`ImageMath` Module

The `ImageMath` module can be used to evaluate “image expressions”. The module provides a single eval function, which takes an expression string and one or more images.

Example: Using the `ImageMath` module

```
from PIL import Image, ImageMath

im1 = Image.open("image1.jpg")
im2 = Image.open("image2.jpg")

out = ImageMath.eval("convert(min(a, b), 'L')", a=im1, b=im2)
out.save("result.png")
```

`PIL.ImageMath.eval`(*expression, environment*)

Evaluate expression in the given environment.

In the current version, `ImageMath` only supports single-layer images. To process multi-band images, use the `split()` method or `merge()` function.

Parameters:
- **expression** – A string which uses the standard Python expression syntax. In addition to the standard operators, you can also use the functions described below.
- **environment** – A dictionary that maps image names to Image instances. You can use one or more keyword arguments instead of a dictionary, as shown in the above example. Note that the names must be valid Python identifiers.

Returns: An image, an integer value, a floating point value, or a pixel tuple, depending on the expression.

Expression syntax

Expressions are standard Python expressions, but they're evaluated in a non-standard environment. You can use PIL methods as usual, plus the following set of operators and functions:

Standard Operators

You can use standard arithmetical operators for addition (+), subtraction (-), multiplication (*), and division (/).

The module also supports unary minus (-), modulo (%), and power (**) operators.

Note that all operations are done with 32-bit integers or 32-bit floating point values, as necessary. For example, if you add two 8-bit images, the result will be a 32-bit integer image. If you add a floating point constant to an 8-bit image, the result will be a 32-bit floating point image.

You can force conversion using the `convert()`, `float()`, and `int()` functions described below.

Bitwise Operators

The module also provides operations that operate on individual bits. This includes and (&), or (|), and exclusive or (^). You can also invert (~) all pixel bits.

Note that the operands are converted to 32-bit signed integers before the bitwise operation is applied. This means that you'll get negative values if you invert an ordinary greyscale image. You can use the and (&) operator to mask off unwanted bits.

Bitwise operators don't work on floating point images.

Logical Operators

Logical operators like `and`, `or`, and `not` work on entire images, rather than individual pixels.

An empty image (all pixels zero) is treated as false. All other images are treated as true.

Note that `and` and `or` return the last evaluated operand, while not always returns a boolean value.

Built-in Functions

These functions are applied to each individual pixel.

`abs`(*image*)

Absolute value.

`convert`(*image, mode*)

Convert image to the given mode. The mode must be given as a string constant.

`float`(*image*)

Convert image to 32-bit floating point. This is equivalent to convert(image, “F”).

`int`(*image*)

Convert image to 32-bit integer. This is equivalent to convert(image, “I”).

Note that 1-bit and 8-bit images are automatically converted to 32-bit integers if necessary to get a correct result.

`max`(*image1, image2*)

Maximum value.

`min`(*image1, image2*)

Minimum value.