Automate Editing in Gimp

Table of Contents

1) Motivation

This tutorial will describe and provide examples for two types of automation functions. The first function is a tool to capture and execute “Macro” commands. The second function is a set of Automation Tools to capture and run a “Flow” or “Process”. The code for this tutorial is written using Gimp-Python and should be platform portable – able to run on either Linux or Windows operating systems. *

The goal of these functions is to provide tools that speed up the editing process, make the editing process more repeatable, and reduce the amount of button pushing the user has to do. Taking over the button pushing and book-keeping chores allows the user to focus on the more creative part of the editing process.

These automation tools are examples so please modify them to suit your needs. The goal of the tutorial is to provide and explain a working example. If the coding style seems a bit rough, I apologize, I am not a programmer.

The main body of this tutorial will be discussing the ideas behind automating the editing process. Hopefully we can keep the focus on the concept without getting lost in the details of the included example. The details of how to setup and run the example code and what the automation example is supposed to accomplish will be covered in appendices to the tutorial. We are not going to cover python programing as it is already well documented.

This tutorial will use some of the concepts covered in an earlier tutorial “Automated Jpg to Xcf”. It would probably be helpful to read through the earlier tutorial for introductory example.

* Hopefully it is easy to adapt it to Apple OS as well.

[TOC]

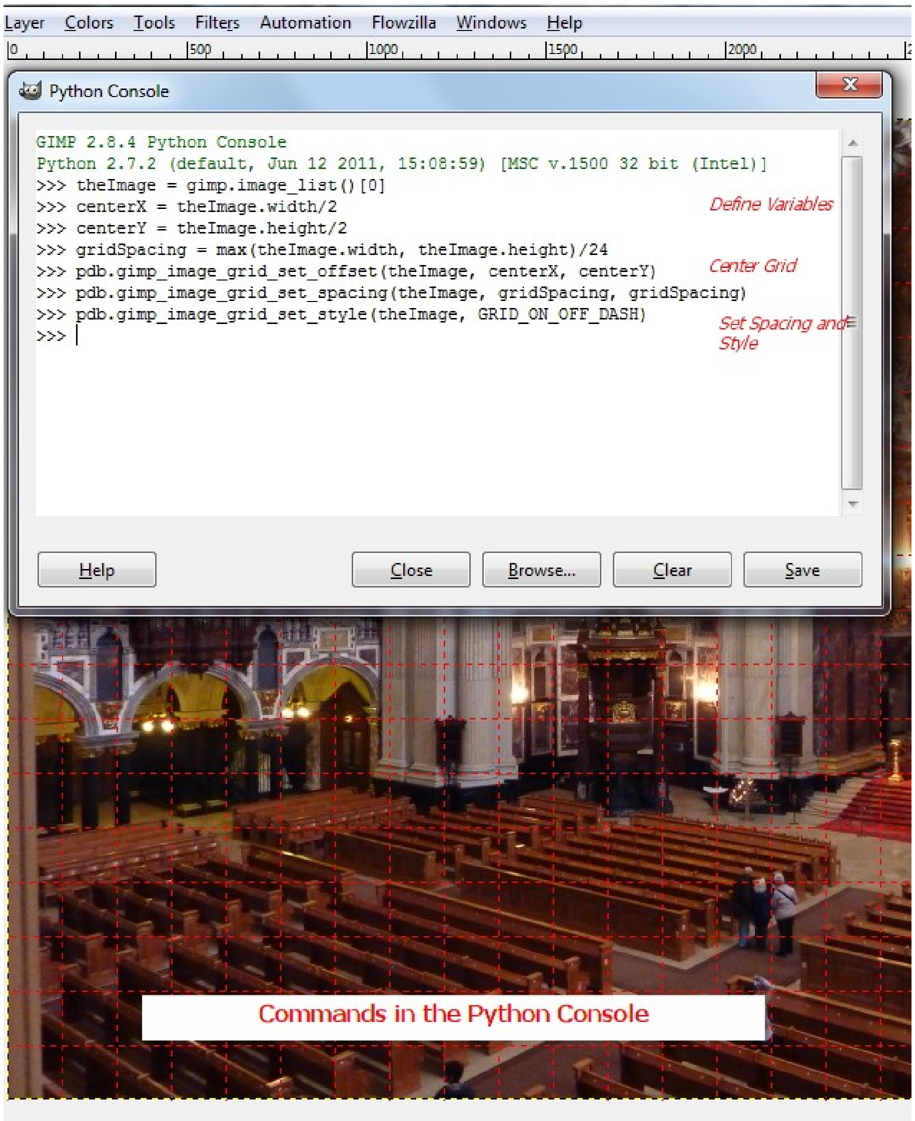
2) An introduction to Macros

Before we dive into a description of the code, let's take a brief walk through the process of capturing and running a “Macro”.

Suppose we wanted to set up the grid spacing so that it is centered on the image, is a square grid, is scaled so there are 24 grid blocks in the 'long' dimension, and is an on-off dash pattern. We could enter the following code fragment in the Gimp “Python Console” (under the “Filters” pull down menu) to set the grid up.

Example - Gimp-Python Code Fragment

```
>>> theImage = gimp.image_list()[0]
>>> centerX = theImage.width/2
>>> centerY = theImage.height/2
>>> gridSpacing = max(theImage.width, theImage.height)/24
>>> pdb.gimp_image_grid_set_offset(theImage, centerX, centerY)
>>> pdb.gimp_image_grid_set_spacing(theImage, gridSpacing, gridSpacing)
>>> pdb.gimp_image_grid_set_style(theImage, GRID_ON_OFF_DASH)
```



Layer Colors Tools Filters Automation Flowzilla Windows Help
Python Console
Define Variables
Center Grid
Set Spacing and Style
Help
Close
Browse...
Clear
Save
Commands in the Python Console

```
GIMP 2.8.4 Python Console
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
>>> theImage = gimp.image_list()[0]
>>> centerX = theImage.width/2
>>> centerY = theImage.height/2
>>> gridSpacing = max(theImage.width, theImage.height)/24
>>> pdb.gimp_image_grid_set_offset(theImage, centerX, centerY)
>>> pdb.gimp_image_grid_set_spacing(theImage, gridSpacing, gridSpacing)
>>> pdb.gimp_image_grid_set_style(theImage, GRID_ON_OFF_DASH)
>>>
```

Image - Commands in the Python Console

If you watch the image as you enter the commands and have the Grid turned “ON” you will see the grid spacing on the active image change as we execute these commands.

The tool we are writing in this tutorial will allow us to copy this code fragment into a text file, add a name and optional comments, and access it through a menu widget so we can rerun this code fragment on other images. The tool will access the “macro” code fragment using the name we assigned through a pull down menu.

The ability to save the macro code fragments allows us to build up a library of editing shortcuts that will not only save time, but do the job better than you could be simply pushing the buttons. In this case we made the grid spacing based on a percentage of the image size rather than a fixed size in either inches, pixels, or cm.

