Python Fu #8: Conditional statements, modifying the Lomo plug-in

September 14, 2015November 30, 2015 | jacksonbates | gimp, gimp scripting, python, python-fu, tutorial
See the other tutorials in this series. (https://jacksonbates.wordpress.com/python-fu-gimp-scripting-tutorial-pages/)

In this tutorial we will be modifying our last plug-in. All of the code is the same until we get to setting the direction that the blend tool runs in. We will offer the user the option to have the blend tool come in from the top, left, bottom and right, as well as from the four corners, and stop in the middle of the image. To do this we will give the options in a drop-down menu, using compass directions to describe where the blend begins. The code for the tutorial can be found here: lomo2.py (https://gist.github.com/JacksonBates/6e7deec52d59b9c5b159).

```
def lomo_opt(image, drawable, direction):
...
register(
    "python-fu-lomo-opt",
    "Lomo effect opt",
    "Creates a lomo effect on a given image and user input",
    "Jackson Bates", "Jackson Bates", "2015",
    "Lomo opt...",
    "RGB", # type of image it works on (*, RGB, RGB*, RGBA etc...)
    [
        (PF_IMAGE, "image", "takes current image", None),
        (PF_DRAWABLE, "drawable", "Input layer", None),
        (PF_OPTION, "direction", "Direction: ", 0,
             (
                 ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]
              )
         )
    ],
    [],
    lomo_opt, menu="/Filters")  # second item is menu location

main()
```

The first thing we will do is change the registry and function name in the appropriate spaces. You'll notice I have added an argument to the function for the 'direction.' I have also added the parameters in the registry section that will provide users with an option to select a direction. The metaphor that made sense to me was that of the compass, so I allow the user to pick one of 8 compass points as the starting point for the blend effect, and keep the centre as the end point as in the previous version of the plug in.

The tricky bit is getting the plug in to alter the values that we pass into the pdb.gimp_edit_blend method we used before. We basically want it to follow the logic: if the user picked N, blend down from the top center, if the user picked NE blend down from the top right corner…and so on.

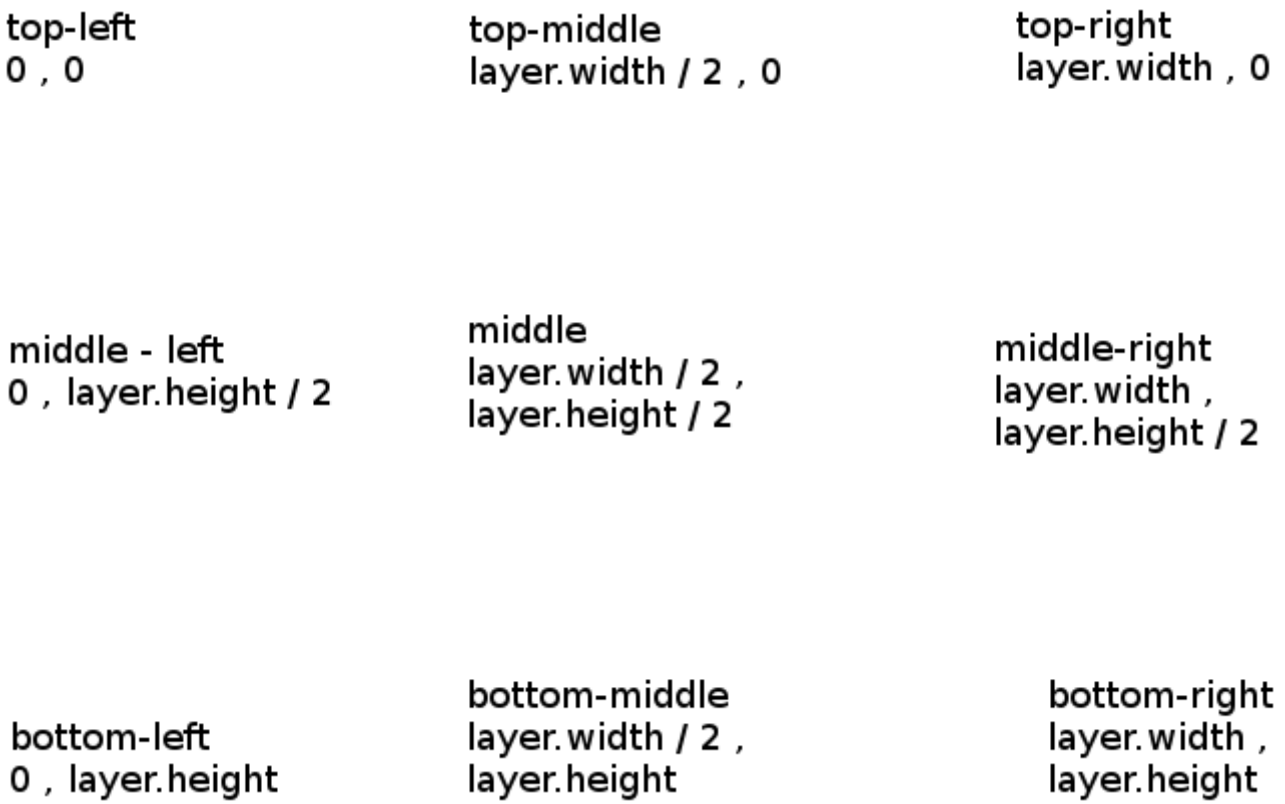
Now there are probably a few solutions to this – and I suspect I picked the slowest, hackiest way to do it – but the final plug in works, so I don't care!

