Gimp Python Documentation

James Henstridge

<james@daa.com.au>

v0.4, 5 July 1999

This document outlines the interfaces to Gimp-Python, which is a set of Python modules that act as a wrapper to `libgimp` allowing the writing of plug-ins for Gimp. In this way, Gimp-Python is similar to Script-Fu, except that you can use the full set of Python extension modules from the plug-in.

Table of Contents

1. Introduction

1.1. What is it?

Gimp-Python is a scripting extension for Gimp, similar to Script-Fu. The main difference is in what is called first. In Script-Fu, the script-fu plugin executes the script, while in Gimp-Python the script is in control.

In fact, you will find that the Gimp-Python scripts start with the line `#!/usr/bin/python`. The gimp extension is loaded with the familiar `import` command.

Another point of difference between Gimp-Python and Script-Fu is that Gimp-Python stores images, layers, channels and other types as objects rather than just storing their ID. This allows better type checking that is missing from Script-Fu, and allows those types to act as objects, complete with attributes and methods.

Also, Gimp-Python is not limited to just calling procedures from the PDB. It also implements the rest of `libgimp`, including tiles and pixel regions, and access to other lower level functions.

1.2. Installation

Gimp-python consists of a Python module written in C and some native python support modules. You can build pygimp with the commands:

```
./configure
make
make install
```

This will build and install gimpmodule and its supporting modules, and install the sample plugins in gimp's plugin directory.

2. The Structure Of A Plugin

The majority of code in this package resides in `gimpmodule.c`, but this provides a poor interface for implementing some portions of a plugin. For this reason, there is a python module called `plugin.py` that sets out a structure for plugins and implements some things that were either too dificult or impossible to do in C.

The main purpose of `plugin.py` was to implement an object oriented structure for plug-ins. As well as this, it handles tracebacks, which are otherwise ignored by `libgimp`, and gives a method to call other Gimp-Python plug-ins without going through the procedural database.

2.1. An Example Plugin

As in a lot of manuals, the first thing you examine is an example, so here is an example. I have included it before explaining what it does to allow more advanced programmers to see the structure up front. It is a translation of the clothify Script-Fu extension:

Example 1. A sample python plugin

```
#!/usr/bin/python
import math
from gimpfu import *

have_gimp11 = gimp.major_version > 1 or ₩
              gimp.major_version == 1 and gimp.minor_version >= 1

def python_clothify(timg, tdrawable, bx=9, by=9,
                    azimuth=135, elevation=45, depth=3):
        bx = 9 ; by = 9 ; azimuth = 135 ; elevation = 45 ; depth = 3
        width = tdrawable.width
        height = tdrawable.height
        img = gimp.image(width, height, RGB)
        layer_one = gimp.layer(img, "X Dots", width, height, RGB_IMAGE,
                               100, NORMAL_MODE)
        img.disable_undo()
        if have_gimp11:
                pdb.gimp_edit_fill(layer_one)
        else:
                pdb.gimp_edit_fill(img, layer_one)
        img.add_layer(layer_one, 0)
        pdb.plug_in_noisify(img, layer_one, 0, 0.7, 0.7, 0.7, 0.7)
        layer_two = layer_one.copy()
        layer_two.mode = MULTIPLY_MODE
        layer_two.name = "Y Dots"
        img.add_layer(layer_two, 0)
        pdb.plug_in_gauss_rle(img, layer_one, bx, 1, 0)
        pdb.plug_in_gauss_rle(img, layer_two, by, 0, 1)
        img.flatten()
        bump_layer = img.active_layer
        pdb.plug_in_c_astretch(img, bump_layer)
        pdb.plug_in_noisify(img, bump_layer, 0, 0.2, 0.2, 0.2, 0.2)
        pdb.plug_in_bump_map(img, tdrawable, bump_layer, azimuth,
                             elevation, depth, 0, 0, 0, 0, TRUE, FALSE, 0)
        gimp.delete(img)

register(
        "python_fu_clothify",
        "Make the specified layer look like it is printed on cloth",
        "Make the specified layer look like it is printed on cloth",
        "James Henstridge",
        "James Henstridge",
        "1997-1999",
        "<Image>/Python-Fu/Alchemy/Clothify",
        "RGB*, GRAY*",
        [
                (PF_INT, "x_blur", "X Blur", 9),
                (PF_INT, "y_blur", "Y Blur", 9),
                (PF_INT, "azimuth", "Azimuth", 135),
                (PF_INT, "elevation", "elevation", 45),
                (PF_INT, "depth", "Depth", 3)
        ],
        [],
        python_clothify)

main()
```