[TOC]

3) Macro Implementation – in Broad Strokes

Let's touch upon the main ideas that we are going to use to implement the macro recording and execution scheme that we can use to capture and run a set of commands.

1. The command set will be read and parsed into a data structure. Each command set block will have a name and consist of commands and comments.
2. The data structure will be read for a list of macro names. This list will be used to populate a GUI widget so we can select a macro by name.
3. From the named Macro, the data structure will return a list of commands.
4. The list of commands will be run in an 'exec' loop:

Example – 'Exec' Loop

```
    for Cmd in commandList:
        exec(Cmd)
```

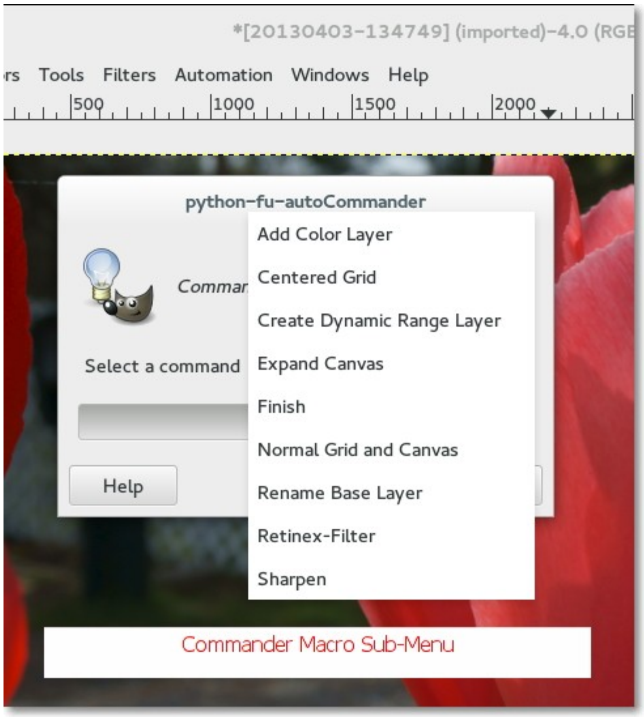


Image - Commander Macro Sub-Menu

[TOC]

A) Architecture

When you want to write a script to provide a single function, the common and obvious approach is to write a single in-line program, register the program, and provide a user interface if it is called for. Our Automation Example is multi-functional – more like a system, and we want to leave some of the functionality somewhat fluid. If we consider the system (data and code) and make good architectural choices we can accomplish our goal without making the code overly complex.

[TOC]

B) Execution Model

We will be breaking the execution of our automation functions into three different categories:

1. User Interface Functions
2. Base Class and Functions
3. Pseudo Code

Let's examine these categories briefly:

The **User Interface Functions** are the top level function that are activated from the menus. They are the 'main' function, the Registration Block, and the function that is registered in the Registration Block. We will be deliberately keeping the User Interface Function code sparse. While the UI Functions afford lot of functionality, they are rather fragile and difficult to debug. The approach we will use is to get the UI running and call a function that can be written and debugged independently minimizing the edits to the User Interface.

The **Base Class and Functions** are just normal python classes and functions that can be run from a shell. These functions do not depend upon the gimpfu library could be run in any python shell (including the Gimp-Python Shell).

- If you make a syntax error in the course of an edit, you want to be able to quickly isolate it to a particular line in a particular function. This is easier to do from the shell than when running the program from a GUI.
- Keeping the generic python code separate from the code that depends upon the gimpfu library minimizes the impact that future releases of gimp will have on the scripts. Because the gimp procedure calls are handled by Pseudo Code rather than the Base Class and Functions we have less risk of compatibility from future releases of Gimp within the Base Class and Function.

The **Pseudo Code** is the portion of the overall system functionality that we want to deliberately leave fluid. The functionality of the Pseudo Code is intended to be simple editing steps (which should cover a pretty wide range of edits).

- The types of things that you might do in Pseudo Code could include: copy layers, name layers, run filters, set layer modes, set layer opacity, merge layers, and run PDB functions. The pseudo code can access basic image characteristics and perform operations with them.
- Pseudo Code is simple Gimp-Python code fragments. Because there is no support for indenting, a simple test for how complex your Pseudo Code can become is whether you need to indent or not (note that you can write a simple 'if' statement on one line). In spite of this restriction we will show with some examples that you can accomplish quite a bit of editing with macros.

A final thing that we need to talk about that is not a 'category' of execution but is something that is an important part of our Execution Model is **Scope**. We are only going to touch on a couple of points that affect when lists are defined for dynamic User Interface selection.

- The Gimp User interface widgets allow you to select items from a list.
- You can specify the list within the widget, or pass the widget a list by name **IF** you define the

list outside of the function being called. The list must be defined at the level of the function main().

- By keeping the scope of the User Interface lists at the top level, we are able to use the list name in **both the user interface and in the function being called**. In this way we can use the actual argument being selected rather than its index position in a list.
- An architectural advantage is we create this list with a function that reads a configuration file. We only have to define and maintain this configuration list in one place within our system and use the resulting list in as many places as we want by calling a reading function. This is how we will get new macros to show up in the menus when we add them.
- The following skeletal code fragments illustrate defining a list 'cmdList' at the top level – 'main', and using it within the registration block and function. Because it is defined at the ‘main’ level, we can reference it within the function and registration block. We can recover the argument from the index (passed by the widget) because we are using the same list in both places:

Example - Lists and Scope in Functions

```
cmdList = cmdrReadObj.CommanderMacros()
#
def autoCommander(theImage, cmdListIndex):
    ...
    commanderName = cmdList[cmdListIndex]
    ...
#
register (
    "autoCommander",         # Name registered in Procedure Browser
    ...
    [
    ...
    ( PF_OPTION, "cmdSet", "Select a command", 0, cmdList ),
    ],
main()
```

[TOC]

C) Data Model

We now need to talk about the form and organization of the data that we intend to use. The way that we choose to organize our data can have a dramatic impact on the complexity of the functions that we need to write. In our case we can take advantage of a couple of somewhat more sophisticated data models to make the actual functions fairly straightforward. We will be using “trees” (Python ElementTree) and XML data.

[TOC]

D) Trees and XML Data

Python has several built in data structures such as dictionaries, lists to name a couple. A very powerful library structure that is well suited to our particular needs is a “tree”. A couple of the key features that make them well suited for our application are:

1. Trees have a natural 'hierarchical' feel to them, kind of like a directory structure or the 'folders' of an operating system. The levels of hierarchy can be thought of as ‘containing’ the contents in the lower level.
2. A branch can hold an indefinite number of elements, and those elements can be either a leaf with attributes or a sub-branch to another level. This lends a lot of flexibility with the way we structure of the data.
3. The input / output format is XML, which is not only hierarchical, but it is text so it is human readable and portable to any platform (computer / OS).

