PYTHON SCRIPTING WITH SCRIBUS

by Gregory Pittman
Published : March 2018

WHAT THIS BOOK IS ABOUT

Scribus has become a robust application for the creation of PDFs, whether this be intended for printed publication in some format, distribution via the internet, creating of presentations, or other instances where the features of a PDF for showing well-constructed layout is either desirable or essential.

At this time, Scribus has a large number of built-in functions which allow for precise control over the layout and appearance of your PDF. Among these has been the ability to use Python to create small programs, or scripts, designed to automate tasks, especially when these might be done repetitively, either in the same document or perhaps commonly performed in a variety of documents.

What I intend to do in this book is to use a number of example scripts to break down scripting into its elements to show how to approach what might end up being a more complicated task you have in mind. In the process, I will try to explain each script so that you can more easily understand its components and the overall structure of the scripts. A constraint that you will find is that, while there are quite a number of Python commands in Scripter, there are only certain kinds of information or operations you have access to. Sometimes there are opportunities for working around some obstacle, but in some there simply is no way to accomplish some task with a script. Knowing these barriers is part of the design of a useful script.

For the most part, the scripts in this book come from the large number of them that I have written over the years. Some of these come from my own needs or wants, some come from user questions on the Scribus mail list, and there are also some that just were some intellectual curiosity I had about whether something could be done with Python. I have tried to select scripts that illustrate various methods for creating and editing documents.

As I have said a number of times on the Scribus mail list, there are also some tasks which will lend themselves to working on the main canvas rather than trying to do them with Python, so one should also become familiar with the capabilities of Scribus without scripting.

All of the fragments of scripts and complete scripts in this book are devoid of any restrictions, and may be copied, used, and modified as desired by the user. This includes the script colorchart_generic.py, which was made in cooperation with freieFarbe e.V. If you make something good, try to share it with others.

THE VERY BEGINNINGS

Just what is a Python script anyway? Python is one of the programming languages which relies on a real-time compiler for its implementation. What this means on a practical level is that you create a plain-text file, which you then "run" by calling up the Python interpreter. Let's look at this very simple Python script, to be run outside of Scribus.

```
#!/usr/bin/env python
print "Hi! What's up?"
```

Using a plain-text editor, which in Linux might be vi, or vim, or emacs, or kwrite, gedit, among others. In Windows, I might use Notepad. The important thing is that I don't want an editor that formats the text in any way. So I make this in the editor, then save as "Hi.py". You don't have to use this .py extension, but it will help you find Python scripts if you do, and you might as well save them in a directory named python, so you can find them later. So now what do you do?

Answer: you "run" it. In a command line setting, you type

```
python Hi.py
```

and press RETURN, assuming you're in the directory where it is located. What you have done is to call up the Python interpreter, and pointed it to the text file you want it to interpret. You should then see on the next line

```
Hi! What's up?
```

All it does is print this message. Exciting, huh? Let's talk about some other elemental things for your script. Look at these modifications.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# Hi.py -gives a simple message
print "Hi! What's up?"
```

Notice how the first three lines begin with a '#'. Generally speaking, this denotes a comment, in other words, a line that will not be interpreted as Python by the compiler. The exception is the first line, which begins with '#!', called 'she-bang', and what follows on that line is where your computer should look for the compiler to run this script. You actually wouldn't need to specify python in the command line if you can changed the permissions of this file to allow it to be executed. In that case you would just put on the command line

```
./Hi.py
```

and it would run. The next line is something mainly used in Linux, which explains that the character encoding of the script is UTF-8, mainly important for indicating new lines in the script. The third line is informational, indicating the name of the script file, and what it does, not so necessary in a simple script like this but very useful in a large script you haven't looked at in a while. You can have as many comments as you want in a script, and a line doesn't have to begin with a '#', but the compiler is going to ignore anything to the right of the '#' on that line.

So far we haven't begun to think about Python and Scribus, but let's move on to that.

SOME BASIC ELEMENTS OF A SCRIBUS SCRIPT

In this chapter I want to show some very basic things to include in a script which you want to run in Scribus.

```
# -*-coding: utf-8 -*
# basic.py

import scribus
```

Especially if you're writing your script on Linux, you should include this first line. Notice that I have not included the line telling where to find the Python compiler. If you're running a script within Scribus, that is totally unnecessary. An absolutely essential line tells Python to '**`import scribus`**'. Alternatively, you may also see scripts which instead say '**`from scribus import *`**'. This latter method is less preferred, since what it does is pull all of Scribus's Scripter commands into the compiler. If you say '**`import scribus`**' each command is used only as it's needed, but we'll see shortly how we need to change our commands to allow for this.

HANDLING ERRORS

What if you tried to run such a script outside of Scribus? You will get a message something like this:

```
Traceback (most recent call last):
File "Hi.py", line 5, in <module>
import scribus
ImportError: No module named scribus
```

Now look at this next example, showing only lines after the initial comments:

```
import sys

try:
import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
if not scribus.haveDoc():
    scribus.messageBox('Scribus -Script Error', "No document open",
          scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(1)
```

We may not need to 'import sys' for our compiler, yet it's safer to include this line. Next we use this try: clause, which if it fails, gives us a useful message while it calmly exits the script. After this, we finally get to some actual Scripter commands, **`haveDoc()`** and **`messageBox()`**.

IMPORTANT: The first thing to notice is that we must prefix these with **`scribus.`**, so that the compiler knows to look to Scribus for these. If we had instead said '**`from scribus import *`**', then we would NOT use this prefix.

The line '**`if not scribus.haveDoc()`**' is used when the script requires that a document already be present, and perhaps more likely than not we are running a script on an existing document. In this case, if there is no document open, we make use of a very handy tool, a **`messageBox()`**, which shows the message 'No document open', with a warning icon and an OK button to close it. Once we click Ok, the script exits. If we didn't use this message, the script would still fail, but it might not be obvious why, since the generic error messages from Python can be confusing.

INDENTATION

Something else worth pointing out in this example above is the importance of indentation in Python. Unlike some other programming languages like C or Perl, Python does not use a semicolon or other character to indicate the end of a line, it's just the newline character itself. Python also doesn't use curly brackets '`{}`' to enclose conditional commands. Certain test commands like **`try:`** or loops like **`while 1:`** require that the commands which are executed as a consequence of these commands are indented and line up at their left margins. If something is out of line you will get an error message from the compiler. You don't HAVE to have the same amount of indentation in one group that you have in some other, but for the sake of neatness and ease of understanding the scripts, it's good practice to have each layer of indentation be the same amount.

A REAL SCRIPT: BCAPTION.PY

```
# -*-coding: utf-8 -*


"""
bcaption.py
Creates a text frame (caption) below one or more selected frames.
"""

try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
numselect = scribus.selectionCount()
count = 0
frames = []
if numselect == 0:
    scribus.messageBox('Selection Count', "You must have at least one image
      frame selected",scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
pageunits = scribus.getUnit()
scribus.setUnit(scribus.UNIT_POINTS)
while count < numselect:
    frames.append(scribus.getSelectedObject(count))
    count += 1
for frame in frames:
    fwidth, fheight = scribus.getSize(frame)
    fx, fy = scribus.getPosition(frame)
    textf = scribus.createText(fx, fy+fheight, fwidth, 24)
scribus.setUnit(pageunits)
scribus.setRedraw(True)
```

This is an early form of a script, caption.py, which was later included with Scribus. The idea is to automatically create a small frame underneath another frame, in which a caption might be entered. While this might most commonly be used for image frames, I didn't see any reason to restrict its use to that. Another feature is that one can do multiple frame captions at once, simply by selecting a number of them. You do this on the main canvas by holding down the SHIFT key while clicking them sequentially, and they can be on more than one page. After the initial check to see if the script is running under Scribus, a count of the number of frames selected is taken, a counter is indexed to zero, and an empty Python list is created (frames). In case the user forgot to select any frames, there is a check for this.

Next comes a step which is a good idea any time you are using and changing object measurements in absolute numbers. Personally, I prefer to work in points, because of the precision you can easily achieve. We first use the command **`getUnit()`** to make note of the user's page units, then **`setUnit()`** to change to points. **`UNIT_POINTS`** is a built-in Scribus variable that ishandier to use than looking up what that constant is. Notice that when we use a built-in variable it requires this **`scribus.`** prefix, so that Python knows where to find it. After we finish whatever we need to do in the script, it's important that we set the units back to what they originally were with **`setUnit(pageunits)`**.

The next little clause after setting the units to points is to fill in our list of selected objects by using the append list function of Python. Once this is done, one by one we first get the dimensions of the object with **`getSize()`** and then its position on the page with **`getPosition()`**. Notice how each of these assigns two variables, since each command returns two variables. InPython lingo, this is called a tuple. Something else to notice is how we create variable names that have some meaning, like **fwidth** and **fheight** so that we're less likely to make mistakes later, and also so that someone else reading the script can easily understand what these represent. We're also helping ourselves, since if we save this file and then open it years later, we can more easily understand our own work, with the added benefit that we can also understand what these Scribus commands return.

Finally, the real point of the script, creating the caption frame at the bottom of the original frame, with the same width, and a height of 24 points. The last command, **`setRedraw(True)`** is there to ask Scribus to redraw the document after everything is finished. I probably should have (and usually do) put in a line saying **`scribus.setRedraw(False)`** somewhere early in the script, which is supposed to suppress any redrawing of the document while the script runs, theoretically slowing it down. I'm not really sure how effective the **`setRedraw()`** command is, though. Later we'll see what I mean in the **centervert.py** script. Something else to mention is that we do not need to say **`scribus.True`** or **`scribus.False`** since these are recognized by Python for boolean operations.

CAPTION.PY

Here is the version which is included with Scribus.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
"""
caption.py

Creates a text frame (caption) in selected location relative to
one or more selected frames.

USAGE

Select one or more object (can be any kind of frame, or even imported
  vector graphics), run script. Dialog appears for choosing placement of all
  caption frames B/T/R/L (Bottom/Top/Right/Left).
Bottom and Top caption frames are sized to width of parent object and
  height of 24 points. Right and Left caption frames sized to 150 points
  width and 40 points height.
"""
try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
numselect = scribus.selectionCount()
count = 0
frames = []

if numselect == 0:
    scribus.messageBox('Selection Count', "You must have at least one
         object selected", scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)

captionloc = scribus.valueDialog("Caption Location","Where to put the
    caption(s) -\n B/T/R/L?","b")
captionloc = captionloc[0]
location = captionloc.upper()
```

```
  pageunits = scribus.getUnit()
  scribus.setUnit(scribus.UNIT_POINTS)
  while count < numselect:
      frames.append(scribus.getSelectedObject(count))
      count += 1
  for frame in frames:
      fwidth, fheight = scribus.getSize(frame)
      fx, fy = scribus.getPosition(frame)
      if location == "B":
        textf = scribus.createText(fx, fy+fheight, fwidth, 24)
      elif location == "T":
        textf = scribus.createText(fx, fy-24, fwidth, 24)
      elif location == "R":
        textf = scribus.createText(fx + fwidth, fy, 150, 40)
      elif location == "L":
        textf = scribus.createText(fx-150, fy + fheight -40, 150, 40)
  scribus.setUnit(pageunits)
  scribus.setRedraw(True)
```

I've highlighted the parts that differ in bold. The idea behind these changes was to allow for the possibility of having a caption frame not just at the bottom of the selected object, but perhaps at the top, or to the left or right. We use this **`valueDialog()`** to allow for user input. As you can see, there are three elements between the parentheses of this command. The first gives a title to the dialog, the next is a little message telling the user what this input is about, in this case asking for a single letter to use for Bottom, Top, Right, or Left location for the caption frame, and lastly the default value is b. These are all strings, thus the enclosing quotation marks. I purposely put a lower case b for the default so that the user would hopefully understand that he can enter either an upper or lower case letter. Afterward, I use the **`upper()`** string method to make the result only upper case.