2.2. Import Modules

In this plugin, a number of modules are imported. The important ones are:

- `gimpfu`: this module provides a simple interface for writing plugins, similar to what script-fu provides. It provides the GUI for entering in parameters in interactive mode and performs some sanity checks when registering the plugin.

  By using "from gimpfu import *", this module also provides an easy way to get all the commonly used symbols into the plugin's namespace.

- `gimp`: the main part of the gimp extension. This is imported with gimpfu.

- `gimpenums`: a number of useful constants. This is also automatically imported with gimpfu.

The pdb variable is a variable for accessing the procedural database. It is imported into the plugin's namespace with gimpfu for convenience.

2.3. Plugin Framework

With pygimp-0.4, the gimpfu module was introduced. It simplifies writing plugins a lot. It handles the run mode (interactive, non interactive or run with last values), providing a GUI for interactive mode and saving the last used settings.

Using the gimpfu plugin, all you need to do is write the function that should be run, make a call to `register`, and finally a call to `main` to get the plugin started.

If the plugin is to be run on an image, the first parameter to the plugin function should be the image, and the second should be the current drawable (do not worry about the run_mode parameter). Plugins that do not act on an existing image (and hence go in the toolbox's menus) do not need these parameters. Any other parameters are specific to the plugin.

After defining the plugin function, you need to call `register` to register the plugin with gimp (When the plugin is run to query it, this information is passed to gimp. When it is run interactively, this information is used to construct the GUI). The parameters to `register` are:

name
blurb
help
author
copyright
date
menupath
imagetypes
params
results
function

Most of these parameters are quite self explanatory. The menupath option should start with <Image%gt;/ for image plugins and <Toolbox>/ for toolbox plugins. The remainder of the menupath is a slash separated path to its menu item.

The params parameter holds a list parameters for the function. It is a list of tuples. Note that you do not have to specify the run_type, image or drawable parameters, as gimpfu will add these automatically for you. The tuple format is (type, name, description, default [, extra]). The allowed type codes are:

PF_INT8
PF_INT16
PF_INT32
PF_INT
PF_FLOAT
PF_STRING
PF_VALUE
PF_INT8ARRAY
PF_INT16ARRAY
PF_INT32ARRAY
PF_INTARRAY
PF_FLOATARRAY
PF_STRINGARRAY

PF_COLOR
PF_COLOUR
PF_REGION
PF_IMAGE
PF_LAYER
PF_CHANNEL
PF_DRAWABLE
PF_TOGGLE
PF_BOOL
PF_SLIDER
PF_SPINNER
PF_ADJUSTMENT
PF_FONT
PF_FILE
PF_BRUSH
PF_PATTERN
PF_GRADIENT

These values map onto the standard PARAM_* constants. The reason to use the extra constants is that they give gimpfu more information, so it can produce a better interface (for instance, the PF_FONT type is equivalent to PARAM_STRING, but in the GUI you get a small button that will bring up a font selection dialog).

The PF_SLIDER, PF_SPINNER and PF_ADJUSTMENT types require the extra parameter. It is of the form (min, max, step), and gives the limits for the spin button or slider.

The results parameter is a list of 3-tuples of the form (type, name, description). It defines the return values for the function. If there is only a single return value, the plugin function should return just that value. If there is more than one, the plugin function should return a tuple of results.

The final parameter to `register` is the plugin function itself.

After registering one or more plugin functions, you must call the `main` function. This will cause the plugin to start running. A GUI will be displayed when needed, and your plugin function will be called at the appropriate times.