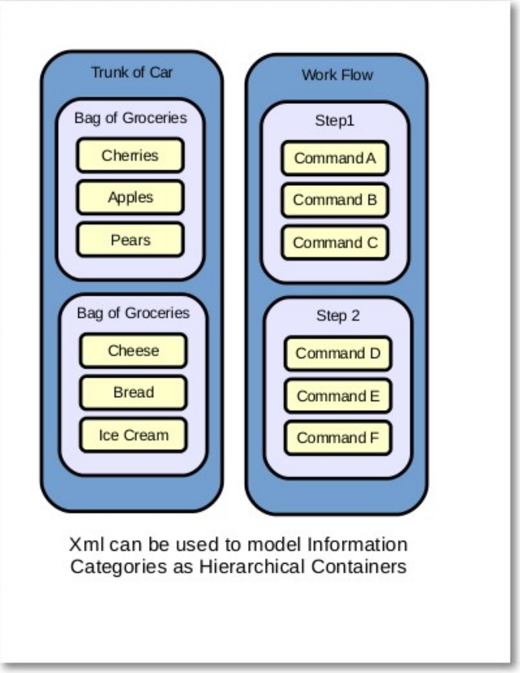


Image - Xml Hierarchy - Containers

The examples use ElementTree to read and write the data between trees and XML. ElementTree is included with Python and described in the Python documentation, so we will not go into detail about the mechanics of tree structures here.

You might be wondering at this point where these XML file will be located. The functions that read and write the XML files are expecting to find the XML under a directory named 'myXml' which you will have to create under you user gimp directory. If you are using Linux and your home directory is 'stephen' the path would look something like '/home/stephen/.gimp-2.8/myXml'. If you are using Windows the path would look something like 'C:\Users\stephen\.gimp-2.8\myXml'. We will be dealing with a couple of types of pseudo code and xml files, and those will be keep in separate directories under myXml, but we will get to that in a bit.

[TOC]

4) Macro Implementation – Revisited

Now that we have talked about the execution model and data model, we can revisit the "Implementation – In Broad Strokes" and tie the steps to the code that is handling them.

A) Pseudo Code to XML

Our first item was to read the pseudo code and parse it into a data structure. That data structure is going to be a Tree.

We can begin writing a pseudo code file by copying and pasting a set of commands from the gimp python console into a text file whose name ends in ".def". The ">>>" preceding each command will be the "keyword" to indicate a command.

The pseudo code will be in a file, in this example it is named NormalGridCanvas.def. Each line begins with a keyword. Keyword choices are: "commander>", "macro>", "comment>", or ">>>".

The class XmlGenerator() in autoBase.py contains a function GenCommanderXml() which reads all of the *.def files in ~/.gimp-2.8/myXml/commander, inserts the lines into a tree (after removing the keyword), and then writes the tree out to a file named combinedCommander.xml.

The keyword will determine both the "tag" associated with the line of pseudo code, and whether it is a "branch" element (macro name) or a "leaf" element (command or comment). We are assigning both a definition and a level in the hierarchy for each line of pseudo code text as we read it into the tree.

Example - Pseudo Code Example - NormalGridCanvas.def

```
commander>Normal Grid and Canvas
   macro>
      comment>Shrink the Canvas back to fit the layer
      >>>theImage.resize_to_layers()
      comment>Set grid to origin and size = image
      >>>pdb.gimp_image_grid_set_offset(theImage, 0, 0)
      >>>pdb.gimp_image_grid_set_spacing(theImage, theImage.width, 
theImage.height)
```

After all of the *.def files are read into the tree and written back out in the form of an XML file, the formatting is done. Writing out a tree automatically generates all of the containing enclosures, essentially making properly formatting the XML a trivial task. The fragment from combinedCommander.xml illustrates the XML from the pseudo code in NormalGridCanvas.def.

Example - combinedCommander.xml (fragment)

```
<combined>
  Definition
 ...
  <commander>
    Normal Grid and Canvas
    <comment>
      Shrink the Canvas back to fit the layer
    </comment>
    <command>
      theImage.resize_to_layers()
    </command>
    <comment>
      Set grid to origin and size = image
    </comment>
    <command>
      pdb.gimp_image_grid_set_offset(theImage, 0, 0)
    </command>
    <command>
      pdb.gimp_image_grid_set_spacing(theImage, theImage.width, theImage.height)
    </command>
  </commander>
  ...
</combined>
```

** The XML above was run through an online XML pretty printer for readability. The XML from ElementTree is functional, but hard to read.*

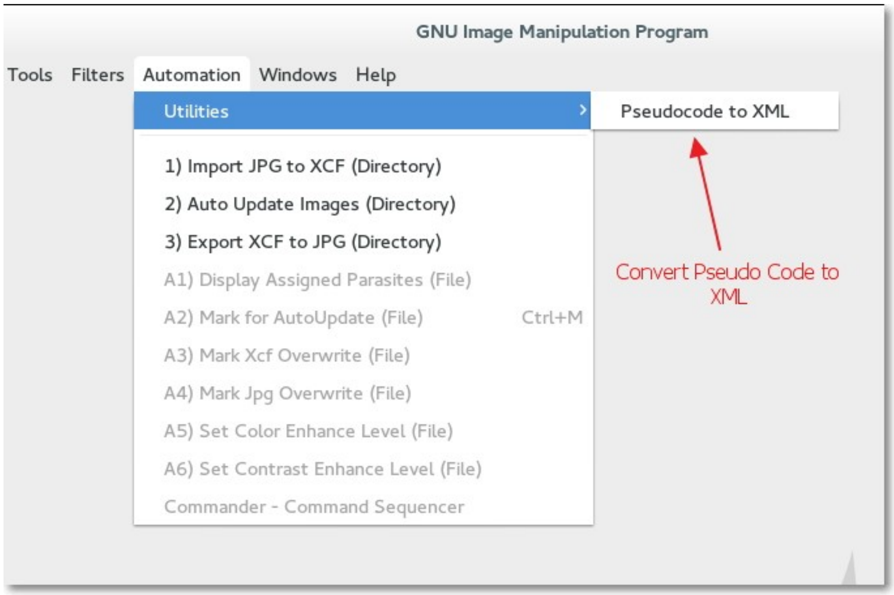
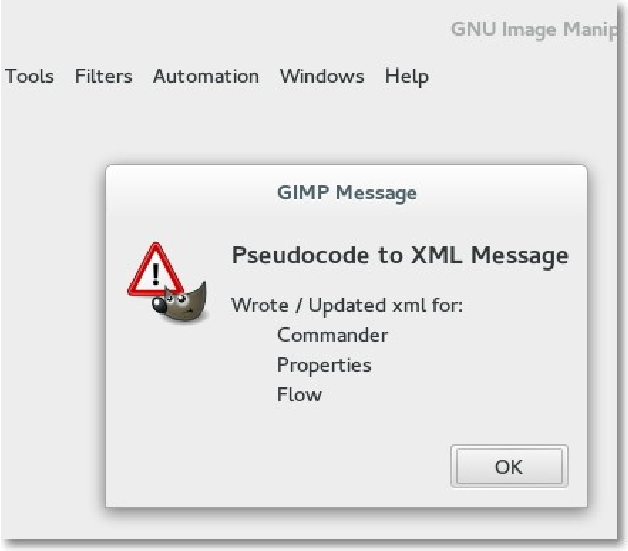