The first thing I did was create a variable for each of the collections of gimp_edit_blend arguments that would be possible. I repeat my self a lot in this code – breaking the ‘don’t repeat yourself’ rule – but I’m not perfect. A pro would find a cleaner way to do this, I am certain.

I know what arguments gimp_edit_blend takes (layer, blend mode, paint_mode etc…) so I create a variable to contain each of those sets of values, labelled for each compass point (N, NE, E etc…).

Since the only part that is changing is the starting point of the blend, I only need to alter the 13th and 14th arguments, which specifiy the x and y axes respectively. The final two arguments specify the end point, but since this is always the centre it will always be layer.width / 2 and layer.height / 2.

It can be tricky to keep the proper pixel positions in your head, so to make it easier for my self I put together a quick reference chart. This should help me keep track of the values I need to put in.



(https://jacksonbates.files.wordpress.com/2015/09/pixel-position-ref.png)

So I go ahead and make the appropriate edits:

```
# arguments for blend, including directions
    # backslash character is used to allow short line lengths
    n = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
        repeat, reverse, supersample,  max_depth, threshold,  dither, \
        layer.width / 2, 0, layer.width / 2, layer.height / 2

    ne = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
         repeat, reverse, supersample,  max_depth, threshold,  dither, \
         layer.width, 0, layer.width / 2, layer.height / 2

    e = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
        repeat, reverse, supersample,  max_depth, threshold,  dither, \
        layer.width, layer.height / 2, layer.width / 2, layer.height / 2

    se = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
         repeat, reverse, supersample,  max_depth, threshold,  dither, \
         layer.width, layer.height, layer.width / 2, layer.height / 2

    s = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
        repeat, reverse, supersample,  max_depth, threshold,  dither, \
        layer.width / 2, layer.height, layer.width / 2, layer.height / 2

    sw = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
         repeat, reverse, supersample,  max_depth, threshold,  dither, \
         0, layer.height, layer.width / 2, layer.height / 2

    w = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
        repeat, reverse, supersample,  max_depth, threshold,  dither, \
        0, layer.height / 2, layer.width / 2, layer.height / 2

    nw = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
         repeat, reverse, supersample,  max_depth, threshold,  dither, \
         0, 0, layer.width / 2, layer.height / 2
```

And there we have the variables containing the arguments we pass into the gimp_edit_blend. It is not pretty.

The next trick is one of the core concepts of programming that you will find useful time and time again: conditional statements.