---

3. The Procedural Database

The procedural database is a registry of things gimp and its plugins can do. When you install a procedure for your plugin, you are extending the procedural database.

The procedural database is self documenting, in that when you install a procedure in it, you also add documentation for it, its parameters and return values.

3.1. The Gimp-Python Model

In Gimp-Python, the procedural database is represented by the object *gimp.pdb*. In most of my plugins, I make an assignment from *gimp.pdb* to *pdb* for convenience.

You can query the procedural database with *pdb*'s method `query`. Its specification is:

```
pdb.query(name, [blurb, [help, [author, [copyright, [date, [type]]]]]])
```

Each parameter is a regular expression that is checked against the corresponding field in the procedural database. The method returns a list of the names of matching procedures. If `query` is called without any arguments, it will return every procedure in the database.

3.2. Procedural Database Procedures

Procedures can be accessed as procedures, or by treating *pdb* as a mapping objest. As an example, the probedure `gimp_edit_fill` can be accessed as either `pdb.gimp_edit_fill` or `pdb['gimp_edit_fill']`. The second form is mainly for procedures whose names are not valid Python names (eg in script-fu-..., the dashes are interpreted as minuses).

These procedure objects have a number of attribute:

proc_name

    The name of the procedure.

proc_blurb

    A short peice of information about the procedure.

proc_help

    More detailed information about the procedure.

proc_author

    The author of the procedure.

proc_copyright

    The copyright holder for the procedure (usually the same as the author).

proc_date

    The date when the procedure was written.

proc_type

    The type of procedure. This will be one of PROC_PLUG_IN, PROC_EXTENSION or PROC_TEMPORARY.

nparams

The number of parameters the procedure takes.

nreturn_vals

The number of return values the procedure gives.

params

A description of parameters of the procedure. It takes the form of a tuple of 3-tuples, where each 3-tuple describes a parameter. The items in the 3-tuple are a parameter type (one of the PARAM_* constants), a name for the parameter, and a description of the parameter.

return_vals

A description of the return values. It takes the same form as the `params` attribute.

A procedure object may also be called. At this point, Gimp-Python doesn't support keyword arguments for PDB procedures. Arguments are passed to the procedure in the normal method. The return depends on the number of return values:

- If there are zero return values, `None` is returned.
- If there is only a single return value, it is returned.
- If there are more return values, then they are returned as a tuple.

3.3. More Information

For more information on invoking PDB procedures, please see the example plugins. For information on individual procedures, please see the PDB Browser plugin (in the Xtns menu). It alows you to peruse to the database interactively.

---

4. Gimp Module Procedures

The `gimp` module contains a number of procedures and functions, as well as the definitions of many gimp types such as images, and the procedural database. This section explains the base level procedures.

4.1. Constructors and Object Deletion

There are a number of functions in the `gimp` module that are used to create the objects used to make up an image in Gimp. Here is a set of descriptions of these constructors:

`gimp.image(`*width*, *height*, *type*`)`

This procedure creates an image with the given dimensions and type (type is one of `RGB`, `GRAY` or `INDEXED`).

`gimp.layer(`*`img`*`, `*`name`*`, `*`width`*`, `*`height`*`, `*`type`*`, `*`opacity`*`, `*`mode`*`)`

Create a new layer called *`name`*, with the given dimensions and *`type`* (one of the `*_IMAGE` constants), `opacity` (float between 0 and 100) and a `mode` (one of the `*_MODE` constants). The layer can then be added to the image with the `img.add_layer` method.

`gimp.channel(`*`img`*`, `*`name`*`, `*`width`*`, `*`height`*`, `*`opacity`*`, `*`colour`*`)`

Create a new channel object with the given dimensions, *`opacity`* and *`colour`* (one of the `*_CHANNEL` constants). This channel can then be added to an image.

`gimp.display(`*`img`*`)`

Create a new display window for the given image. The window will not be displayed until a call to `gimp.displays_flush` is made.