In the lower bolded text we see how we put this into action. If I hadn't converted the **`valueDialog()`** entry to upper case, I would have had to have, for example

```
  if ((location == "b") or (location == "B")):
```

So some extra typing is saved. For the bottom location, you can see we do exactly what was done in the **bcaption.py** script, and for the other choices we change the math and settings as needed.

This is also a good time to add a comment about multiline comments. For individual lines you enter '#' and then what follows is your comment. For a multiple line comment, you can enter 3 double quotes on a line, then on the following lines type your extended comment, after which to type 3 double quotes to end the comment. Scribus uses this block of comment when, from the menu, you select Scripter > About Script..., after which you find the script of interest with a file dialog. A window then displays the text between the sets of quotes.

CENTERVERT.PY

Here is a script that seemed like a good idea at the time. The problem it solved in its own way was to figure out how to center text vertically in a frame. Scribus has always had center justification to place the text horizontally in the middle, but there wasn't any way other than manually entering enough lines or space somehow before your text for vertical centering. What this script does is to get the linespacing, multiply that by the number of lines of text, and then adjust the top distance (the space between the top edge of the frame and the first line of text) so that the space above and below the text is even. I don't know if anyone other than me ever used this script, but subsequently a new setting was added for text in the Columns and Text Distances subtab of Text Properties, *Vertical Alignment*, with choices of Top, Middle, and Bottom. At any rate, it is useful for showing how Scripter handles text distances.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*

"""
centervert.py
Centers text vertically in a text frame. Works with fixed linespacing.
USAGE
Select a text frame, run script.
"""
try:
    import scribus

except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
if not scribus.haveDoc():
    scribus.messageBox('Scribus -Script Error', "No document open",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(1)
if scribus.selectionCount() == 0:
    scribus.messageBox('Scribus -Script Error', "There is no object
           selected.\nPlease select a text frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
if scribus.selectionCount() > 1:
    scribus.messageBox('Scribus -Script Error', "You have more than one
         object selected.\nPlease select one text frame and try again.",
         scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
textbox = scribus.getSelectedObject()
pageitems = scribus.getPageItems()
scribus.setRedraw(False)
for item in pageitems:
if (item[0] == textbox):
    if (item[1] != 4):
      scribus.messageBox('Scribus -Script Error', "This is not a textframe.
        Try again.", scribus.ICON_WARNING, scribus.BUTTON_OK)
      scribus.setRedraw(True)
      sys.exit(2)

    lines = scribus.getTextLines(textbox)
    distances = scribus.getTextDistances(textbox)
```

```
      linespace = scribus.getLineSpacing(textbox)
      dimensions = scribus.getSize(textbox) # (width, height)
      if (scribus.textOverflows(textbox, 1) > 0):
         scribus.messageBox('User Error', "This frame is already overflowing",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
         scribus.setRedraw(True)
         sys.exit(2)
      newtopdist = (dimensions[1] -linespace * lines)/2
      scribus.setTextDistances(distances[0],distances[1],newtopdist,
             distances[3],textbox)
      scribus.moveObject(-4,0,textbox)
      scribus.moveObject(4,0,textbox)
   scribus.setRedraw(True)
```

Something you might notice right away, once you get past the comments, is that about half of the actual commands are for error-handling. First we check to see the script is being used with Scribus, next we check if a document is open, next we check to make sure an object is selected, and then we check to make sure only one is selected, and we're not even done with error-checking!

We've assigned our object to this variable textbox, but actually we don't know if this is a text frame or not. This is important, since we have operations ahead that will only work for text frames. We sort this out by making a Python list of all the objects on the page, then going through the list until one matches our textbox. It turns out that these individual pageitems are themselves lists. The first element, **`pageitem[0]`**, is its name. The second, **`pageitem[1]`** is its type (aha!), and the value of **`pageitem[1]`** should be 4 if it is a text frame. So now you can see the reason for collecting, then cycling through the pageitems. In the end, if this is not a text frame, we generate another error message and quit.

We begin the actual work of centering the text by getting the number of lines of text, the text distances settings, the line spacing, and then the width and height of the frame. I should mention here that if I were making this script for general consumption I should have done the **`getUnit()`**, **`setUnit()`**, **`setUnit()`** process that I showed in **bcaption.py**. There is one last error check - making sure that text is not overflowing in this frame. If that's the case, there is no solution. The entries between the parentheses in **`textOverflows()`** are first the frame name, and secondly the 1 indicates we only want to know about this frame, not those it links to. If text is overflowing, this command returns 1. It would probably be better to have written this as

```
   if (scribus.textOverflows(textbox, 1) == 1):
```

but either works. A little math follows, where we take the height of the frame, subtract the quotient of linespacing times the number of lines divided by 2. The result is then plugged into setting a new top distance with **`setTextDistances()`**.

But then look at the next two lines. I'm moving the frame 4 points to the left, then moving 4 points to the right. This shows the source of one of the uncertainties I have about **`setRedraw(True)`**. Notice that in this script I have already used **`setRedraw(False)`**. It turns out that if I don't do this little moving wiggle, when the script is done, you don't see the text brought down in the frame. You can see that the vertical distance is set as it should, but the text stays at the top of the frame. One work around is to save the file, then open it again. I also found that if I moved the frame a little, then the text moved to where it was supposed to be, so it's some kind of display issue. Later, I found out that if I did this little movement in the script, it worked the same way.

SETTOPDISTANCE.PY

One of the reasons I keep old scripts even when the original reason for their existence disappears is that you never know when you might recycle them in some form. Here is a mini version of this script that simply sets a top distance.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*

"""
settopdistance.py
USAGE
Select a text frame, run script.
"""
import scribus
textbox = scribus.getSelectedObject()
units = scribus.getUnit()
scribus.setUnit(scribus.UNIT_POINTS)
scribus.setRedraw(False)
distances = scribus.getTextDistances(textbox)
linespace = scribus.getLineSpacing(textbox)
scribus.setTextDistances(distances[0],distances[1],5.00, distances[3],
    textbox)
scribus.moveObject(-4,0,textbox)
scribus.moveObject(4,0,textbox)
scribus.setUnit(units)
scribus.setRedraw(True)
```

This is a script I shared on the mail list as a work around for the situation where someone wanted to automatically set the Line Offset in text frames to Linespacing mode. It turns out that there is no setting in Preferences for this, and no Scripter command for it either. What this script does is use the top distance as a proxy. I found that a 5 points top distance was indistinguishable from the spacing you see with Line Offset set to Linespacing, at least for a 12-point font.

This is a very basic script, and as you can see contains no error-checking. I mainly was presenting it so that whoever might use it could modify as desired. Sometimes with very simple scripts you can get by without the added lines.

EN+EMDASH.PY

Here is another script which came about as a modification of another script, in this case **Autoquote.py**. I'm presenting this one first because it is shorter, and hopefully simpler to understand. Both of these scripts have at their core the idea of analyzing the content of a text frame, character by character, with some transformation in mind. In Autoquote.py, the idea was to transform typewriter quotes such as we have on our keyboards to typographic quotes. This was a question posed on the mailing list, regarding whether this could be done in some automated fashion. Most word processor programs automatically do this as you type, but not so in Scribus, where you might be typing directly into a text frame, but often you are loading a text file into the frame.

en+emdash.py has as its task transforming typewriter hyphens into the typographic endashes and emdashes. The idea is that a single hyphen would stay as a hyphen, but two in a row would be changed to an endash, and three in a row would become an emdash.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*#
File: en+emdash.py -convert hyphens to en and em dashes

"""
USAGE
You must have a document open, and a text frame selected.
There are no dialogs. The presumption is that you have encoded a single
hyphen to mean a single hyphen, two hyphens to mean an en-dash, and three
hyphens to mean an em-dash.
"""
import scribus

if scribus.haveDoc():
    c=0
else:
    scribus.messageBox('Usage Error', 'You need a Document open', icon=0,
      button1=1)
    sys.exit(2)
if scribus.selectionCount() == 0:
    scribus.messageBox('Scribus -Usage Error', "There is no object
           selected.\nPlease select a text frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
if scribus.selectionCount() > 1:
    scribus.messageBox('Scribus - Usage Error', "You have more than one
        object selected.\nPlease select one text frame and try again.",
        scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
textbox = scribus.getSelectedObject()
pageitems = scribus.getPageItems()
boxcount = 1
for item in pageitems:
    if (item[0] == textbox):
        if (item[1] != 4):
            scribus.messageBox('Scribus - Usage Error', "This is not a
               textframe. Try again.", scribus.ICON_WARNING,
               scribus.BUTTON_OK)
            sys.exit(2)
contents = scribus.getTextLength(textbox)
ndash = u"\u2013"
mdash = u"\u2014"
```

```
  prevchar = ''
  while 1:
      if ((c == contents) or (c > contents)): break
      if ((c + 1) > contents -1):
        nextchar = ' '
      else:
        scribus.selectText(c+1, 1, textbox)
        nextchar = scribus.getText(textbox)
      scribus.selectText(c, 1, textbox)
      char = scribus.getText(textbox)
      if (len(char) != 1):
        c += 1
        continue
      if (prevchar == chr(45)):
        if (ord(char) == 45):
          scribus.deleteText(textbox)
          char = ndash
          c -= 1
        else:
          scribus.deleteText(textbox)
          scribus.insertText(chr(45) + char, c, textbox)
          c += 1
      elif (prevchar == ndash):
        if (ord(char) == 45):
          scribus.deleteText(textbox)
          scribus.insertText(mdash, c, textbox)
          char = mdash
        else:
          scribus.deleteText(textbox)
          scribus.insertText(ndash, c, textbox)
          c += 1
          scribus.insertText(char, c, textbox)

      else:
        if (ord(char) == 45):
          scribus.deleteText(textbox)
          c -= 1
        else:
          scribus.deleteText(textbox)
          scribus.insertText(char, c, textbox)
      c += 1
      prevchar = char
      contents = scribus.getTextLength(textbox)
  scribus.setRedraw(1)
  scribus.docChanged(1)
  scribus.messageBox("Finished", "That should do it!",
    icon=scribus.ICON_NONE, button1=scribus.BUTTON_OK)
```

We start out checking for an open document, and then whether some object is selected. Just as we did with **centervert.py**, we then check to make sure this is a text frame. Once these are satisfied, we check the length of the text in the frame. This is because we're going to use this as an indexing mechanism for the characters. We then define our variables for the typographic endash and emdash. In case you're wondering, this notation is how you specify unicode characters in Python. Incidentally, if you wanted to enter an endash on the main canvas you can use its Unicode value. Press Ctrl+Shift+U, and then enter 2013.

Our while loop, specified with **`while 1:`**, is going to keep running until we break it internally, which comes in the next line when our indexing goes beyond the length of our text, where c is our indexing variable. In Python, the first character of some variable **mytext** could be specified by **`mytext[0]`**. Since we have specified the count of the characters as contents, the last character is going to be mytext[contents -1]. I chose to work with 3 characters, the current character, called char, the previous one, called prevchar, and the following character, nextchar.

Look at these two lines:

```
  scribus.selectText(c, 1, textbox)
  char = scribus.getText(textbox)
```

This is what you must do to pluck one or more characters from somewhere in the middle of a text frame. You first select the character(s), then you **`getText()`**, a command which gets whatever has been selected. Later on, you can see that when we get ready to substitute a character, we simply **`deleteText()`**, which deletes whatever is selected. The structured variable naming is then keeping track of what's selected.

Overall, the script works in the following way. As we march through the text frame, using our indexing method, we are looking to see if our char character is a hyphen (in Python, **`chr(45)`**). If it is, then we want to see what the previous character is (**prevchar**). If that is also a hyphen, we delete both and substitute an ndash. The next test if we have a hyphen is whether **prevchar** is an endash. If that is the case, it's the same as saying that there were three hyphens in a row, and we then delete the ndash and the following hyphen and substitute an emdash.

When these substitutions occur, we have effectively shortened the length of the text in the text frame, and so that we don't skip any, we have adjust our indexing with **`c -= 1`** whenever we substitute ndashes and emdashes for hyphens. This also explains why we need to refresh our **`getTextLength()`** at the end of this while loop.

Here is a script where I have made use of the **`docChanged()`** command. The theoretical reason for this is so that Scribus recognizes that you have modified the document, and if you might try to exit it or shut down Scribus, it will give you a warning about saving the document. The last command, the **`messageBox()`**, is just there to show that the script is done and completed successfully.

As I look at this script and analyze it, I can see that I actually never made use of the variable **nextchar**. This is a holdover from **Autoquote.py**, but harms nothing here. Another thing about this script is that there is a certain kind of "error" it doesn't handle, which is when you have four or more hyphens in a row. One of the reasons I didn't work on that is that it only brings up the question of what should the script do in that case? Another reason is that this is supposed to be working on a conscientiously designed body of text, where someone has put in double or triple hyphens expressly for this purpose. Here is what some experimentation shows:

4 dashes = emdash then hyphen
5 dashes = emdash then endash
6 dashes = emdash then emdash
7 dashes = emdash then emdash then hyphen

So extrapolating from this, you will have as many emdashes in a row as possible, then either nothing or a hyphen or an endash. You won't ever see an endash then a hyphen because that would be made into an emdash.

AUTOQUOTE.PY

This is the mother or perhaps older sister of **en+emdash.py**. I'm not going to go through this one as carefully as before, and in fact I have truncated what you see here from the original to shorten the list of languages which are indicated. You can find **Autoquote.py** in Scribus if you want all these details.

As I said in the previous chapter, this came about when someone was searching for an automated way to convert typewriter quotes, both single and double, to typographic quotes (also called "curly quotes"). Once I had the logic worked out, I created a wiki page about it, and was then challenged to make it usable for other languages. This led to a LOT of research, since different languages use different typographic characters. The French and Russians, and some Germans use guillemets (« and »), and there are differences in how they are used also. In the end there are 23 possible language choices.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*#
File: quotes.py -changes typewriter quotes to typographic quotes
"""
USAGE
You must have a document open, and a text frame selected.
There will be a valueDialog asking for your language for the quotes,
the default is 'en', but change the default to suit your needs.
Detected errors shut down the script with an appropriate message.
"""
import scribus

if scribus.haveDoc() > 0:
    c=0
    lang = scribus.valueDialog("Choose by language or country", 'Language:
      af, be, ch, cs, de, de-g, en, es, et, fi, fr,\n hu, is, lt, mk, nl,
      pl, ru, se, sk, sl, sq and uk\n are current choices','en')
    if (lang == 'en'):
      lead_double = u"\u201c"
      follow_double = u"\u201d"
      lead_single = u"\u2018"
      follow_single = u"\u2019"
    elif (lang == 'de'):
      lead_double = u"\u201e"
      follow_double = u"\u201c"
      lead_single = u"\u2019"
      follow_single = u"\u201a"
    elif (lang == 'de-g'): # German -inverted guillemets for quotes
      lead_double = u"\u00bb"
      follow_double = u"\u00ab"
      lead_single = u"\u2019"
      follow_single = u"\u201a"
    elif (lang == 'fr'):
      lead_double = u"\u00ab"
      follow_double = u"\u00bb"
      lead_single = u"\u2018"
      follow_single = u"\u2019" # am hoping this will cover contractions
        like je t'aime
    else:
      scribus.messageBox('Language Error', 'You need to choose an available
        language', scribus.ICON_WARNING, scribus.BUTTON_OK)
      sys.exit(2)
```

```
else:
    scribus.messageBox('Usage Error', 'You need a Document open',
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
if scribus.selectionCount() == 0:
    scribus.messageBox('Scribus -Usage Error', "There is no object
           selected.\nPlease select a text frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
if scribus.selectionCount() > 1:
    scribus.messageBox('Scribus -Usage Error', "You have more than one
         object selected.\nPlease select one text frame and try again.",
         scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
textbox = scribus.getSelectedObject()
pageitems = scribus.getPageItems()
boxcount = 1
for item in pageitems:
    if (item[0] == textbox):
      if (item[1] != 4):
        scribus.messageBox('Scribus -Usage Error', "This is not a
          textframe. Try again.",scribus.ICON_WARNING, scribus.BUTTON_OK)
      sys.exit(2)
contents = scribus.getTextLength(textbox)
while c <= (contents -1):
    if ((c + 1) > contents -1):
      nextchar = ' '
    else:
      scribus.selectText(c+1, 1, textbox)
      nextchar = scribus.getText(textbox)
    scribus.selectText(c, 1, textbox)
    char = scribus.getText(textbox)
    if (len(char) != 1):
      c += 1
      continue
    if ((ord(char) == 34) and (c == 0)):
      scribus.deleteText(textbox)
      scribus.insertText(lead_double, c, textbox)
    elif (ord(char) == 34):
      if ((prevchar == '.') or (prevchar == ',') or (prevchar == '?') or
          (prevchar == '!') or (prevchar == '…')):
        scribus.deleteText(textbox)
        scribus.insertText(follow_double, c, textbox)
      elif ((ord(prevchar) == 39) and ((nextchar != ' ') and (nextchar !=
          ',') and (nextchar != '.'))):
        scribus.deleteText(textbox)
        scribus.insertText(lead_double, c, textbox)
      elif ((nextchar == '.') or (nextchar == ',')):
        scribus.deleteText(textbox)
        scribus.insertText(follow_double, c, textbox)
      elif ((prevchar == ' ') or ((nextchar != ' ') and (ord(nextchar) !=
          39))):
        scribus.deleteText(textbox)
        scribus.insertText(lead_double, c, textbox)
      else:
        scribus.deleteText(textbox)
        scribus.insertText(follow_double, c, textbox)
      if ((ord(char) == 39) and (c == 0)):
        scribus.deleteText(textbox)
        scribus.insertText(lead_single, c, textbox)
      elif (ord(char) == 39):
        if ((prevchar == '.') or (prevchar == ',') or (prevchar == '?') or
            (prevchar == '!')):
          scribus.deleteText(textbox)
          scribus.insertText(follow_single, c, textbox)
        elif ((ord(prevchar) == 34) and ((nextchar != ' ') and (nextchar !=
            ',') and (nextchar != '.'))):
          scribus.deleteText(textbox)
          scribus.insertText(lead_single, c, textbox)
        elif ((prevchar != ' ') and (ord(prevchar) != 34) and (nextchar !=
            ' ')):
          scribus.deleteText(textbox)
          scribus.insertText(follow_single, c, textbox)
        elif ((prevchar == ' ') or ((nextchar != ' ') and (ord(nextchar) !=
            34))):
          scribus.deleteText(textbox)
          scribus.insertText(lead_single, c, textbox)

        else:
          scribus.deleteText(textbox)
          scribus.insertText(follow_single, c, textbox)
      c += 1
      prevchar = char
  scribus.setRedraw(1)
  scribus.docChanged(1)
  endmessage = 'Successfully ran script\n Last character read was '+str(char)
    # Change this message to your liking
  scribus.messageBox("Finished", endmessage, icon=scribus.ICON_NONE,
      button1=scribus.BUTTON_OK)
```

In retrospect I can't recall what led me to the indexing method of analyzing the contents of the text frame. I do know that one of the concerns was not to disrupt any control characters, things like newline characters, or various control characters used to change font styles. So what I'm doing here is plucking out characters to inspect them, then delete/change them or just moving on.

After that, then there was the major question of how to decide when to insert a left typographic quote and when to insert a right quote. Some obvious ones seemed to be when the quote was the first character of a paragraph it should be left and if it's the last, a right. There also comes up word boundaries as defined by a space before and a space after. The end of a sentence could be detected by punctuation as a sign for a right quote. I also tried to keep track of whether the last quote was a left or a right.

In the end, all I can say is that there was a lot of trial and error involved with this in trying to think about the use-case situations. One kind of issue I never figured out was with something like a sentence beginning with 'Twas. It should be a right quote, but where is the logic for that? Probably only a very rare situation. Still, I was left with a little anxiety, which is indicated by the last two commands, where I use a message to show what the last character analyzed was, so the user could be sure he made it all the way through the text. After all, it's easy to imagine that a text frame might have a quotation mark as its last character.

Something the script made me aware of is that word processors are very good but imperfect at assigning typographic quotes, and in some situations my script outperforms them.

