample** – Optional resampling filter. This can be one of `PIL.Image.NEAREST`, `PIL.Image.BILINEAR`, `PIL.Image.BICUBIC`, or `PIL.Image.ANTIALIAS` (best quality). If omitted, it defaults to `PIL.Image.ANTIALIAS`. (was `PIL.Image.NEAREST` prior to version 2.5.0)

Returns: None

`Image.`**`tobitmap`**(*name='image'*) [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.`**`tostring`**(**args*, ***kw*) [source]

`Image.`**`transform`**(*size*, *method*, *data=None*, *resample=0*, *fill=1*) [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.
- **method** – The transformation method. This is one of `PIL.Image.EXTENT` (cut out a rectangular subregion), `PIL.Image.AFFINE` (affine transform), `PIL.Image.PERSPECTIVE` (perspective transform), `PIL.Image.QUAD` (map a quadrilateral to a rectangle), or `PIL.Image.MESH` (map a number of source quadrilaterals in one operation).
- **data** – Extra data to the transformation method.
- **resample** – Optional resampling filter. It can be one of `PIL.Image.NEAREST` (use nearest neighbour), `PIL.Image.BILINEAR` (linear interpolation in a 2x2 environment), or `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 `PIL.Image.NEAREST`.

Returns: An `Image` object.

`Image.`**`transpose`**(*method*) [source]

Transpose image (flip or rotate in 90 degree steps)

Parameters: **method** – One of `PIL.Image.FLIP_LEFT_RIGHT`, `PIL.Image.FLIP_TOP_BOTTOM`, `PIL.Image.ROTATE_90`, `PIL.Image.ROTATE_180`, or `PIL.Image.ROTATE_270`.
Returns: Returns a flipped or rotated copy of this image.

`Image.`**`verify`**() [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.**fromstring**(**args*, ***kw*) [source]

Deprecated alias to frombytes.

Deprecated since version 2.0.

Image.**load**() [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. This method will close the file associated with the image.

Returns: An image access object.

Image.**close**() [source]

Closes the file pointer, if possible.

This operation will destroy the image core and release it's 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 **load()** method.

Attributes

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

PIL.Image.**format**

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.

Type : **string** or None

PIL.Image.**mode**

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

Type : **string**

PIL.Image.**size**

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

Type : (width, height)

PIL.Image.**palette**

Colour palette table, if any. If mode is "P", this should be an instance of the **ImagePalette** class. Otherwise, it should be set to None.

Type : **ImagePalette** or None

PIL.Image.**info**

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.

Type : `dict`