GNU Image Manipulation Program
Tools
Filters
Automation
Windows
Help
Utilities
Pseudocode to XML
1) Import JPG to XCF (Directory)
2) Auto Update Images (Directory)
3) Export XCF to JPG (Directory)
A1) Display Assigned Parasites (File)
A2) Mark for AutoUpdate (File)
Ctrl+M
A3) Mark Xcf Overwrite (File)
A4) Mark Jpg Overwrite (File)
A5) Set Color Enhance Level (File)
A6) Set Contrast Enhance Level (File)
Commander - Command Sequencer
Convert Pseudo Code to XML

*Image - Creating XML from *.def files*

The Xml generator can be called from a GUI menu.



Pseudocode to XML Message

Wrote / Updated xml for:
Commander
Properties
Flow

Image - Xml files built

[TOC]

B) Displaying the Macro Names in a Menu

In our discussion of **Scope** in the Section “Execution Model”, we showed an example code fragment where we created a list “cmdList”. The code was from the example autoCommander.py and uses a class BaseXmlReader and the function CommanderMacros() which resides in autoBase.py.

- The list of Macro Command Names is created by loading the XML file combinedCommander.xml into an ElementTree.
- The tree is traversed at the branch level (the tag <commander>), and the branches text which are the names of the macros are built into a list. The list is essentially built with a “for loop”.
- The list is passed to the widget and used to select the macro you want to use.

[TOC]

C) Running a Specific Macro

The final point to expand upon is how we fetch and run the specific set of commands for a selected Macro.

- We can derive the name of the Macro by way of the menu selection (registration block of autoCommander.py).
- We will again use the BaseXmlReader class, but this time we will utilize the CommanderExtract function passing the Macro name as an argument. The CommanderExtract function traverses the tree branch by branch as before when we were gathering the names of the Macros, except it is comparing the names against the passed argument as we go. When the CommanderExtract function finds a branch matching the argument, it drops down a level to read the leaf elements under that branch.
- The leaf arguments whose tags are set to “command” are appended to the list that will be returned for processing. The leafs whose tags are “comment” will be ignored.

- The newly created returned list will be run through in a “for loop” which will process each line as a separate command using the python “exec” function.

The variable 'theImage' is defined with the autoCommander function and can be used as a handle to access information about the active image by the Macro commands.

[TOC]

D) Commander Macros – Summary

The discussion above has described how we can generate a macro by running a set of commands in the Python Console, paste those commands into a text file adding a name and comments, and then making it available for use.

The code for transforming the pseudo code into a macro is in autoWriteXml.py. The code to display the menu of Macros you have written and execute them is in autoCommander.py. The Classes referenced by these two scripts are in autoBase.py.

The text files that you write for your macro definition need to be put in a directory ~/.gimp-2.x/myXml/commander and have an extension of '.def'. Create a separate *.def file for each macro.

[TOC]

5) An Introduction to Automated Editing

Macros are a terrific way to make the editing process faster and more repeatable, they do though have some limitations.

1. There are times when the order of the editing steps are important.
2. They have to be applied one at a time.
3. You have to keep track of what has already been done and what yet still remains.

A Workflow, or a Process, or a Recipe, what ever you may be used to calling it (I prefer Workflow) can be viewed as a set of ordered steps. These steps would usually correspond to the actions that you would typically code up in a set of Macros. We will automate a Workflow that runs the right steps at the right time and records the editing progress on each of the images.

Let's quickly go over how how capturing and running a “Workflow” is going to work before we dive in. We capture the code fragments that describe the editing process in the same way that we did for the macros. We can either copy and paste from the python console to generate a series of commands, or we could copy them from a working macro.

The steps in a workflow aren't in fact very different than the Commander Macros that we described in the earlier part of this tutorial, the only real difference is they are an ordered set to be followed.

The way that we are going to use the automation tools is a bit different than using macros. When we want to use a macro we are running it on the image that is open in editor window. The automation tools on the other hand run on a directory of images, so we run it without an image being open in the editor window.

We mark the images that are ready to move to the next step of the workflow and then run the automation tools. The automated update looks at each image in the work directory, if it is marked for update, it gets updated to the next step, if it is not marked for update it is left alone.

The automated update will move the marked images to the next step of their assigned workflow essentially keeping track of the 'book-keeping' for us. A time management benefit is if there are several images requiring a filter that takes a bit of time to run, you can go do something else while they are running.

[TOC]

6) Automation Tool Implementation – In Broad Strokes

Let's touch upon the main ideas that we are going to use to implement the automated workflow recording and execution tasks that we are talking about above.

1. The pseudo code will be read and parsed into a data structure. Each pseudo code block (or step) will have a name and consist of commands and comments. Each workflow will have a name and will contain an ordered set of steps or pseudo code blocks.
2. The automation flow will generate a list of images in a given directory. It will then step through that list checking whether an image is marked for update or not.
3. If an image is marked for update, the automation flow will determine the name of the workflow and the name next step to perform on the image. The name of the 'workflow' and the 'next step' will be used to select a list of commands.
4. The list of commands will be run in an 'exec' loop:
5. The 'current step' and 'next step' for the image will be incremented and then saved with the image.

[TOC]

A) Architecture

We commented in the section on macros that good architectural choices can accomplish our goal without making the code overly complex. This argument is as compelling for automated updating of our images. The Architecture and Code for our automated updating tools will be very similar to the ones used for capturing and running the “Commander” macros.

[TOC]

B) Execution Model

As in our discussion of the Macros, we will be breaking the execution of our Automation Functions into three different categories:

1. User Interface Functions
2. Base Class and Functions
3. Pseudo Code

We won't rehash these topics but instead comment on the difference, which is the structure of the pseudo code. The organization of workflows as a set of steps where each step is a set of commands prompts us to organize the pseudo code in a similar manner, where we think of steps containing a sequence of commands, and workflows containing a sequence of steps. In order to reflect this additional level of hierarchy or containment, we will use an additional keyword in the workflow pseudo code.

[TOC]

C) Data Model

There are two special types of data for the automated flow:

1. Trees and XML data for the similar ti the “Commander Macros”.
2. Parasite (or Property) data to keep track of the particular workflow that is being used to edit an image, the next step that is to be used, and whether an image is ready to be incremented to the next step.