`gimp.parasite(name, flags, data)`

Create a new parasite. The parasite can then be attached to gimp, an image or a drawable. This is only available in gimp >= 1.1

When any of these objects get removed from memory (such as when their name goes out of range), the gimp thing it represents does not get deleted with it (otherwise when your plugin finished running, it would delete all its work). In order to delete the thing the Python object represents, you should use the `gimp.delete` procedure. It deletes the gimp thing associated with the Python object given as a parameter. If the object is not an image, layer, channel, drawable or display `gimp.delete` does nothing.

4.2. Configuration Information

There are a number of functions that can be used to gather information about the environment the plugin is running in:

`gimp.color_cube()` or `gimp.colour_cube()`

Returns the current colour cube.

`gimp.gamma()`

Returns the current gamma correction.

`gimp.install_cmap()`

Returns non-zero if a colour map has been installed.

`gimp.use_xshm()`

Returns non-zero if Gimp is using X shared memory.

`gimp.gtkrc()`

Returns the file name of the GTK configuration file.

4.3. Palette Operations

These functions alter the currently selected foreground and background.

`gimp.get_background`()

Returns a 3-tuple containing the current background colour in RGB form.

`gimp.get_foreground`()

Returns a 3-tuple containing the current foreground colour in RGB form.

`gimp.set_background`(*r*, *g*, *b*)

Sets the current background colour. The three arguments can be replaced by a single 3-tuple like that returned by `gimp.get_background`.

`gimp.set_foreground`(*r*, *g*, *b*)

Sets the current foreground colour. Like `gimp.set_background`, the arguments may be replaced by a 3-tuple.

4.4. Gradient Operations

These functions perform operations on gradients:

`gimp.gradients_get_active`()

Returns the name of the active gradient.

`gimp.gradients_set_active`(*name*)

Sets the active gradient.

`gimp.gradients_get_list`()

Returns a list of the names of the available gradients.

`gimp.gradients_sample_uniform`(*num*)

Returns a list of *num* samples, where samples consist of 4-tuples of floats representing the red, green, blue and alpha values for the sample.

`gimp.gradients_sample_custom`(*pos*)

Similar to `gimp.gradients_sample_uniform`, except the samples are taken at the positions given in the list of floats *pos* instead of uniformly through the gradient.

4.5. PDB Registration Functions

These functions either install procedures into the PDB or alert gimp to their special use (eg as file handlers).

For simple plugins, you will usually only need to use `register` from gimpfu.

`gimp.install_procedure`(*name*, *blurb*, *help*, *author*, *copyright*, *date*, *menu_path*, *image_types*, *type*, *params*, *ret_vals*)

This procedure is used to install a procedure into the PDB. The first eight parameters are strings, *type* is a one of the `PROC_*` constants, and the last two parameters are sequences describing the parameters and return values. Their format is the same as the param and ret_vals methods or PDB procedures.

`gimp.install_temp_proc`(*name*, *blurb*, *help*, *author*, *copyright*, *date*, *menu_path*, *image_types*, *type*, *params*, *ret_vals*)

This procedure is used to install a procedure into the PDB temporarily. That is, it must be added again every time gimp is run. This procedure will be called the same way as all other procedures for a plugin.

`gimp.uninstall_temp_proc`(*name*)

This removes a temporary procedure from the PDB.

`gimp.register_magic_load_handler`(*name*, *extensions*, *prefixes*, *magics*)

This procedure tells Gimp that the PDB procedure *name* can load files with *extensions* and *prefixes* (eg http:) with magic information *magics*.

`gimp.register_load_handler`(*name*, *extensions*, *prefixes*)

This procedure tells Gimp that the PDB procedure *name* can load files with *extensions* and *prefixes* (eg http:).

`gimp.register_save_handler`(*name*, *extensions*, *prefixes*)

This procedure tells Gimp that the PDB procedure *name* can save files with *extensions* and *prefixes* (eg http:).

4.6. Other Functions

These are the other functions in the `gimp` module.