We'll look at a simple example in the console:

We will create 3 possible conditions and provide different responses based on which condition is met.
We'll create it as a function so we can test all three outcomes easily.
We create the function with using the def command:

```
>>>def greater(a, b):
```

and I say it will take two arguments, a and b. Then we have to remember to indent – I'll use two spaces each time I indent. I want my function to check which value is greater, and print a message accordingly.

```
...  if a > b:
...    print "A is greater"
...  elif b > a:
...    print "B is greater"
...  else:
...    print "same same, but different"
```

So this should be pretty readable, but I'll explain it. the first line checks the condition, "If a is greater than b" and the next line provides the action to take if the condition is true. The remaining lines follow the same principle, but the language is slightly different. Elif means 'else if', so if the first condition is not met, it tries the next one. The final one, else, is for when all other conditions fail. You can have as many elifs as you like in the middle, which is handy, because we have 8 conditions for the plugin.

So we can test the function by calling it and passing in the appropriate values:

```
>>>greater(2,1)
A is greater
>>>greater(1,2)
B is greater
>>>greater(2,2)
same, same, but different
```

We will use that principle for the final part of our plug in.

The condition we are checking is the value of the ‘direction’ variable that is selected by the user. We can find those in the PF_OPTION information in the registry section. The values are stored as strings in a list, but we will be referring to them by their position in the list. So instead of ‘if direction == “N” we would write if direction == 0, because it is the first item in the list, and computers start counting at zero.

So each conditonal statement will look like this:

```
if direction == 0:
  pdb.gimp_edit_blend(*n)
```

The asterix is required because for some reason I could not fathom, simply passing in the variable led to the wrong number of parameters. I did some googling and found that the asterix allows a flexible number of values in the variable. Like I said previously, this was a hacky solution, and probably not best practice, but it worked.

If I fill in the rest of the conditions now, you should see how it all comes together.

```
# the number used denotes the list position of the direction chosen
    if direction == 0:
        pdb.gimp_edit_blend(*n)
    elif direction == 1:
        pdb.gimp_edit_blend(*ne)
    elif direction == 2:
        pdb.gimp_edit_blend(*e)
    elif direction == 3:
        pdb.gimp_edit_blend(*se)
    elif direction == 4:
        pdb.gimp_edit_blend(*s)
    elif direction == 5:
        pdb.gimp_edit_blend(*sw)
    elif direction == 6:
        pdb.gimp_edit_blend(*w)
    else:
        pdb.gimp_edit_blend(*nw)
```

Finally we save it and check that our plug in has worked.

And of course it does, so we are finished. I hope you've found this series helpful. It should give you enough of an idea to be able to create your own original plug ins. It won't always be easy, but you should get there with some detective work and learning to ask the right questions on forums.

Related posts: Python Fu GIMP tutorials (https://jacksonbates.wordpress.com/python-fu-gimp-scripting-tutorial-pages/)