[TOC]

D) The Image and Parasites (or Property) Data

The Image type that we will be using for all of our work is the native Gimp *.xcf format. This image format saves all of the layers and modes that we might set while editing and also saves a type of data called Parasites, which are similar to Properties on many other systems.

Parasites are like variable can be referenced by name in order to access their value. Like variables they can be assigned and read. Unlike variables, parasites are persistent, very persistent. A parasite, when assigned to an image, becomes part of the image and is saved with the image file. A parasite that is assigned and saved with an image can be read after gimp is closed and reopened, it is just like any other file data in that respect.

Parasites are also very portable, you can read and write parasites using either the scheme based or python based scripts. They are also independent of the operating system, so you can write a parasite to an image on your Linux Desktop machine, and read it a week later on your windows based laptop assuming that you saved the images in the native gimp *.xcf file format. Parasites can also be written to specific layers, but for our present needs, the image parasites are all we are using.

Because the parasite is associated with the image, and it is persistent until it is overwritten or removed, it is an ideal tool for keeping track of the state of the image's progress in the editing process. Beyond being able to just take notes, the values of the parasites can be used like a property to make decisions and influence the execution of the script that has read the parasite data.

If for example we opened an image that had two parasites (properties), named 'UpdateFlag' and 'Flow', we could use the values from those parasites to make decisions:

Example – Decisions based on Parasite / Property Values

```
UpdateFlag = str(theImage.parasite_find('UpdateFlag'))
Flow = str(theImage.parasite_find('Flow'))
if (UpdateFlag == 'YES'):
      if (Flow == 'Standard'):
            { run commands for Standard flow }
      elif (Flow == 'SemiAuto'):
            { run commands for SemiAuto Flow }
elif (UpdateFlag == 'NO'):
        { do nothing }
```

Reading and writing parasites to an image does have one idiosyncrasy worth comment on which is the format of the data being written. You must express the parasite as an ordered set of 'Name', 'Index', and 'Value'. Name and Value are both strings, and the Index is a small integer, (stay between 1 and 254). If you have not used parasites before you might be wondering how you determine a 'correct' value for the index. You may:

1. Throw a dart at a dartboard and use the result (assuming you hit the board).
2. Or, feel free to use my personal favorite '5'.

As long as you pick a number and preferably stick with it, everything will be fine. When you read the parasite value, the functions in the Scheme scripting interface will give you the 'Name', 'Index', and 'Value'; the functions in the Python scripting interface will only return the 'Name' and 'Value'.

Writing the parasite is called '**attaching**' and reading the value back is called either '**get**' or '**find**' depending on the method you choose to use. You can read and write parasites from within scripts or from either of the python or scheme consoles.

[TOC]

E) Running the Automation Tools on a Workflow

Our final topic about an Automated Workflow in our “Broad Strokes” is how to setup and run the Workflow on a set of images. We are running on a set of images rather than just one so we open up gimp without an image in the editor window. The tools will open up, work on, save, and close the images one by one.

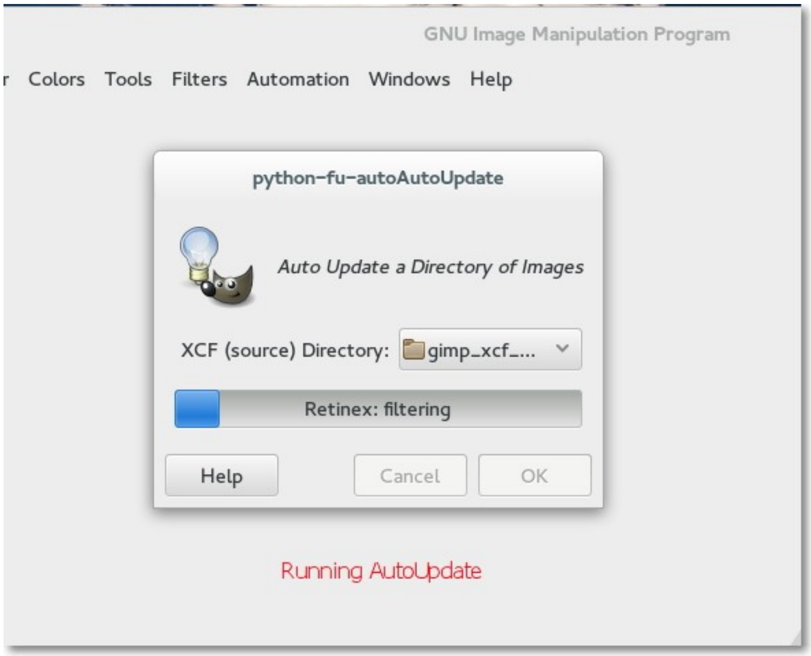
In an earlier tutorial “AUTOMATE CREATION OF XCF FROM JPG” we outlined how to import a directory containing jpeg images into a directory with gimp xcf images. The automation tool implementation modifies the import function to add the assignments of parasites to the images. The images are assigned to a particular flow as they are imported, and the initial set of flow control properties are written and saved as part of the import process. The jpeg-to-xcf function also runs the Automation Process (Auto Update Image) one time to put all of the images on the first step of the assigned flow, so they are ready for their first manual adjustment right after they are imported.

After opening and (optionally) adjusting an image you mark the image as ready for the next step by setting the “UpdateFlag” parasite to a value of “YES”. This is accomplished with a function that is available from the menu: “Automation → A2 Mark for AutoUpdate (File)”. (Note: Since you will be doing this a lot, it is very convenient to use a keyboard shortcut to run this function).

Images are moved to the next step in their flow by running the “Auto Update Images (Directory)” function from the menu. This will increment all of the images whose UpdateFlags are set to YES to the next step in their flow. Note that since each image has a record of its own next step and flow, there is no requirement for the images in a directory to be on the same step or even using the same workflow.

The mechanics for creating XML from pseudo code for the workflows, properties, and commander macros is to run the function Pseudocode to XML from the menu (Automation -> Utilities -> Pseudocode to XML).

The export XCF to JPG function in the “Automation” menu opens each xcf file in the source / work directory and looks at the properties of the image. If the image is “Finished”, at the end of the flow, it is exported. The images that are still being work on are left alone.


GNU Image Manipulation Program
r Colors Tools Filters Automation Windows Help
python-fu-autoAutoUpdate
Help
Cancel
OK

Auto Update a Directory of Images

XCF (source) Directory: gimp_xcf_....

Retinex: filtering

Running AutoUpdate

Image - Running AutoUpdate

[TOC]

7) Automation Tool Implementation – Details

A) Pseudo Code

i. Workflow Pseudo Code

You can generate the list of commands that we wish to perform on an image in the python console and when you have it working just right for the step you would like to perform you can copy and paste them into a pseudo code text file. The “Pseudocode to XML” function will be looking for files that have the file extension “.def” so it will convert all of the *.def files in the flow directory into XML.