`gimp.main`(*init_func*, *quit_func*, *query_func*, *run_func*)

This function is the one that controls the execution of a Gimp-Python plugin. It is better to not use this directly but rather subclass the plugin class, defined in the Section 6.3.

gimp.pdb

The procedural database object.

`gimp.progress_init`(*[label]*)

(Re)Initialise the progress meter with *label* (or the plugin name) as a label in the window.

`gimp.progress_update`(*percnt*)

Set the progress meter to *percnt* done.

`gimp.query_images()`

Returns a list of all the image objects.

`gimp.quit()`

Stops execution imediately and exits.

`gimp.displays_flush()`

Update all the display windows.

`gimp.tile_width()`

The maximum width of a tile.

`gimp.tile_height()`

The maximum height of a tile.

`gimp.tile_cache_size(`*kb*`)`

Set the size of the tile cache in kilobytes.

`gimp.tile_cache_ntiles(`*n*`)`

Set the size of the tile cache in tiles.

`gimp.get_data(`*key*`)`

Get the information associated with *key*. The data will be a string. This function should probably be used through the Section 6.4.

`gimp.set_data(`*key*, *data*`)`

Set the information in the string *data* with *key*. The data will persist for the whole gimp session. Rather than directly accessing this function, it is better to go through the Section 6.4.

`gimp.extension_ack()`

Tells gimp that the plugin has finished its work, while keeping the plugin connection open. This is used by an extension plugin to tell gimp it can continue, while leaving the plugin connection open. This is what the script-fu plugin does so that only one scheme interpretter is needed.

`gimp.extension_process(`*timeout*`)`

Makes the plugin check for messages from gimp. generally this is not needed, as messages are checked during most calls in the gimp module.

4.7. Parasites

In gimp >= 1.1, it is possible to attach arbitrary data to an image through the use of parasites. Parasites are simply wrappers for the data, containing its name and some flags. Parasites have the following parameters:

data

The data for the parasite -- a string

flags

The flags for the parasite

is_persistent

True if this parasite is persistent

is_undoable

True if this parasite is undoable

name

The name of the parasite

Parasites also have the methods `copy`, `is_type` and `has_flag`.

There is a set of four functions that are used to manipulate parasites. They exist as functions in the `gimp` module, and methods for image and drawable objects. They are:

`parasite_find(`*`name`*`)`

find a parasite by its name.

`parasite_attach(`*`parasite`*`)`

Attach a parasite to this object.

`attach_new_parasite(`*`name`*`, `*`flags`*`, `*`data`*`)`

Create a new parasite and attach it.

`parasite_detach(`*`name`*`)`

Detach the named parasite

---

5. Gimp Objects

Gimp-Python implements a number of special object types that represent the different types of parameters you can pass to a PDB procedure. Rather than just making these place holders, I have added a number of members and methods to them that allow a lot of configurability without directly calling PDB procedures.

There are also a couple of extra objects that allow low level manipulation of images. These are tile objects (working) and pixel regions (not quite finished).

5.1. Image Object

This is the object that represents an open image. In this section, *image* represents a generic image object.

5.1.1. Image Members

image.active_channel

This is the active channel of the image. You can also assign to this member, or *None* if there is no active channel.

image.active_layer

This is the active layer of the image. You can also assign to this member, or *None* if there is no active layer.

image.base_type

This is the type of the image (eg RGB, INDEXED).

image.channels

This is a list of the channels of the image. Altering this list has no effect, and you can not assign to this member.

image.cmap

This is the colour map for the image.

image.filename

This is the filename for the image. A file load or save handler might assign to this.

image.height

This is the height of the image. You can't assign to this member.

image.floating_selection

The floating selection layer, or *None* if there is no floating selection.

image.layers

This is a list of the layers of the image.

image.selection

The selection mask for the image.

image.*width*

This is the width of the image. You can't assign to this member.