Here's the completed code for the tutorial:

```
#!/usr/bin/env python

# Tutorial available at: https://www.youtube.com/watch?v=oNn9D_8d4zQ
# Feedback welcome: jacksonbates@hotmail.com

from gimpfu import *


def lomo_opt(image, drawable, direction):
    pdb.gimp_image_undo_group_start(image)

    #adjust curves on RG and B channels
    s_curve = (0, 0, 96, 64, 128, 128, 160, 192, 255, 255)
    inverted_s_curve = (0, 0, 64, 96, 128, 128, 192, 160, 255, 255)
    num_points = 10
    pdb.gimp_curves_spline(drawable, HISTOGRAM_RED, num_points, s_curve)
    pdb.gimp_curves_spline(drawable, HISTOGRAM_GREEN, num_points, s_curve)
    pdb.gimp_curves_spline(drawable, HISTOGRAM_BLUE, num_points,
                           inverted_s_curve)

    #add new layer & Set to 'overlay'
    opacity_100 = 100
    layer = pdb.gimp_layer_new(image, image.width, image.height, RGB_IMAGE,
                               "Overlay", opacity_100, OVERLAY_MODE)
    layer_position = 0
    pdb.gimp_image_insert_layer(image, layer, None, layer_position)

    # blend argument variables
    blend_mode = 0
    paint_mode = 0
    gradient_type = 0
    offset = 0
    repeat = 0
```

```
reverse = False
supersample = False
max_depth = 1
threshold = 0
dither = True

# arguments for blend, including directions
# backslash character is used to allow short line lengths
n = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
    repeat, reverse, supersample,  max_depth, threshold,  dither, \
    layer.width / 2, 0, layer.width / 2, layer.height / 2

ne = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
     repeat, reverse, supersample,  max_depth, threshold,  dither, \
     layer.width, 0, layer.width / 2, layer.height / 2

e = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
    repeat, reverse, supersample,  max_depth, threshold,  dither, \
    layer.width, layer.height / 2, layer.width / 2, layer.height / 2

se = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
     repeat, reverse, supersample,  max_depth, threshold,  dither, \
     layer.width, layer.height, layer.width / 2, layer.height / 2

s = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
    repeat, reverse, supersample,  max_depth, threshold,  dither, \
    layer.width / 2, layer.height, layer.width / 2, layer.height / 2

sw = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
     repeat, reverse, supersample,  max_depth, threshold,  dither, \
     0, layer.height, layer.width / 2, layer.height / 2

w = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
    repeat, reverse, supersample,  max_depth, threshold,  dither, \
    0, layer.height / 2, layer.width / 2, layer.height / 2
```

```
    nw = layer, blend_mode, paint_mode, gradient_type, opacity_100, offset, \
         repeat, reverse, supersample,  max_depth, threshold,  dither, \
         0, 0, layer.width / 2, layer.height / 2

    # blend tool, linear gradient (to add user entry,
    # based on diagonal desired for gradient direction?)
    # the number used denotes the list position of the direction chosen
    if direction == 0:
        pdb.gimp_edit_blend(*n)
    elif direction == 1:
        pdb.gimp_edit_blend(*ne)
    elif direction == 2:
        pdb.gimp_edit_blend(*e)
    elif direction == 3:
        pdb.gimp_edit_blend(*se)
    elif direction == 4:
        pdb.gimp_edit_blend(*s)
    elif direction == 5:
        pdb.gimp_edit_blend(*sw)
    elif direction == 6:
        pdb.gimp_edit_blend(*w)
    else:
        pdb.gimp_edit_blend(*nw)

    #merge all layers, and end undo group
    layer = pdb.gimp_image_merge_visible_layers(image, 0)
    pdb.gimp_image_undo_group_end(image)


register(
    "python-fu-lomo-opt",
    "Lomo effect opt",
    "Creates a lomo effect on a given image and user input",
    "Jackson Bates", "Jackson Bates", "2015",
```

```
    "Lomo opt...",
    "RGB", # type of image it works on (*, RGB, RGB*, RGBA etc...)
    [
        (PF_IMAGE, "image", "takes current image", None),
        (PF_DRAWABLE, "drawable", "Input layer", None),
        (PF_OPTION, "direction", "Direction: ", 0,
            (
                ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]
             )
         )
    ],
    [],
    lomo_opt, menu="/Filters")  # second item is menu location

main()
```

One thought on “Python Fu #8: Conditional statements, modifying the Lomo plug-in”

UGAJIN SAYS: Congratulations Jackson, you put a lot of effort into your tutorials. I learned about Lomo images, too.
It is an issue (for me), when I cannot see the effect of adjusting a value in a script’s UI dialogue. I do not know how to get around this.

BTW have you seen this suite of script-fu tutorials: https://www.youtube.com/watch?v=dAuJJ6P8Jxs.

1. Published in 2012, they are nonetheless current, and you might enjoy them, as they are a bit different from what you do, and Lisp is fun!

March 7, 2016 at 10:24 pm • *Reply »*