REPLACETEXT.PY

Here is another variant of this text changing scheme, but on a larger scale. Someone asked me if there was a way to scramble the text in a document. The reason he wanted to do this, was that he wanted to use the document as an exampleof its layout, but the actual text content was information he didn't want to share. One answer might have been to delete the text and substitute Lorem Ipsum or the like, but this could be a lot of work.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*#
File: replacetext.py
"""
USAGE
You must have a document open.
WARNING: this script irreversibly scrambles your text on all pages.
You would be wise to work on a copy of the original to avoid
accidentally saving this scrambled version only to lose the original.
"""
import scribus
import random

if scribus.haveDoc():
    c=0
else:
    scribus.messageBox('Usage Error', 'You need a Document open',
           scribus.ICON_NONE, scribus.BUTTON_OK)
    sys.exit(2)
scribus.messagebarText("Getting ready to process Page 1")
scribus.redrawAll()
warnresult = scribus.valueDialog('Warning!', 'This script is going to
  irreveribly alter the textin your document.\nChange this default value to
  abort', 'Ok!')
if (warnresult != 'Ok!'):
    sys.exit(2)
page = 1
pagenum = scribus.pageCount()
while (page <= pagenum):
    scribus.gotoPage(page)
    scribus.messagebarText("Processing Page "+str(page))
    scribus.redrawAll()
    pageitems = scribus.getPageItems()
    for item in pageitems:
      if (item[1] == 4):
        c=0
        textbox = item[0]
        scribus.selectObject(textbox)
        contents = scribus.getTextLength(textbox)
        while 1:
          if ((c == contents) or (c > contents)): break
          if ((c + 1) > contents -1):
            nextchar = ' '
          else:
            scribus.selectText(c+1, 1, textbox)
            nextchar = scribus.getText(textbox)
          scribus.selectText(c, 1, textbox)
          char = scribus.getText(textbox)
          if (len(char) != 1):
             c += 1
             continue
           alpha = random.randint(1,26)
           letter = chr(alpha + 96)
           LETTER = chr(alpha + 64)
           if ((ord(char)>96)and(ord(char)<123)):
             scribus.deleteText(textbox)
             scribus.insertText(letter, c, textbox)
           if ((ord(char)>64)and(ord(char)<91)):
             scribus.deleteText(textbox)
             scribus.insertText(LETTER, c, textbox)
           c += 1
           contents = scribus.getTextLength(textbox)
    page += 1
scribus.setRedraw(1)
scribus.docChanged(1)
scribus.messageBox("Finished", "That should do it!",
               icon=scribus.ICON_NONE, button1=scribus.BUTTON_OK)
```

We start out with page 1, and get a count of the pages using **`pageCount()`**. One of the things I have done here is to add a **`messagebarText()`** command. It takes some time to process a mulitipage document, so this message in the lower left corner of the window gives an indicator of where the script is working. You've seen this method before of getting all the objects on a page, then sifting out which ones are text frames. So next we're marching through the characters in the text frame. Look at this clause:

```
if (len(char) != 1):
c += 1
continue
```

All the printable characters will have a length of 1, so if a char fails this test, then it must be some kind of control character, and we don't want to scramble those, so we just move on to the next. Now look at this:

```
alpha = random.randint(1,26)
letter = chr(alpha + 96)
LETTER = chr(alpha + 64)
```

Here is the generator of the randomness, where I pick a number between 1 and 26 (as in letters of the alphabet), then create two possible replacements, one a lower case letter, the other an upper case. I wanted to retain the appearance of normal text, with capitalization where it occurred in the original, and thus it might appear like some foreign language.

The next step was to see if the original char was an upper case or lower case letter, then make the appropriate switch. I elected not to change numbers, for one reason considering there might be lists in the text, and there was no reason to disturb these. If someone had some sensitive numbers they did not want preserved, then numbers might also need scrambling. Once we have marched through the document we then end with a succinct message.

As I review this script and then try it out on a multipage document, something occurs to me. I don't know if I might be recursively going over this text. As we will see in the next script, ExtractText.py, when you pull out the text from a series of linked frames, you will find that accessing the first frame in the link gets all of the text for the linked frames. In a sense, this isn't a huge issue here, but might make a script run longer than necessary, since it may be rescrambling the same text over and over.

*A note about Windows and packages like **random***

Scribus versions for Windows which are downloadable from Sourceforge have Python 2.7 included with them, so you don't have to have Python on your system to use Scripter. On the other hand, this included Python has only a few additional packages. Even if you have Python on your system, you will notice that Scribus in its basic state will not use it. The answer comes from finding inside the Scribus folder in C:\Program Files the directory named python\. If you change its name to anything else, then Scribus will look to your system for Python – make sure that you have Python 2.7 (as of this writing).

EXTRACTTEXT.PY

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
"""
SYNOPSIS
This script takes the current document and extracts all the text from text
  frames,
and also gets the pathnames to all images. This is then saved to a file
  named
by the user.
USAGE
Start the script. A file dialog appears for the name of the file
to save to. The above information is saved to the file.
"""

import scribus

def exportText(textfile):
    page = 1
    pagenum = scribus.pageCount()
    T = []
    content = []
    while (page <= pagenum):
      scribus.gotoPage(page)
      d = scribus.getPageItems()
      strpage = str(page)
      T.append('Page '+ strpage + '\n\n')
      for item in d:
        if (item[1] == 4):
          contents = scribus.getAllText(item[0])
          if (contents in content):
            contents = 'Duplication, perhaps linked-to frame')
          T.append(item[0]+': '+ contents + '\n\n')
          content.append(contents)
        elif (item[1] == 2):
          imgname = scribus.getImageFile(item[0])
          T.append(item[0]+': ' + imgname + '\n')
      page += 1
      T.append('\n')
    output_file = open(textfile,'w')
    output_file.writelines(T)
    output_file.close()
    endmessage = textfile + ' was created'scribus.messageBox("Finished",
        endmessage, scribus.ICON_NONE, scribus.BUTTON_OK)

if scribus.haveDoc():
    textfile = scribus.fileDialog('Enter name of file to save to', \
          filter='Text Files (*.txt);;All Files (*)')
    try:
      if textfile == '':
        raise Exception
      exportText(textfile)

    except Exception, e:
      print e
```

```
else:
    scribus.messageBox('Export Error', 'You need a Document open, and a
        frame selected.', scribus.ICON_NONE, scribus.BUTTON_OK)
```

The idea behind this script was that you may have been working on a document, and separate from the document itself you just want to save its content. You might want to run the text by someone for review, maybe even someone that doesn't have or use Scribus. Initially, I only wrote this to extract the text and save that to a file, but later decided to also get the filenames of the images and save that also.

At the top of the script is a function called exportText, so the action begins down lower, at if **`scribus.haveDoc()`**. Here is where we ask the user for a filename to save the results, using a **`fileDialog()`**, so that we have not only the filename, but a path. After this, the real work begins by accessing **exportText**.

Just as we have done before, we get a **`pageCount()`**, then **`getPageItems()`** on the page and go through them. This time we not only are interested in text frames, but also image frames. You might wonder what the "order" of frames is. In this example, it will be the order in which they were created for the page the script is working on. As we go through the text and image frames, we are creating a list, T, which starts with the text 'Page ' and the number of the current page, after which is appended the text from text frames, and the image file names. In regard to the text frames, though, look at this:

```
contents = scribus.getAllText(item[0])
if (contents in content):
    contents = 'Duplication, perhaps linked-to frame'
```

Before we append our text frame content to the list T we look to see if it is a duplicate of something already there. I found out early in the development of this script that if you had a series of linked frames, each time to accessed one of the frames in that series, you got all the text from the linkage. So to keep the file from being repetitive, I substituted this message, 'Duplication, perhaps linked-to frame.' The reason I say perhaps is that it is possible someone might have two or more frames with identical content, even though not linked. There might be some more sophisticated way to look for linked frames, but it wasn't clear this would add any great utility to the script.

Finally, when we've finished, we send an appropriate message. This is a script that is included with Scribus, though you won't find it in the menu with **Scripter > Scribus Scripts**. Depending on your installation, it is located in a directory called samples.

SCRIBALBUM.PY

This in its original state was one of the first major scripts I wrote. As I look back in my old files, I see that it began February 13, 2005 (I guess that wasn't a Friday). What I wanted to be able to do was to simply point my computer at a directory of images, and have the script then load all the images into a document or album. I made versions for putting 4 on a page, 6 on a page, and eventually allowed the user to choose which. For my own purposes, I would tend to use this after I had been on some trip, accumulated a number of images to be able to review without the use of a computer, and also put some notes beside certain pictures when desired. Sometimes there are some particulars about the image that you soon or eventually forget that you wish you hadn't. Four pictures on US Letter paper leaves a lot more space for these notes, but sometimes the images themselves are the whole story more or less.

I also made versions for A4 paper, but here I'll show the US letter version. The main difference has to do with the placement of the image frames, which is hardcoded into the scripts.

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

# File: scribalbum.py
# This version for US Letter paper
# Puts 4 or 6 pictures per page
# This version uses fileDialog to choose the directory
# Value dialog to choose file type(s)
# Filters out files ending with .jpg, .png, .tif, .gif, .pdf, and
# uppercase equivalents.
# Makes a new document, and will not fault if you already have one.


import scribus
import os


filetype = []
dicttype = {'j':'.jpg','p':'.png','t':'.tif','g':'.gif','P':'.pdf'}
Dicttype = {'j':'.JPG','p':'.PNG','t':'.TIF','g':'.GIF','P':'.PDF'}
nrimages = '0'
while ((nrimages != '4') and (nrimages != '6')):


nrimages = scribus.valueDialog('Pictures','-US Letter Paper -\n Images per
  Page?\nOnly validentries are 4 or 6','4')
nrimages = int(nrimages)

imagedir = scribus.fileDialog('Select Image Directory','Directories',
  isdir=True)
imagetype = scribus.valueDialog('Image Types','Enter the Image Types, where
  \n
j=jpg,p=png,t=tif,g=gif,P=pdf\n "jptgP" selects all','jptgP')
for t in imagetype[0:]:


filetype.append(dicttype[t])