5.1.2. Image Methods

image.`add_channel`(*channel*, *position*)

Adds *channel* to *image* in position *position*.

image.`add_layer`(*layer*, *position*)

Adds *layer* to *image* in position *position*.

image.`add_layer_mask`(*layer*, *mask*)

Adds the mask *mask* to *layer*.

image.`clean_all`()

Unsets the dirty flag on the image.

image.`disable_undo`()

Disables undo for *image*.

image.`enable_undo`()

Enables undo for *image*. You might use these commands round a plugin, so that the plugin's actions can be undone in a single step.

image.`flatten`()

Returns the resulting layer after merging all the visible layers, discarding non visible ones and stripping the alpha channel.

image.`get_component_active`(*component*)

Returns true if *component* (one of the `*_CHANNEL` constants) is active.

image.`get_component_visible`(*component*)

Returns true if *component* is visible.

image.`set_component_active`(*component*, *active*)

Sets the activeness of *component*.

image.`set_component_visible`(*component*, *active*)

Sets the visibility of *component*.

image.`lower_channel`(*channel*)

Lowers *channel*.

image.lower_layer(*layer*)

Lowers *layer*.

image.merge_visible_layers(*type*)

Merges the visible layers of *image* using the given merge type.

image.pick_correlate_layer(*x*, *y*)

Returns the layer that is visible at the point *(x,y)*, or *None* if no layer matches.

image.raise_channel(*channel*)

Raises *channel*.

image.raise_layer(*layer*)

Raises *layer*.

image.remove_channel(*channel*)

Removes *channel* from *image*.

image.remove_layer(*layer*)

Removes *layer* from *image*.

image.remove_layer_mask(*layer*, *mode*)

Removes the mask from *layer*, with the given *mode* (either APPLY or DISCARD).

image.resize(*width*, *height*, *x*, *y*)

Resizes the image to size *(width, height)* and places the old contents at position *(x,y)*.

5.2. Channel Objects

These objects represent a Gimp Image's colour channels. In this section, *channel* will refer to a generic channel object.

5.2.1. Channel Members

channel.colour or *channel.color*

The colour of the channel.

channel.height

The height of the channel.

channel.*width*

The width of the channel.

channel.*image*

The image the channel belongs to, or *None* if it isn't attached yet.

channel.*layer*

The channel's layer (??) or *None* if one doesn't exist.

channel.*layer_mask*

Non zero if the channel is a layer mask.

channel.*name*

The name of the channel.

channel.*opacity*

The opacity of the channel.

channel.*show_masked*

The show_masked value of the channel.

channel.*visible*

Non-zero if the channel is visible.

5.2.2. Channel Methods

channel.copy()

returns a copy of the channel.

5.3. Layer Objects

Layer objects represent the layers of a Gimp image. In this section I will refer to a generic layer called *layer*.

5.3.1. Layer Members

layer.*apply_mask*

The apply mask setting. (non zero if the layer mask is being composited with the layer's alpha channel).

layer.*bpp*

The number of bytes per pixel.

layer.*edit_mask*

The edit mask setting. (non zero if the mask is active, rather than the layer).

layer.*height*

The height of the layer.

layer.*image*

The image the layer is part of, or *None* if the layer isn't attached.

layer.*is_floating_selection*

Non zero if this layer is the image's floating selection.

layer.*mask*

The layer's mask, or *None* if it doesn't have one.

layer.*mode*

The mode of the layer.

layer.*name*

The name of the layer.

layer.*opacity*

The opacity of the layer.

layer.*preserve_transparency*

The layer's preserve transparency setting.

5.3.2. Layer Methods

layer.add_alpha()

Adds an alpha component to the layer.

layer.copy(*[alpha]*)

Creates a copy of the layer, optionally with an alpha layer.

layer.create_mask(*type*)

Creates a layer mask of type *type*.

layer.resize(*w*, *h*, *x*, *y*)

Resizes the layer to *(w, h)*, positioning the original contents at *(x,y)*.

layer.scale(*h*, *w*, *origin*)

Scales the layer to *(w, h)*, using the specified *origin* (local or image).

layer.set_offsets(*x*, *y*)

Sets the offset of the layer, relative to the image's origin

layer.translate(*x*, *y*)

Moves the layer to *(x, y)* relative to its current position.