As in the case of the macros, you can put in comment lines beginning with “comment>” and blank lines to make the code fragments more readable and easier to understand when you come back to make enhancements in a couple of months.

Each set of commands is contained by a “Step” which uses the key “step>”. The top level container is the Workflow which uses the key “flow>”. Each workflow can be specified in its own “.def” file. The “Pseudocode to XML” function will read all of the *.def files and create a single XML file named combinedFlow.xml in the myXml/flow directory.

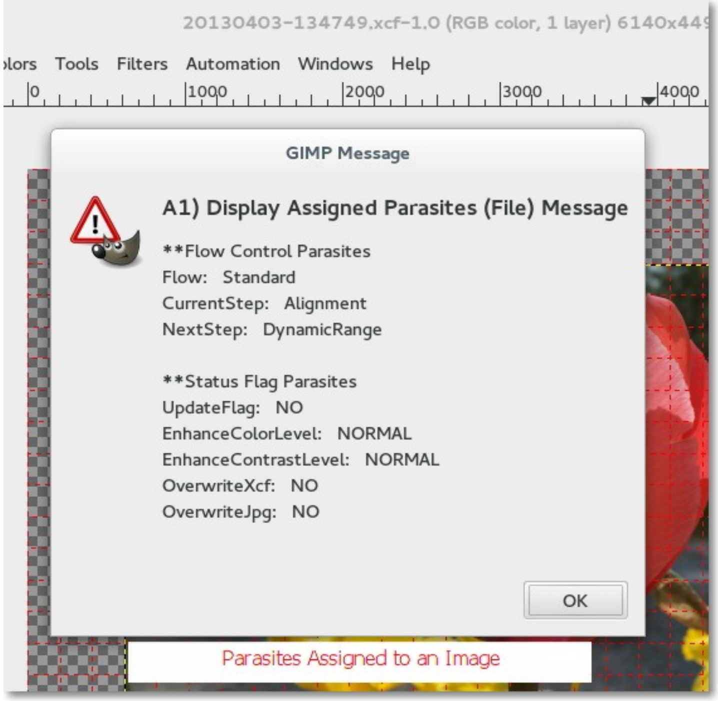
[TOC]

ii. Property / Parasite Pseudo Code

The other type of pseudo code that we need to talk about is the code for properties. The pseudo code for properties is contained in the “myXml/property” directory. The file flagProperties.xml is created by the “Pseudocode to XML” function from the flagProperties.def file. In the case of properties it only really makes sense to have one set of properties for all flows. The properties defined in the flagProperties.xml file will be the “flag” properties. You can set the property name, comments, the option values, and the default value (initial setting). The “property>” key sets the property name and contains the other property values within the XML. The other keys are “comment>”, “default>”, and “options>”. The key “flags>” with the name Control Properties is used at the beginning of the file to define the top level container.

There are three properties that are assigned by the automation scripts and are not normally edited by the user or defined in the “.def” file. They are created, read and modified by the scripts. These are the flow control properties, “Flow”,   “CurrentStep”, “NextStep”.

You can see all of the properties and current assigned values for a particular image using the menu function “Automation” -> “A1) Display Assigned Parasites (File)”.


20130403-134749.xcf-1.0 (RGB color, 1 layer) 6140x449
olors Tools Filters Automation Windows Help
0
1000
2000
3000
4000
GIMP Message
OK
Parasites Assigned to an Image

A1) Display Assigned Parasites (File) Message

```
**Flow Control Parasites
Flow:   Standard
CurrentStep:   Alignment
NextStep:   DynamicRange

**Status Flag Parasites
UpdateFlag:   NO
EnhanceColorLevel:   NORMAL
EnhanceContrastLevel:   NORMAL
OverwriteXcf:   NO
OverwriteJpg:   NO
```

Image - Assigned Parasites

[TOC]

B)   Properties and Image State – Flow Control Parasites

One way to think of a workflow is as a series of states. These states are what we have been referring to as steps. As the image evolves it transitions from one state to another it moves through the workflow from beginning to end, state by state (or step by step).

The Flow Control Parasites provide a method to make each image “self aware” of its own state, which is its assigned “Flow” and “CurrentStep”. It is also aware of the next step to which it will proceed, which is determined by its assigned “Flow” and “NextStep”. The order for the steps is determined by the order they are listed in the pseudo code. When an image is updated the “NextStep becomes the new “CurrentStep” and a new “NextStep” is looked up and written to the image as a parasite.

Let's examine the steps of the “Standard” flow example that is included with this tutorial. The steps the image will go through are:

Example – States or Steps in the Standard Flow Example

```
    CurrentStep        NextStep
 1. First              Alignment
 2. Alignment          DynamicRange
 3. DynamicRange       Retinex-Filter
 4. Retinex-Filter     Sharpen
 5. Sharpen            ColorAdjust
 6. ColorAdjust        FINISHED
 7. FINISHED           FINISHED
```

The state "First" is assigned by the Jpeg to Xcf function. This step is assigned automatically regardless of the flow. The steps following the "First" step, "Alignment", "DynamicRange", "Retinex-Filter", "Sharpen", and "ColorAdjust" are assigned through the Xml representation of the flow. The Step "FINISHED" is assigned automatically when the end of the list of steps is reached.

When the images are first imported, the autoUpdate function is run to move the images from the automatically assigned "First" step to executing the first real step of the flow. When the CurrentStep becomes "FINISHED", the image is ready to be exported by the Xcf to Jpeg function.

[TOC]

C) *Status Flag Parasites*

The Flag Parasites are for decision making. They will be assigned with a default value when the image is imported into a flow. The parasite whose function is flow control is the "UpdateFlag". The value of the UpdateFlag is read to determine if an image is ready to be moved to the next state.

Flag Parasites other than the UpdateFlag can be modified or eliminated to suit the needs of your pseudo code functions. You could for example add Flags with values that determine certain layer Opacity, or determine whether certain filters will be run. There are a lot of possibilities for this powerful capability.

[TOC]

D) *Automation Workflow - Summary*

Running an Automated Workflow is almost trivially easy, once you have it set up to your liking. There is some work to be sure in setting up a workflow, but the reward is a consistent and apparent (obvious which steps have been run) workflow or process.

There are several things you may need to set up Gimp Preferences in order to optimize the operation.

1. The example flow, for example adjusts and uses the grid. You need to turn on the grid by default before you import all of your images into gimp xcf format for this to be useful.
2. There are some commands you will use so frequently that you will want to assign keyboard shortcuts just so you don't have to keep pulling down the menu each time.

[TOC]

8) Conclusion

The combination of an available programming language, well suited data structures, the ability to affix properties to images, and a rich set of editing features offer powerful possibilities for automating the editing process.

Using an automated workflow has changed the way that I use Gimp for editing my photos. I hope that you can leverage these examples to make working with Gimp more productive for yourself. Links to the example scripts and pseudo code are in the following appendices.