filetype.append(Dicttype[t])
d = os.listdir(imagedir)
```

```
D = []
for file in d:
    for format in filetype:
      if file.endswith(format):
      D.append(file)
D.sort()
# When 4 pics per page, coords are: (15, 42),(310, 187), (15, 388), (310,
  533)
# When 6 pics per page: (15, 42),(310, 42), (15, 290), (310, 290),(15,533),
  (310,533)
if nrimages == 4:
xpos = [15, 310, 15, 310]
ypos = [42, 187, 388, 533]
if nrimages == 6:
    xpos = [15, 310, 15, 310, 15, 310]
    ypos = [42, 42, 290, 290, 533, 533]
# This proportion is right for photographs (at least for my digital Nikon)
pwidth = 288
pheight = 193.33
imagecount = 0
if len(D) > 0:
    if scribus.newDocument(scribus.PAPER_LETTER, (10,10,20,20),
           scribus.PORTRAIT, 1, scribus.UNIT_POINTS, scribus.NOFACINGPAGES,
           scribus.FIRSTPAGERIGHT,1):
      while imagecount < len(D):
        if imagecount > 0:
          scribus.newPage(-1)
          framecount = 0
# L is the frame at the top of each page showing the directory name
      L = scribus.createText(15, 20, 300, 20)
      scribus.setText("Dir: " + imagedir, L)
      scribus.setTextAlignment(scribus.ALIGN_LEFT, L)
      scribus.setFont("DejaVu Sans Book", L)
      scribus.setFontSize(10, L)
# Here is where we're loading images into the page, four or six at a time,
  then go back up for anewPage
      for x,y in zip(xpos,ypos):
        if imagecount < len(D):
          f = scribus.createImage(x, y, pwidth, pheight)
          scribus.loadImage(imagedir + '/' + D[imagecount], f)
          scribus.setScaleImageToFrame(scaletoframe=1, proportional=1,
               name=f)
          lenfilename = len(D[imagecount])
          Lpiclen = int(5.3 * lenfilename)
# Lpic is the label for each picture, with position and length adjusted
# according to the text length, so if you change the font or its size,
# you may need to adjust this only approximate calculation.
          Lpic = scribus.createText(x, y + 195, Lpiclen, 15)
          scribus.setText(D[imagecount], Lpic)
          scribus.setTextAlignment(scribus.ALIGN_RIGHT, Lpic)
          scribus.setFont("DejaVu Sans Book", Lpic)
          scribus.setFontSize(8, Lpic)
          scribus.setFillColor("White", Lpic)
          imagecount += 1
scribus.setRedraw(1)
scribus.redrawAll()

else:
result = scribus.messageBox ('Not Found','No Images found with\n this
        search selection',scribus.BUTTON_OK)
```

When I was first using this script on a directory with a large number of images, it was quite a strain on resources when you were creating a 20 or 30 page or more album. Since then Scribus uses image caching, which helps, but of course I have a computer with more memory as well.

At the beginning of the script I create a couple of Python dictionaries, **`dicttype{}`** and **`Dicttype{}`**. Later on, these allow for using the initial letter to choose a file extension, for example j for .jpg or .JPG. This minimizes the amount of typing in the **`valueDialog()`**. The default is all image types, including PDFs (if you import a PDF into an image frame, you will only get the first page).

Now look at this sequence:

```
    nrimages = '0'
    while ((nrimages != '4') and (nrimages != '6')):
        nrimages = scribus.valueDialog('Pictures','-US Letter Paper -\n Images
          per Page?\nOnly valid  entries are 4 or 6','4')
    nrimages = int(nrimages)
```

After initializing nrimages with '0', I then use a **`while`** loop to only allow the user to enter 4 or 6 for the number of images per page. If something else is entered, the **`valueDialog()`** appears again with its default value. Another lesson here is that the input from a **`valueDialog()`** is always a string, so you must convert if you're going to use it as an integer.

The next task is for the user to choose a directory, presumably containing images, with a **`fileDialog()`** – notice that the last element in the command, **`isdir=True`** makes sure that only directories are shown in the dialog. After the user indicates any particular filetypes to include, a list is created which contains only those extensions.

After that, we can now open the directory, and as we read it, we only append to our image list D a filename if it fits our selected extension list. The script then sorts this list alphabetically – this is a line you might want to comment out (with a # at the beginning of it).

Depending on which layout has been chosen, two lists, **`xpos[]`** and **`ypos[]`** are created for positioning the images. With some experimentation, I decided on these hardcoded values for **pwidth** and **pheight**, which are used to create the image frames. Occasionally I may have an image I have taken sideways, but adjusting those frames is something to be done after the script runs. Finally, we're ready to actually make the document with the **`newDoc()`** command, with the first step on each page being to create a text frame **L** in the upper left corner which will show the directory the images came from.

Until I reviewed this script, I had forgotten about the zip command. This explains the odd look that the xpos and ypos lists. zip presents a series of tuples, **`xpos[0]`**, **`ypos[0]`**, then **`xpos[1]`**, **`ypos[1]`** and so on, and thus we march down the page with our 4 or 6 images per page. After each image is placed into its frame, we then create a text frame underneath showing the image filename. After each image, we check with our while command whether we have exhausted the image file list. If we have exhausted what **zip** can do for one page, we create a new page with **`newPage(-1)`**, and begin the **zip** process again.

At the very bottom of the script you see how we have allowed for the user to have chosen a directory containing no images. It may be difficult to see (you have to disregard all the comment lines), but this **else** clause lines up with if **`len(D) >0`**, so it's the alternative when the list D has no elements.

SHIFTOBJECTS.PY

Here is another mail list-driven set of scripts. The problem as posed was that a document had been created with left and right pages. For many situations this means that content on a right page is shifted to the right making a smaller outer margin, and the reverse for left-sided page. The questioner had then inserted a new page somewhere in the middle of the document, and thus the content after the new page was shifted in the wrong direction. Was there any help for this issue? Scripter, of course.

The first script, **shiftobjects.py**, would shift the content on the current page. The user was asked how much to shift in the current page units, and then which direction to shift, and then all of the items on the page were shifted accordingly. This meant of course that the user had to run this script page by page for all which needed shifting. Thus, **shiftobj2.py**.

SHIFTOBJ2.PY

The second version of this process allowed the user to enter a series of page numbers to be shifted, then asked about the amount of shift and direction.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
"""
shift_objects.py
USAGE
Select a page where you need to shift all objects, run script.
The first dialog asks how much to shift, using a distance of 40 points
as the default, but modified to your units.
The second dialog asks whether to shift objects Left (the default), or
  Right.
Simply enter anything other than Left to shift right.
This version allows for a series of pages to be shifted at once.
"""

try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
if not scribus.haveDoc():
    scribus.messageBox('Scribus -Script Error', "No document open",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(1)
pgerr = 0
pgcount = scribus.pageCount()
page = scribus.currentPage()
scribus.setRedraw(False)
units = scribus.getUnit()
if units == 0:
    shift = 40
elif units == 1:
    shift = 14.111 # millimeters = points/2.8346
elif units == 2:
    shift = 0.5556 # inches = points/72.0
shiftamount = scribus.valueDialog('Amount of shift?','Enter the amount to
      shift',str(shift))
```

```
shiftamount = float(shiftamount)
shiftdir = scribus.valueDialog('Direction to shift?','Which direction to
      shift...\nAny change from default shifts to Right','Left')
if ((shiftdir == 'Left') or (shiftdir == 'left')):
    shiftamount = -shiftamount
pages = scribus.valueDialog('Pages to Alter','List pages to shift,
           separated by white space\n(no commas)',str(page))
pagelist = pages.split()
for pg in pagelist:
    if ((int(pg) < 1) or (int(pg) > pgcount)):
      pgerr = pgerr + 1
    else:
      scribus.gotoPage(int(pg))
      pageitems = scribus.getAllObjects()
      for item in pageitems:
      scribus.moveObject(shiftamount, 0, item)
scribus.setRedraw(True)
if pgerr == 1:
    scribus.messageBox('OOPS!',str(pgerr)+' of the pages you entered was\n
         outside the range of the document\n and therefore ignored.',
         scribus.ICON_WARNING, scribus.BUTTON_OK)
if pgerr > 1:
    scribus.messageBox('OOPS!',str(pgerr)+' of the pages you entered were\n
         outside the range of the document\n and therefore ignored.',
         scribus.ICON_WARNING, scribus.BUTTON_OK)
```

Not surprisingly, we start out with our error-checking for Scribus and also that a document is open. Next we get a page count, and then we find out which page units the document is using. We're a bit shallow here, since we are only looking for points, millimeters, or inches, but that will likely cover most users. We do this so we can set a default value for shift, the equivalent of 40 points. I could have done the shift to points, then back again, as we saw before, but chances are if the user is used to millimeters, he'll know how much to shift in those units, not in points.

We then ask how much the user wants to shift the objects, and then which direction, left or right. Notice how I have simplified the logic. I might have tried to cover all the possible entries, l, r, L, R, Left, Right, left, right, and so on. Instead, I make the default 'Left', and in the next line I only look for Left or left, and if neither of these is the entry, then Right is assumed. All this does is to decide whether to negate the amount of shift. Realistically, I should have added language to the **`valueDialog()`** to indicate an absolute value should be entered, especially since the amount was asked for before the direction. Alternatively and perhaps better would have been to change **`shiftamount = -shiftamount`** to **`shiftamount = (abs(shiftamount))`**.

Next we ask for a series of pages to alter, and we don't want commas, just whitespace between the numbers, since this is the default way that the **`split()`** string function works. Now let's imagine that, in his haste, the user has entered some values which don't exist, maybe 0 or 21 in a twenty page document. We cover this by seeing if any entered value is outside the page range and create an error counter, pgerr, and skip over those. Look how small the actual action part of the script is:

```
scribus.gotoPage(int(pg))
pageitems = scribus.getAllObjects()
for item in pageitems:
    scribus.moveObject(shiftamount, 0, item)
```

That's it! The rest is all error-handling and user input. But we do also throw in some feedback at the end in case there were erroneous pages, and we even go the extra grammatical mile to cover one versus many in our feedback.

SHIFTOBJ3.PY

After this, what's left? How about some even better automation? Imagine that your user is working on a 100 page document, and somewhere about page 7 is where he inserted a new page. Now he will have to enter '8 10 12 14 16 18' and so on all the way to the end of the document. What can we do for this?

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
"""
shift_obj3.py
USAGE
Select a page where you need to shift all objects, run script.
The first dialog asks how much to shift, using a distance of 40 points
as the default, but modified to your units.
The second dialog asks whether to shift objects Left (the default), or
  Right.
Simply enter anything other than Left to shift right.
This version allows for a series of pages to be shifted at once.
Enter odd or even to shift every other page to the end.
"""
try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
         within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
         Script."
    sys.exit(1)
if not scribus.haveDoc():

    scribus.messageBox('Scribus -Script Error', "No document open",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(1)
pgerr = 0
pgcount = scribus.pageCount()
page = scribus.currentPage()
scribus.setRedraw(False)
units = scribus.getUnit()
if units == 0:
    shift = 40
elif units == 1:
    shift = 14.111 # millimeters = points/2.8346
elif units == 2:
    shift = 0.5556 # inches = points/72.0
shiftamount = scribus.valueDialog('Amount of shift?','Enter the amount to
         shift',str(shift))
shiftamount = float(shiftamount)
shiftdir = scribus.valueDialog('Direction to shift?','Which direction to
         shift...\nAny change from default shifts to Right','Left')
if ((shiftdir == 'Left') or (shiftdir == 'left')):
    shiftamount = -shiftamount
pages = scribus.valueDialog('Pages to Alter','List pages to shift,
           separated by white space\n(nocommas)\nEnter odd or even for every
           other page',str(page))
if (pages == 'odd') or (pages == 'even'):
    startendpages = scribus.valueDialog('Odd/Even','Where to start (and
           end)?\n(One or 2 page numbers, separated by white space)',
           str(page) + ' ' +str(pgcount))
    startend = startendpages.split()
    pgadd = int(startend[0])
    if (startend[1] != ""):
      pgcount = int(startend[1])
    pagelist = []
    while pgadd <= pgcount:
      pagelist.append(str(pgadd))
      pgadd += 2
else:
    pagelist = pages.split()
for pg in pagelist:
    if ((int(pg) < 1) or (int(pg) > pgcount)):
      pgerr += 1
    else:
      scribus.gotoPage(int(pg))
      pageitems = scribus.getAllObjects()
      for item in pageitems:
        scribus.moveObject(shiftamount, 0, item)
scribus.setRedraw(True)
if pgerr == 1:
    scribus.messageBox('OOPS!',str(pgerr)+' of the pages you entered was\n
        outside the range ofthe document\n and therefore ignored.',
        scribus.ICON_WARNING, scribus.BUTTON_OK)
if pgerr > 1:
    scribus.messageBox('OOPS!',str(pgerr)+' of the pages you entered were\n
        outside the range ofthe document\n and therefore ignored.',
        scribus.ICON_WARNING, scribus.BUTTON_OK)
```

Most of this should look familiar, that is, until we get to this:

```
  pages = scribus.valueDialog('Pages to Alter','List pages to shift,
    separated by white space\n(nocommas)\nEnter odd or even for every other
    page',str(page))
  if (pages == 'odd') or (pages == 'even'):
      startendpages = scribus.valueDialog('Odd/Even','Where to start (and
          end)?\n(One or 2 pagenumbers, separated by white space)',str(page)
          + ' ' +str(pgcount))
```

First of all, in addition to the option of listing page numbers, we can just enter 'odd' or 'even'. If we do that, then we get another **`valueDialog()`** asking where to start, with an option to put in where to end. After all, the user might have inserted a page, then somewhere later in the document inserted another, so everything after that second insertion will be Ok. Notice however, that we don't check for the error where a user might say 'odd', but then enter even numbers. At some point you have to draw the line on error-checking. Another reason to save your document just before you run any script.

So with the following lines we have to handle all these possibilities a series of numbers, odd or even, and if these latter two, one or maybe two more page numbers. We then wind our way through these issues to create our list of pages to shift, and finish with our message about any page errors.

SWAPIMAGE.PY

Here's a fun little script, that I can't say anyone including me has ever used in a purposeful way. The idea is that maybe you want to play with some layout containing images, and just want to see the difference in impact by interchanging a couple of images.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
"""
swapimage.py
USAGE
Select 2 image frames, no more, no less, both must be image frames. Can be
  on different pages.
Run the script, the images are swapped.
"""
try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
      within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
      Script."
    sys.exit(1)
if scribus.selectionCount() != 2:
    scribus.messageBox('Selection Count', "You must have 2 image frames
         selected", scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
frame1 = scribus.getSelectedObject(0)
frame2 = scribus.getSelectedObject(1)
ftype1 = scribus.getObjectType(frame1)
ftype2 = scribus.getObjectType(frame2)
if ((ftype1 != "ImageFrame") or (ftype2 != "ImageFrame")):
    scribus.messageBox('Object Type', "Both selected objects must be image
         frames", scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
scribus.setRedraw(False)
filename1 = scribus.getImageFile(frame1)
filename2 = scribus.getImageFile(frame2)
scribus.loadImage(filename2, frame1)
scribus.loadImage(filename1, frame2)
scribus.setRedraw(True)
```

The usage information suggests right off the bat that most of what is here is error-checking. Aside from checking about running this in Scribus, we make sure that two and only two objects are selected, AND that they're both image frames. Notice here I have a different style of checking frame type. The command **`getObjectType()`** will return the string '**ImageFrame**' if that's what it is, and note also there is no space, that's not a typo.

Once we clear the error hurdles, in four lines our work is done. If the user decides he doesn't like the switch, he just runs the script again.

Something I noticed on playing with this script is that it doesn't work so well in a situation where one or both pictures have an X-offset or Y-offset. A few more lines would be needed to get this information, and then transfer it.

INVISIBLEIMAGES.PY

Here is a script which again was a request from someone. What he was looking for was a quick way to hide the images in a document. You can do this manually on the main canvas, but only one by one. My approach here was to create a new layer in the document, then transfer all the images to that layer, and finally make the layer invisible.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# File: invisibleImages.py
"""
USAGE
You must have a document open.
This script moves all images to a new layer called "Ghostlayer",
the idea being that you can show/hide, print-export/not as desired.
"""
import scribus
if scribus.haveDoc():
    fantome = "Ghostlayer"
    scribus.createLayer(fantome)
    working = scribus.getActiveLayer()
    page = 1
    pagenum = scribus.pageCount()
    while (page <= pagenum):
      scribus.gotoPage(page)
      scribus.setActiveLayer(working) # maybe not necessary?
      pageitems = scribus.getPageItems()
      for item in pageitems:
      if (item[1] == 2):
      imagebox = item[0]
      scribus.selectObject(imagebox)
      scribus.copyObject(imagebox)
      scribus.setActiveLayer(fantome)
      scribus.pasteObject(imagebox)
      scribus.deleteObject(imagebox)
      scribus.setActiveLayer(working)
      page += 1
    scribus.setLayerPrintable(fantome, 0) # comment this out to do manually
        later
    scribus.setLayerVisible(fantome, 0) # comment this out to do manually
        later
    scribus.setRedraw(1)
    scribus.docChanged(1)
    scribus.messageBox("Finished", "That should do it!", scribus.ICON_NONE,
        scribus.BUTTON_OK)
else:
scribus.messageBox('Usage Error', 'You need a Document open',
scribus.ICON_NONE,scribus.BUTTON_OK)
sys.exit(2)
```

The first thing we do is to create our new layer, named **fantome**. Now, just as we've seen in several other scripts, we go through the document page by page, and on each page **`getPageItems()`**, and this time sift out all the image frames. For each frame we select it, copy it, then switch our active layer to **fantome** and paste there. Once this is done, we can safely delete the original. As I look through this script, I would have thought that I might have needed to switch the active layer back to working, but apparently not, since this does work as desired.

Once all the image frames have been copied and the originals deleted, then we not only make the fantome layer non-printable (which also means that those images would not be exported to PDF), but then make the content invisible.

In case any of your images has Text Flows Around Frame set, this still is in operation. If you want to make your images visible once again, just bring up the Layers dialog and check the appropriate boxes for the layer. This script also would open the possibility for easily making a document with more than one set of images. You would need to modify it by commenting out the line which deletes the original image frames, then load new images into one set or the other. Since the new layer is on top of the old one, making fantome visible only shows its images.

BAR CODES

POSTNET.PY

This was something I made strictly for my own use. Scribus of course has the Barcode Generator plugin, which works Ok, though personally I find the interface a little clunky. It does what it does and you cannot change features, like bar length or the font for the text/numbers if included. It generates a PostScript item that you place on the page. It's also not scriptable.

The first task for creating Postnet codes for the US Postal Service is research into the requirements, so that you make bars of adequate length and they are spaced properly. There are two lengths of bars, with patterns of these bars corresponding to various digits. In addition, at the end of the bars there is a "correction code", which is an added pattern of bars so that the sum of all the digits is divisible by 10.

```
#!/usr/bin/env python
# File: postnet.py
# this version 2006.03.07 adds the correction code:
# the sum of all the digits must be divisible by 10

import scribus
postcode = ['llsss', 'sssll', 'sslsl', 'sslls', 'slssl', 'slsls', 'sllss',
      'lsssl','lssls','lslss']
a = 1.44  #line width
b="Black" #line color
relx=47   #Start X (relx = 'relative x')
rely=323  #Start Y
correctnum = 0
if scribus.haveDoc():
    S = scribus.valueDialog('Postnet Barcode','Enter your Zip or Zip+4')
    scribus.setRedraw(1)
    scribus.setUnit(0)
    d = scribus.createLine(relx,rely,relx+9,rely,) #Begin long framing line
    scribus.setLineWidth(a, d)
    scribus.setLineColor(b, d)
    rely = rely + 3.4
    for x in S[0:]:
      if x.isdigit():
        xnum = int(x)
        correctnum += xnum # Building the sum of all the digits
        code = postcode[xnum]
        for y in code[0:]:
          if y == 'l':
            d = scribus.createLine(relx,rely,relx+9,rely,) #Long line
            scribus.setLineWidth(a, d)
            scribus.setLineColor(b, d)
            rely = rely + 3.4
          elif y == 's':
            d = scribus.createLine(relx,rely,relx+3.6,rely,) #short line
            scribus.setLineWidth(a, d)
            scribus.setLineColor(b, d)
            rely = rely + 3.4
    correctstr = str(correctnum) # Here the correction code is added
    correctdig = int(correctstr[-1])
    if correctdig != 0:
      correctdig = 10 -correctdig
    code = postcode[correctdig]
    for y in code[0:]:
      if y == 'l':
        d = scribus.createLine(relx,rely,relx+9,rely,) #Long line
        scribus.setLineWidth(a, d)
        scribus.setLineColor(b, d)
        rely = rely + 3.4
      elif y == 's':
        d = scribus.createLine(relx,rely,relx+3.6,rely,) #short line
        scribus.setLineWidth(a, d)
        scribus.setLineColor(b, d)
        rely = rely + 3.4


    d = scribus.createLine(relx,rely,relx+9,rely,) #Endingframing line
    scribus.setLineWidth(a, d)
    scribus.setLineColor(b, d)
    scribus.redrawAll()
```

At the beginning of the script you see I create a list, **`postnet[]`**. What this contains are the rules for the various digits. As you can see, there are 5 bars for each digit. postnet[0] or 'llsss' is for the digit 0, which means that there are 2 long lines followed by 3 short lines. We then set some variables for the width of the lines we'll make, the color, and the starting point for these lines, which is at **XPos**, **YPos** of 43, 323. correctnum is going to be our correction digit, which we initialize at 0.

As you can see, even though we use an if statement to check if there is a document, I don't have a corresponding else clause, so if there is no document, the script just quits, without any feedback. In the US, the basic postal code is a 5-digit Zipcode, which directs the mail to a particular Post Office branch. There is also the option of Zip+4, where 4 more digits are added which correspond to a particular address. This script will automatically handle both types. Zip+4 is usually written as a sequence of 5 digits, a hyphen, then the last 4 digits, even though the Postnet code contains only the digits. Postnet bar codes also begin and end with what's called a framing line, basically a single long line. Here we see the sequence for creating lines with Scripter:

```
  d = scribus.createLine(relx,rely,relx+9,rely,) #Beginning long framing line
  scribus.setLineWidth(a, d)
  scribus.setLineColor(b, d)
```

We first create the line's dimensions with a starting **XPos**, **YPos**, then the ending **XPos**, **YPos** as you see, the line is horizontal and has a length of 9 points. Once the line is created, then we can assign a width, and a line color.

After the framing line, we then proceed to pick apart **`postnet[]`** digit by digit, marching 3.4 points down the page after each line. In case you wonder why I'm creating horizontal lines in a vertical sequence, this is because of how I have used this script. I also have a template for No. 10 envelopes, which is placed and oriented so that I can print them with my printer, where I feed the left edge of the envelope first into the printer. This location and arrangement of the Postnet code corresponds to the place under the mailing address for this template.

So what about this correction code? If we think about it, what we're saying for the sum of all digits to be divisible by 10, the sum should end with a zero. To that end, we convert the correctnum to a string and check the last character, correctstr[1]. Our correction digit is then either 0 or 10-int(correctstr[-1]). After adding the code for that and the final framing line, we're done.

CODE39.PY

This was also a personal project, but part of something bigger. Being the zealous Scribus user that I still am, I wanted to recreate my own version of the pages of the patients' hospital charts. This was back in the time when our hospital still had paper charts for notes, even though an increasing amount of information was on the computer system. If I could create my own chart pages, customized to the particular patient, I could then use Scribus to make my notes as I made my hospital rounds each day. As a consequence, I had the most legible notes of any doctors making rounds, and needless to say, people were impressed. This also meant that I could make an additional copy of the page to take back to the office for the patient's office chart (when this was useful). Some doctors even came up with their own schemes for making their notes this way. With Scribus, I still had them beat by a mile, because I could also incorporate into my notes selected images from a patient's CT or MRI scans, and thus explain what I felt was important in the studies, and I received many thanks from the doctor and nursing staff for the teaching that came from this.

Breaking down this idea into what I needed to know, the overall layout of the page was simple, with a series of lines. Even though I was mostly using Scribus for the notes, sometimes I might add something hand-written, and other doctors could also add to the page to fill it. In the upper right corner of page was patient identification information, including their hospital number in legible form and also as a barcode. But what barcode? This turned out to be an easy question to answer, since I just went to Scribus and Barcode Generator, plugging in hospital numbers, and quickly found out that this was a Code 39 barcode.

This then led to research into what Code 39 is, easily found on the internet with precise descriptions. Code 39 is a sequence of thin and thick lines, separated by thin and thick spaces. Unlike Postnet, Code 39 can handle all the letters of the alphabet, all digits, and even some symbols.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# File: code39.py
# Now will encode all defined characters, plus the start/stop character
# Any other character causes the script to exit

import scribus
import sys
# first the codes for numbers 0-9
code39 =
['nnswwn', 'wnsnnw', 'nwsnnw', 'wwsnnn', 'nnswnw', 'wnswnn', 'nwswnn',
       'nnsnww', 'wnsnwn','nwsnwn']
# next add the codes for A-Z and the rest
code39.extend(['wnnsnw','nwnsnw','wwnsnn','nnwsnw','wnwsnn','nwwsnn',
       'nnnsww', 'wnnswn','nwnswn'])
code39.extend(['nnwswn','wnnnsw','nwnnsw','wwnnsn','nnwnsw','wnwnsn',
       'nwwnsn','nnnwsw','wnnwsn'])
code39.extend(['nwnwsn','nnwwsn','wsnnnw','nswnnw','wswnnn','nsnwnw',
       'wsnwnn','nswwnn'])
code39.extend(['nswnwn','nsnsnsnn','nnsnsnsn','nsnnww','wsnnwn','nsnsnnsn',
       'nsnnsnsn'])
startstop = 'nsnwwn'
narrow = 1.0 # narrow line width
wide = 2.2 # wide line width
b="Black" #line color
if scribus.haveDoc():
    S = scribus.valueDialog('Code 39','Enter Numbers or Uppercase Letters')
    height = scribus.valueDialog('Code 39','Enter Height of Bars (Points)')
    height = int(height)
    location = scribus.valueDialog('Code 39','Enter New X-Pos Y-Pos\n(No
         Comma, just space between)', '200 100')
    relpos = location.split()
    relx = int(relpos[0])
    rely = int(relpos[1])
    origx = relx
    humanread = scribus.valueDialog('Code 39','Print Human-Readable Code
         Too?\n(Any change means no)', 'Yes')
    scribus.setRedraw(1)
    scribus.setUnit(0)
    code = startstop
```

```
        for x in S[0:]:
          if x.isdigit():
            xnum = int(x)
            code = code + code39[xnum]
          elif ((ord(x) > 64) and (ord(x) < 91)):
              # ord(character) yields the ascii value
            xnum = ord(x) -55
            code = code + code39[xnum]
          elif (ord(x) == 32): # space
            code = code + code39[36]
          elif ((ord(x) == 36) or (ord(x) == 37)): # $ and %
            xnum = ord(x) + 1
            code = code + code39[xnum]
          elif ((ord(x) > 44) and (ord(x) < 48)): # -. /
            xnum = ord(x) -6
            code = code + code39[xnum]
          elif (ord(x) == 43): # +
            code = code + code39[42]
          elif (ord(x) == 42):
            code = code + startstop # * (adding for completeness)
          else:
            scribus.messageBox('OOPS!',S + '\nCode 39 does not encode this
                character: '+x,scribus.ICON_NONE,button1=scribus.BUTTON_OK)
            sys.exit(1)
        code = code + startstop
        for y in code[0:]:
          if y == 'n':
            relx = relx + narrow/2
            d = scribus.createLine(relx,rely,relx,rely + height,) #narrow line
            scribus.setLineWidth(narrow, d)
            scribus.setLineColor(b, d)
            scribus.setFillColor(b, d)
            relx = relx + narrow + narrow/2
          elif y == 'w':
            relx = relx + wide/2
            d = scribus.createLine(relx,rely,relx,rely + height,) #wide line
            scribus.setLineWidth(wide, d)
            scribus.setLineColor(b, d)
            scribus.setFillColor(b, d)
            relx = relx + narrow + wide/2
          elif y == 's': #for wide space
            relx = relx + wide
        if (humanread == 'Yes'):
          t = scribus.createText(origx, rely+height+3, relx -origx, 15)
          scribus.setText(S, t)
          scribus.setFont("DejaVu Sans Book", t)
          scribus.setFontSize(8, t)
          scribus.setTextAlignment(scribus.ALIGN_CENTERED, t)
        scribus.redrawAll()

    else:
        scribus.messageBox('OOPS!','You must have a document open to run this
            script',scribus.ICON_WARNING,button1=scribus.BUTTON_OK)
        sys.exit(1)
```

At the beginning, you can see the greater complexity of the bar and space sequences, where n and w represent narrow and wide lines. The s denotes a wider space than the default space I create between all bars, the narrow space.

User input starts with entering the sequence of numbers, letters, and symbols to be encoded. We then parse the entered sequence, creating one continuous string of n's, w's, and s's. In case we find a character which is not part of code 39, we bail out with the message at the bottom. Now we have this large string, which character by character creates the sequence of bars and spaces. We also have an option at the end to add human-readable characters to our barcode. Notice how we exactly size this text frame to the length of the code, then center the character glyphs.

As I suggested above, this Python code was then incorporated into the larger script which created the documentation page, complete with patient information and this barcode. Once I created this, I could then save the document on a hospital computer, so I wouldn't have to re-create the basic form again. When the patient was discharged, I deleted the file.

SETGUIDES2OBJECT.PY

Conceptually and in fact this is a simple script. You have some object on the page, and you want to create some guides that line up with the edges, so that you can create other content that also abuts one or more of these guides also.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# setguides2object.py
# Sets horizontal and vertical guides at the
# borders of a selected object
# preserves existing guides.

import scribus

if scribus.haveDoc():
    try:
      f = scribus.getSelectedObject()
      xpos, ypos = scribus.getPosition(f)
      width, height = scribus.getSize(f)
      scribus.setHGuides(scribus.getHGuides() + [ypos, ypos + height])
      scribus.setVGuides(scribus.getVGuides() + [xpos, xpos + width])
      scribus.setRedraw(1)
    except:
      result = scribus.messageBox('Error', 'You must select an object')
      sys.exit(1)
else:
    result = scribus.messageBox('Error','You need a Document open')
```

We first check to see if an object is selected, then although we don't directly check if there is a selected object, we use **try** in a way that an error is generated if we can't **`getSelectedObject()`**. Once we get the position and size of the object we know where the guides should be.

Something you do need to know is that you can't just add an individual guide. **`setHGuides()`** and **`setVGuides()`** require a list of all the guides you wish to have on the page, so you must get existing guides, presuming that you want to preserve those, then append any new guides.

RESIZEOBJECT.PY

Here's another simple script. We might even just call this a utility. All it does is enlarge or shrink an object by a certain amount. Perhaps there may only be a reason to use it if you need to resize a number of objects, so you could just go down the document, selecting objects and running the script. It would not take too much work to operate on more than one selected object.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# resizeobject.py
"""
Select an object, start script.
Enter a value to shrink/enlarge by, click Ok.
"""
try:
    import scribus
except ImportError:
    print "Unable to import the 'scribus' module. This script will only run
         within"
    print "the Python interpreter embedded in Scribus. Try Script->Execute
         Script."
    sys.exit(1)
if not scribus.haveDoc():
    scribus.messageBox('Scribus -Script Error', "No document open",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(1)
if scribus.selectionCount() == 0:
    scribus.messageBox('Scribus -Script Error', "There is no object
           selected.\nPlease select a frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
if scribus.selectionCount() > 1:
    scribus.messageBox('Scribus -Script Error', "You have more than one
           object selected.\nPlease select one frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
selected_frame = scribus.getSelectedObject()
scribus.setRedraw(False)
dimensions = scribus.getSize(selected_frame) # (width, height)
factor = scribus.valueDialog("Resize Object", "Resize by multiple or
           decimal fraction", "0.5")
factor = float(factor)
newwidth = dimensions[0]*factor
newheight = dimensions[1]*factor
scribus.sizeObject(newwidth, newheight)
scribus.setRedraw(True)
scribus.redrawAll()
```

At this point you should be quite familiar with these methods for checking for scribus, checking for a document, if an object is selected, and that only one object is selected.

The only user input is to give a floating point number for the change, the default being half-size or 0.5. Something you may notice is that the position of the object stays the same, that is, the **X-Pos**, **Y-Pos** of the upper left corner. It might be interesting to figure out the math so that the center of the object remains the same, but the corners go in or out in a symmetrical fashion. In case you might wonder, even if you set the object's basepoint to the center in Properties, the effect of the script is the same.

And so, here are the additions:

```
  dx = (dimensions[0] -newwidth)/2
  dy = (dimensions[1] -newheight)/2
  scribus.moveObject(dx, dy)
```

Put these 3 lines right after **`scribus.sizeObject()`**. Note that we don't need to know the **X-Pos** and **Y-Pos** of the object, since **`moveObject()`** simply deals with relative movement. If you do change the basepoint of the object to its center, you can see that this does not move with this resizing.

PASTE2ALL.PY

This script makes use of the only 2 Scripter commands I had a hand in creating. I noticed that there was a **`duplicateObject()`** command, and on looking at the C program code for it, I could see that what the command did internally was to copy the object, then paste it. I thought there was enough structure that I could split this into two new commands, **`copyObject()`** and **`pasteObject()`**, and with enough help from other Scribus developers, I managed to get it done.