5.4. Drawable Objects

Both layers and channels are drawables. Hence there are a number of operations that can be performed on both objects. They also have some common attributes and methods. In the description of these attributes, I will refer to a generic drawable called *drawable*.

5.4.1. Drawable Members

drawable.bpp

The number of bytes per pixel.

drawable.is_colour or *drawable.is_color* or *drawable.is_rgb*

Non zero if the drawable is colour.

drawable.is_grey or *drawable.is_gray*

Non zero if the drawable is greyscale.

drawable.has_alpha

Non zero if the drawable has an alpha channel.

drawable.height

The height of the drawable.

drawable.image

The image the drawable belongs to.

drawable.is_indexed

Non zero if the drawable uses an indexed colour scheme.

drawable.mask_bounds

The bounds of the drawable's selection.

drawable.*name*

The name of the drawable.

drawable.*offsets*

The offset of the top left hand corner of the drawable.

drawable.*type*

The type of the drawable.

drawable.*visible*

Non zero if the drawable is visible.

drawable.*width*

The width of the drawable.

5.4.2. Drawable Methods

drawable.`fill`(*fill_type*)

Fills the drawable with given *fill_type* (one of the `*_FILL` constants).

drawable.`flush`()

Flush the changes to the drawable.

drawable.`get_pixel_rgn`(*x*, *y*, *w*, *h*, [*dirty*, [*shadow*]])

Creates a pixel region for the drawable. It will cover the region with origin *(x,y)* and dimensions *w x h*. The *dirty* argument sets whether any changes to the pixel region will be reflected in the drawable (default is TRUE). The *shadow* argument sets whether the pixel region acts on the shadow tiles or not (default is FALSE). If you draw on the shadow tiles, you must call *drawable*.`merge_shadow`() for changes to take effect.

drawable.`get_tile`(*shadow*, *row*, *col*)

Get a tile at *(row, col)*. Either on or off the *shadow* buffer.

drawable.`get_tile2`(*shadow*, *x*, *y*)

Get the tile that contains the pixel *(x, y)*.

drawable.`merge_shadow`()

Merge the shadow buffer back into the drawable.

drawable.`update`(*x*, *y*, *w*, *h*)

Update the given portion of the drawable.

5.5. Tile Objects

Tile objects represent the way Gimp stores information. A tile is basically just a 64x64 pixel region of the drawable. The reason Gimp breaks the image into small pieces like this is so that the whole image doesn't have to be loaded into memory in order to alter one part of it. This becomes important with larger images.

In Gimp-Python, you would use Tiles if you wanted to perform some low level operation on the image, instead of using procedures in the PDB. This type of object gives a Gimp-Python plugin the power of a C plugin, rather than just the power of a Script-Fu script. Tile objects are created with either the *drawable*.`get_tile`() or *drawable*.`get_tile2`() functions. In this section, I will refer to a generic tile object named *tile*.

5.5.1. Tile Members

All tile members are read only.

tile.bpp

The number of bytes per pixel.

tile.dirty

If there have been changes to the tile since it was last flushed.

tile.drawable

The drawable that the tile is from.

tile.eheight

The actual height of the tile.

tile.ewidth

The actual width of the tile.

tile.ref_count

The reference count of the tile. (this is independent of the Python object reference count).

tile.shadow

Non zero if the tile is part of the shadow buffer.

5.5.2. Tile Methods

tile.`flush`()

Flush any changes in the tile. Note that the tile is automatically flushed when the Python object is deleted from memory.

5.5.3. Tile Mapping Behaviour

Tile objects also act as a mapping, or sequence. You can access the pixels in the tile in one of two ways. You can either access them with a single number, which refers to its position in the tile (eg. *tile*`[64]` refers to the first pixel in the second row of a 64x64 pixel tile). The other way is with a tuple, representing the coordinates on the tile (eg. *tile*`[0, 1]` refers to the first pixel on the second row of the tile).

The type of these subscripts is a string of length *tile.bpp*. When you assign to a subscript, the dirty flag is automatically set on the tile, so you don't have to explicitly set the flag, or flush the tile.