[TOC]

Appendix

1) Appendix – Notes

The following Appendices contain notes which are more specific to setting up the example scripts, the example *.def files, and comments on debugging.

A) Setting up the Example Scripts

All of the example scripts begin with “auto”, e.g. autoAutoUpdate.py, autoBase.py, ... If you try them but then decide you don't like them they should be pretty easy to find and remove. The following example scripts should be loaded into your gimp/plug-ins directory. Something like /home/stephen/.gimp-2.8/plug-ins if your user name is stephen and you were using gimp 2.8. Click on the filename to download.

1. **autoAutoUpdate.py** - Runs the auto update function on a directory of images.
2. **autoBase.py** - Contains the classes that read and write the XML files that affect how the update works.
3. **autoCommander.py** - Runs the 'Commander' macros.
4. **autoJpegToXcf.py** - Imports the images into xcf format and assigns properties to image.
5. **autoRWparasites.py** - User Interface functions to read and write image parasites from the menu.
6. **autoWriteXml.py** - Reads *.def files and generates XML for commander macros, workflows, and properties.
7. **autoXcfToJpg.py** – Exports the finished images back to jpeg format.

[TOC]

B) Setting up the Example Pseudo Code

Underneath your gimp directory (something like /home/stephen/.gimp-2.8) you need to create a directory named 'myXml'. Don't get creative here, the scripts are looking for this specific directory. It will be in the same directory that contains your plug-ins directory. Underneath the myXml directory create three more directories, 'commander', 'flow', 'property'. These will be where your pseudo code and three kinds of XML will be located.

[TOC]

i. Pseudo Code for Commander Macros

Copy the following example *.def files into the “commander” directory (/home/stephen/.gimp-2.8/myXml/commander – assuming a home directory of /user/stephen and a gimp version 2.8). They are example Commander Macros pseudo code files. Click on the filename to download.

1. **centeredgrid.def**
2. **colorAdjust.def**
3. **createColorLayer.def**
4. **createDynamicRangeLayer.def**
5. **expandCanvas.def**
6. **normalGridCanvas.def**
7. **renameBaseLayer.def**
8. **retinexLayer.def**
9. **sharpenLayer.def**

Hopefully with the comments and by running them, their function will be apparent. They should be enough to get you started writing some macros of your own.

When you run the “Pseudocode to XML” Utility function, it will read all of the *.def files in this directory and write an XML file in this directory called “combinedCommander.xml”. “combinedCommander.xml” is the file that is accessed to list and run all of your macros.

[TOC]

ii. Pseudo Code for Automation Workflows

Copy the following *.def files into the “flow” directory (/home/stephen/.gimp-2.8/myXml/flow). They are example Workflow pseudo code files.

1. **fullauto.def**
2. **semiauto.def**
3. **standard.def**

These three workflows all follow the same basicsteps of the standard workflow. The semiauto and fullauto workflows combine some of the steps. The idea is to give you a couple of different workflows to play with. The fullauto illustrates that you really can pack a lot of editing into a “step” but is probably too automatic to be of practical use.

When you run the “Pseudocode to XML” Utility function, it will read all of the *.def files in this directory and write an XML file in this directory called “combinedFlow.xml”.

[TOC]

iii. Pseudo Code for Properties

Copy the following *.def file into the “property” directory (/home/stephen/.gimp-2.8/myXml/property). It is an example Property pseudo code file (for Flag Properties / Parasites).

1. **flagProperties.def**

When you run the “Pseudocode to XML” Utility function, it will read *.def file in this directory and write an XML file in this directory called “flagProperties.xml”.

[TOC]

C) *Pseudo Code Syntax*

i. Commander Pseudo Code

There are three keywords used for writing commander pseudo code:

1. **commander>** - The text following this keyword is the Macro Name. This keyword must be the first keyword in the file. This is the “container” or the root of the tree for the following comments and commands.
2. **comment>** - The text following this keyword is for descriptive comments. The comments will be represented in the pseudo code *.def file and in the resulting XML. When the XML is read for processing, comments will be ignored.
3. >>> - The text following this keyword is taken as a python statement. Then the resulting XML is read, the command statements will be passed to the commander script to be processed in order.

Note that lines beginning with “#” are ignored. You may indent if you like for readability. Leading white space is stripped off.

Example - Commander Pseudo Code Example

```
commander>Centered Grid
   comment>** Set up the grid for Rotate and or Perspective Transform
   comment>*    Set values from python-fu image object
   >>>centerX = theImage.width/2
   >>>centerY = theImage.height/2
   >>>gridSpacing = max(theImage.width, theImage.height)/24
   comment>*    configure grid with PDB functions
   >>>pdb.gimp_image_grid_set_offset(theImage, centerX, centerY)
   >>>pdb.gimp_image_grid_set_spacing(theImage, gridSpacing, gridSpacing)
   >>>pdb.gimp_image_grid_set_style(theImage, GRID_ON_OFF_DASH)
```

[TOC]

ii. Property Pseudo Code

There are five keywords used for writing property pseudo code:

1. **flags** – The text following this keyword is the top level container, or in other words, the root of

the tree.

2. **property** – The text following the keyword is the name of the property / parasite. This is a second level container or a branch of the tree. It will contain all of the following leaf keywords (comment, default, and option) until the next property statement.
3. **comment** - The text following this keyword is for descriptive comments. The comments will be represented in the pseudo code *.def file and in the resulting XML. When the XML is read for processing, comments will be ignored. The comments are leafs of the tree.
4. **default** – The text following this keyword is the default property value. The default value is a leaf of the tree.
5. **option** – The text following this keyword is one of the possible property values. There can be several option values for any given property. The option values are leafs of the tree.

Example - Property Pseudo Code Example

```
flags>Control Properties
property>UpdateFlag
    comment>Initial value set on import to Xcf
    comment>Set by user on Image from Automation Menu
    comment>Read by autoAutoUpdate (updateImage function)
    comment>Updates Image (executes Next Step in Flow) if YES
    comment>Reset to NO by updateImage
    default>YES
    option>NO
    option>YES

property>EnhanceColorLevel
    default>NORMAL
    option>EXTRA
    option>NORMAL
    option>MID
    option>NONE
```

[TOC]

iii. Flow Pseudo Code

There are four keywords used for writing flow pseudo code:

1. **flow** - The text following this keyword is the top level container, or in other words, the root of the tree. This is the name of the flow.
2. **step** – The text following this keyword is the first level branch from the root. The step contains comments and commands. The steps select groups of commands for execution.
3. **comment** – Comments are leafs of the tree.
4. >>> - Commands are leafs of the tree. When the Xml is read the commands will be processed in the order in which they appear in the step.

Lines beginning with a “#” are ignored. Leading white space before the keywords is stripped out. White space after the keyword is stripped out.