5.6. Pixel Regions

Pixel region objects give an interface for low level operations to act on large regions of an image, instead of on small 64x64 pixel tiles. In this section I will refer to a generic pixel region called *pr*. For an example of a pixel region's use, please see the example plugin `whirlpinch.py`.

5.6.1. Pixel Region Members

pr.drawable

The drawable this pixel region is for.

pr.bpp

The number of bytes per pixel for the drawable.

pr.rowstride

The rowstride for the pixel region.

pr.x

The x coordinate of the top left hand corner.

pr.y

The y coordinate of the top left hand corner.

pr.w

The width of the pixel region.

pr.h

The height of the pixel region.

pr.dirty

Non zero if changes to the pixel region will be reflected in the drawable.

pr.*shadow*

Non zero if the pixel region acts on the shadow tiles of the drawable.

5.6.2. Pixel Region Methods

pr.`resize`(*x*, *y*, *w*, *h*)

resize the pixel region so that it operates on the the region with corner *(x, y)* with dimensions *w x h*.

5.6.3. Pixel Region Mapping Behaviour

The pixel region acts as a mapping. The index is a 2-tuple with components that are either integers or slices. The subscripts may be read and assigned to. The type of the subscripts is a string containing the binary data of the requested region. Here is a description of the posible operations:

pr[*x*, *y*]

Get/Set the pixel at *(x,y)*

pr[*x1*:*x2*, *y*]

Get/Set the row starting at *(x1, y)*, width *x2 - x1*.

pr[*x*, *y1*:*y2*]

Get/Set the column starting at *(x, y1)*, height *y2 - y1*.

pr[*x1*:*x2*, *y1*:*y1*]

Get/Set the rectangle starting at *(x1, y1)*, width *x2 - x1* and height *y2 - y1*.

---

6. Support Modules

This section describes the modules that help make using the `gimp` module easier. These range from a set of constants to storing persistent data.

6.1. The gimpenums Module

This module contains all the constants found in the header `libgimp/gimpenums.h`, as well as some extra constants that are available in Script-Fu.

6.2. The gimpfu Module

This module was fully described in an earlier section. It provides an easy interface for writing plugins, where you do not need to worry about run_modes, GUI's and saving previous values. It is the recommended module for writing plugins.

6.3. The gimpplugin Module

This module provides the framework for writing Gimp plugins in Python. It gives more flexibility for writing plugins than the gimpfu module, but does not offer as many features (such as automatic GUI building).

To use this framework you subclass `gimpplugin.plugin` like so:

```
import gimpplugin
class myplugin(gimpplugin.plugin):
        def init(self):
                # initialisation routines
                # called when gimp starts.
        def quit(self):
                # clean up routines
                # called when gimp exits (normally).
        def query(self):
                # called to find what functionality the plugin provides.
                gimp.install_procedure("procname", ...)
        # note that this method name matches the first arg of
        # gimp.install_procedure
        def procname(self, arg1, ...):
                # do what ever this plugin should do
```

6.4. The gimpshelf Module

This module gives a nicer interface to the persistent storage interface for Gimp plugins. Due to the complicated nature of Python objects (there is often a lot of connections between them), it can be dificult to work out what to store in gimp's persistent storage. The python interface only allows storage of strings, so this module wraps pickle and unpickle to allow persistentstorage of any python object.

Here is some examples of using this module:

```
>>> from gimpshelf import shelf
>>> shelf['james'] = ['forty-two', (42, 42L, 42.0)]
>>> shelf.has_key('james')
1
>>> shelf['james']
['forty-two', (42, 42L, 42.0)]
```

Anything you store with `gimpshelf.shelf` will exist until Gimp exits. This makes this interface perfect for when a plugin is executed with the run mode `RUN_WITH_LAST_VALS`.

7. End Note

This package is not yet complete, but it has enough in it to be useful for writing plugins for Gimp. If you write any plugins that might be useful as examples, please mail me at james@daa.com.au.