[TOC]

D) Running code in the Gimp Python Console

The Python-Fu console is a python shell in which you can run not only the gimp pdb functions, but your own python functions as well.

First, set up and verify Python Path to include your user plug-ins directory

Example – Setting the Python path in the Python-Fu Console

```
>>> import sys
>>> sys.path.append('/home/stephen/.gimp-2.8/plug-ins/')
>>> sys.path
```

[echos back a list of paths that include path added above]

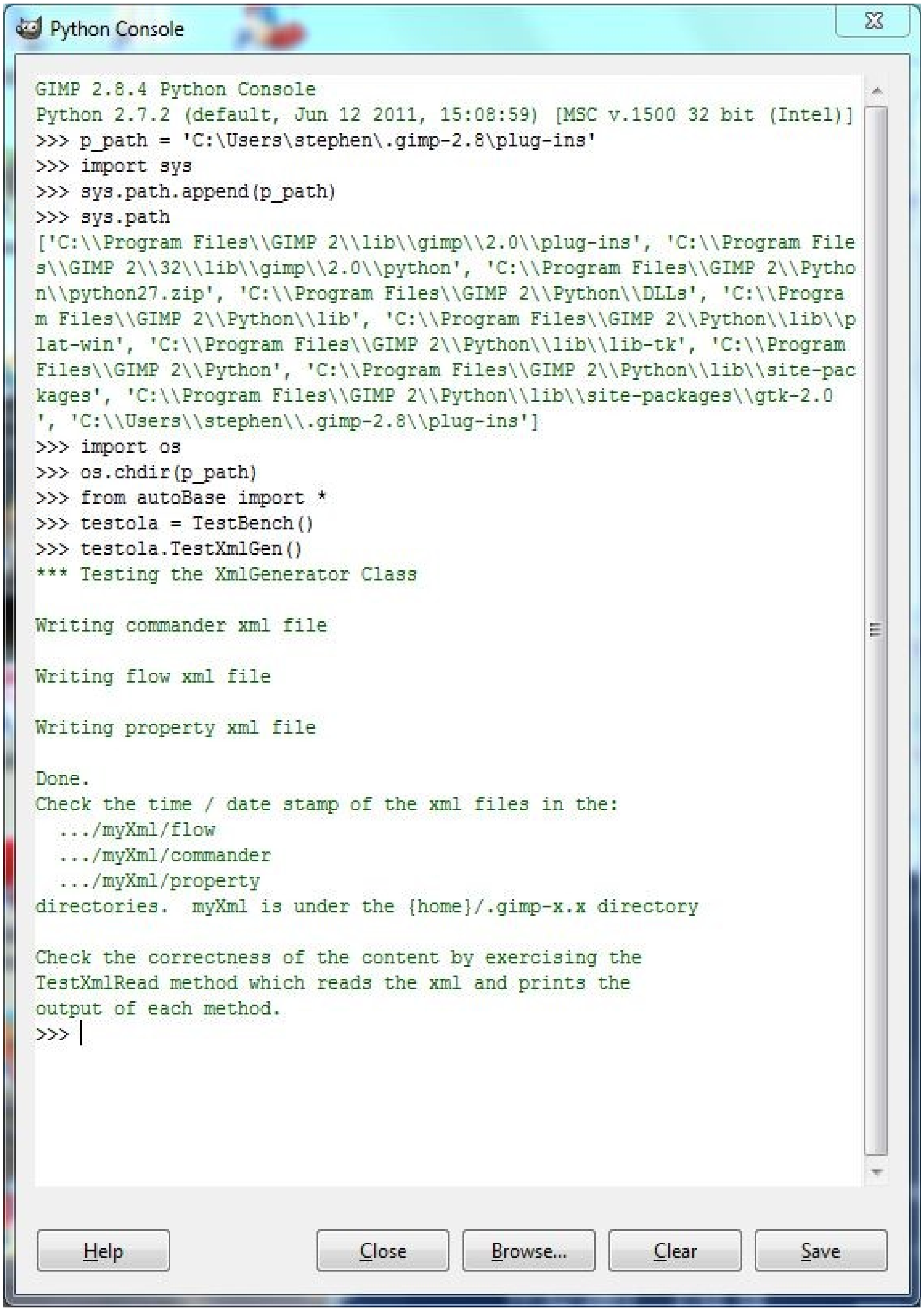
Next, run your python functions in the Gimp Python-Console . This example uses the 'TestBench' class to run functions in the other classes in the autoBase.py module. Object instances of the TestBench class echo back results to the screen.

1. Set the working directory to the user plug-ins directory
2. Import the autoBase module functions
3. Create and instance of the TestBench class
4. Run the TestXmlGen and TestXmlRead functions

Example – Running your own Functions in the Python-Fu Console

```
>>> import os
>>> os.chdir('/home/stephen/.gimp-2.8/plug-ins')
>>> from autoBase import *
>>> testola = TestBench()
>>> testola.TestXmlGen()
>>> testola.TestXmlRead()
```

The screen shot below illustrates the process on the Windows version of Gimp / Python Console (TestXmlGen is pictured, TestXmlRead produces several pages of output):



```
GIMP 2.8.4 Python Console
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
>>> p_path = 'C:\Users\stephen\.gimp-2.8\plug-ins'
>>> import sys
>>> sys.path.append(p_path)
>>> sys.path
['C:\\Program Files\\GIMP 2\\lib\\gimp\\2.0\\plug-ins', 'C:\\Program File
s\\GIMP 2\\32\\lib\\gimp\\2.0\\python', 'C:\\Program Files\\GIMP 2\\Pytho
n\\python27.zip', 'C:\\Program Files\\GIMP 2\\Python\\DLLs', 'C:\\Progra
m Files\\GIMP 2\\Python\\lib', 'C:\\Program Files\\GIMP 2\\Python\\lib\\p
lat-win', 'C:\\Program Files\\GIMP 2\\Python\\lib\\lib-tk', 'C:\\Program
Files\\GIMP 2\\Python', 'C:\\Program Files\\GIMP 2\\Python\\lib\\site-pac
kages', 'C:\\Program Files\\GIMP 2\\Python\\lib\\site-packages\\gtk-2.0
', 'C:\\Users\\stephen\\.gimp-2.8\\plug-ins']
>>> import os
>>> os.chdir(p_path)
>>> from autoBase import *
>>> testola = TestBench()
>>> testola.TestXmlGen()
*** Testing the XmlGenerator Class

Writing commander xml file

Writing flow xml file

Writing property xml file

Done.
Check the time / date stamp of the xml files in the:
  .../myXml/flow
  .../myXml/commander
  .../myXml/property
directories.  myXml is under the {home}/.gimp-x.x directory

Check the correctness of the content by exercising the
TestXmlRead method which reads the xml and prints the
output of each method.
>>> 
```

Image - Running your code in the Gimp Python Console

[TOC]