The script for this chapter is one that I wrote to illustrate how one might use these commands. Although the name is **paste2all.py**, we'll see that it's more flexible than that.

```
# -*-coding: utf-8 -*
# File: paste2all.py
"""
USAGE
You must have a document open. Select a single object. Run the script,
  which asks whether you want all, odd, or even pages to get a copy of the
  selected object (no copy is made to the original page of the object). You
  may also specify the nth page of a recurring number of pages. For example,
  if you specify 2/3, the object will be copied to the second of every 3
  pages.
Any other input is ignored. When pasted, the copies go to the same page
coordinates of the original.
The script does not work with groups, and in fact Scribus will hang and
  then crash if you have selected a group. If you select more than one item
  without grouping, only one of those will be copied and pasted.
"""
import scribus
if scribus.haveDoc():
if scribus.selectionCount() == 0:
     scribus.messageBox('Scribus-Usage Error', "There is no object
              selected. \nPlease try again.", scribus.ICON_WARNING,
              scribus.BUTTON_OK)
     sys.exit(2)
     if scribus.selectionCount() > 1:
        scribus.messageBox('Scribus -Usage Error', "You have more than one
              object selected.\nPlease select one frame and try again.",
              scribus.ICON_WARNING, scribus.BUTTON_OK)
        sys.exit(2)
     paste2 = scribus.valueDialog('Paste to...',"Paste where?\n(all, odd,
              even)","all")
     selframe = scribus.getSelectedObject()
     pages = scribus.pageCount()
     currpage = scribus.currentPage()
     scribus.copyObject(selframe)
     if (paste2 == 'all'):
        i=1
        jump = 1
     elif (paste2 == 'odd'):
        i=1
        jump = 2
     elif (paste2 == 'even'):
        i=2
        jump = 2
```

```
      elif ('/' in paste2):
        pattern = paste2.split('/')
        i = int(pattern[0])
        jump = int(pattern[1])
      while (i <= pages):
        if (i != currpage):
          scribus.gotoPage(i)
          scribus.pasteObject(selframe)
        i = i + jump
      scribus.setRedraw(1)
      scribus.docChanged(1)
      scribus.messageBox("Finished", "Done",icon=0,button1=1)
  else:
      scribus.messageBox('Usage Error', 'You need a Document open', icon=0,
             button1=1)
      sys.exit(2)
```

You have seen my standard error-detection schemes, as we see at the beginning – we need to have a document, and an object selected. The first user input comes when we ask what sort of pattern of pasting is desired. The obvious choices in the **`messageBox()`** are all, odd, or even. Notice that I don't do any error-checking for unorthodox entries. If you enter something strange, the script fails all the options and is quickly over. There is also a weird possible entry, something like 1/3, or 4/5 as described in the USAGE comment. This chooses a single page at some integer interval. 1/3 pastes to the first of every 3 pages, 4/5 pastes to the 4th of every 5 pages. This may never be used, but the logic seemed interesting.

Notice that **`getSelectedObject()`** only gets you the name of the object, this is why the **`copyObject()`** command was needed.

So now if we look at how we manage these schemes, you can see that I set up a starting point, i, and jump factor. For '**all**' pages we start at 1 and jump by 1. For '**odd**' and '**even**' we jump two but start at page 1 or 2 respectively. Finally, with our weird scheme, we split the string at the '/', with the first integer being our starting point, then the second integer the jump factor.

After this, it's just a matter of going to the appropriate pages and pasting.

PIECHART.PY

Although I haven't used this script as often as I thought I would, if nothing else this represents another proof of concept sort of project. The idea was to use Scripter to create a pie chart by entering the data on the fly. At the time, there was no built-inway to do this with Scripter all by itself, but after playing around with SVGs I realized that one could create an SVG file using Python which could then be imported into Scribus.

```
#!/usr/bin/env python
# File piechart.py
# Automatically creates a piechart SVG file from a list of data

import math, scribus
# We'll create a list L, append to it, copy the list to a file at the end
L = ['<?xml version="1.0" encoding="UTF-8" standalone="no"?>\n']
L.append('<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"\n')
L.append('"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">\n')
L.append('<svg width="20cm" height="20cm" xmlns="http://www.w3.org/
        2000/svg" version="1.1">\n')
svgfile = scribus.valueDialog('SVG File','Enter name of file to save
        to\n".svg" will be appended')
svgfile = svgfile + '.svg'
nvalues = []
newvalues = scribus.valueDialog('Data Entry','Enter Data Values, separated
        by spaces')
nvalues = map(float, newvalues.split())
i=0
total = 0
seg = 0radius = 150
startx = 200
starty = 200lastx = radius
lasty = 0
ykey = 40
colors = ['red','blue','yellow','magenta','green', 'slateblue','slategrey',
         'greenyellow','wheat', 'violet']
bordercolor = 'black'
for n in nvalues:
    total = total + n
for n in nvalues:
    arc = "0"
    seg = n/total * 360 + seg
    if ((n/total * 360) > 180):
      arc = "1"
    radseg = math.radians(seg)
    nextx = int(math.cos(radseg) * radius)
    nexty = int(math.sin(radseg) * radius)
    L.append('<path d="M '+str(startx)+','+str(starty) + ' l '
            +str(lastx) +','+str(-(lasty))+ 'a150,150 0 ' + arc + ',0 '
            +str(nextx -lastx)+','+str(-(nexty -lasty))+ ' z" \n')
    L.append('fill="'+colors[i]+'" stroke="' + bordercolor + '" stroke-
            width="0.5" stroke-linejoin="round" />\n')
    L.append('<rect x="375" y="'+ str(ykey) + '" width="40" height="30"
            fill="'+colors[i] + '"stroke="black" stroke-width="0.5"/>\n')
    ykey = ykey + 35
    lastx = nextx
    lasty = nexty
    i += 1
L.append('</svg>')
output = open(svgfile,'w')
output.writelines(L)
output.close()
endmessage = svgfile + ' was created'
scribus.messageBox("Finished",endmessage,icon=scribus.ICON_NONE,
            button1=scribus.BUTTON_OK)
if scribus.haveDoc():
    scribus.placeSVG(svgfile, 100, 100)
    Totalframe = scribus.createText(340, 300, 100, 50)
    scribus.insertText("Total = " + str(total), 0, Totalframe)
```

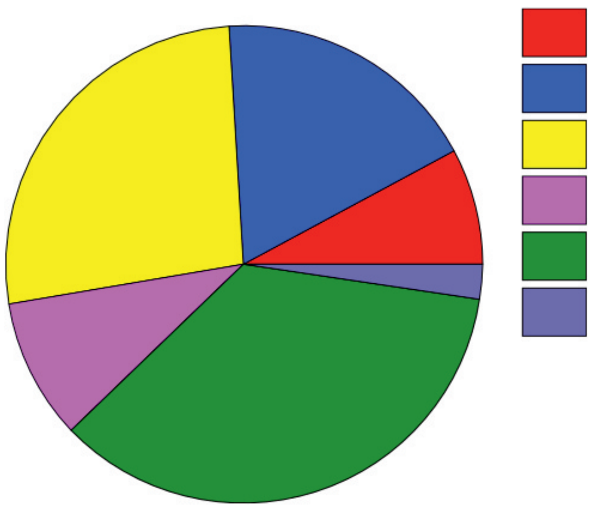
If you go through this script, you will realize that we only use Scripter for the dialogs to get the filename to save the SVG to, the data points, and at the end for a **`messageBox()`** and for an informational text frame, though this does place the SVG at 100,100. I can easily imagine this could be modified to use from the command line, and perhaps view the SVG in Inkscape.

As we begin the script we create a list L, which consists of the lines of the SVG file we are going to save. Once we create the preamble, we then ask for a filename, and also save the user from having to add the .svg extension. Next we ask the user to enter all the values for the segments of our chart, separated by spaces. We then use the **map** function to convert all of these to float values as we **`split()`** the string.

Subsequently, you can see that we have hardcoded the size and starting point of our pie chart, create a list of colors to use (and if there are more than 10 segments to the chart, we're in trouble). Because the math works out easier, I decided to start the first segment of the pie chart as a horizontal ray to the right of the center, after which we create right-sized arcs counterclockwise around the circle. The language you see here is how to create pie segments with SVG notation. Once we go through the data, we save the file.

One of the important bits of information about a pie chart is that you want to know the size of the whole data set, so this is why we create this text frame at the end. After the script runs, loading the SVG into Scribus is a manual operation, and of course you can then edit its overall size, placement on the page and even the colors quite easily. Something to be aware of in that regard is that this SVG will actually be a group of objects, so you have to **Ungroup** to be able to edit colors, borders, and even positioning of individual segments.

If you check the Scribus wiki at **https://wiki.scribus.net/canvas/Making_a_Pie_Chart** you will see a more detailed explanation of the math, and also an added feature, simultaneously creating a key to the side of the pie chart so that segments can be labeled easily. Here is an example of a completed piechart:



This piechart was created with a version of the script that also adds these rectangles on the side, to be used as a key with accompanying text to explain what each segment represents.

COLORCHART_GENERIC.PY

We had a request from the German nonprofit organization, freieFarbe e.V., to create a script that would take a color file, a plain text list of color names, and the values for making Lab colors, and use that to make color charts with appropriate labels. freieFarbe means "free colors", and their objective is to spread the use of freely available color sets, especially in the field of commercial printing. You can find them at **www.freieFarbe.de**.

The first task was to have a look at these files, and they go something like this in the CIE-HLC.clf file:

```
HLC 000 50 00  50       0     0
HLC 000 60 00  60       0     0
HLC 000 70 00  70       0     0
HLC 000 80 00  80       0     0
HLC 000 90 00  90       0     0
HLC 010 20 10  20       9,9   1,7
HLC 010 20 20  20       19,7  3,5
HLC 010 20 30  20       29,5  5,2
HLC 010 30 10  30       9,9   1,7
HLC 010 30 20  30       19,7  3,5
HLC 010 30 30  30       29,5  5,2
HLC 010 30 40  30       39,4  7
HLC 010 30 50  30       49,2  8,7
```

Each line begins with a color name, which may have some spaces in it, and then separated by tabs are 3 values, the L, a, and b which are used to create the color. Notice that the values use a comma for a decimal separator. Fortunately, the most recent development versions of Scribus are able to create colors using floating point Lab values.

```
#!/usr/bin/env python# -*-coding: utf-8 -*#
colorchart_generic.py
"""
# ABOUT
# This script is designed to create color swatches according to certain
# specifications, on A4 paper, with swatches# measuring 22x12mm.
# Color patches will be placed in a 7x7 array, colors created to# match the
# L, a, and b values for each color. A label is placed under# each color
# patch with its name, and the L, a, and b values.
# Each line of the clf file source should contain a color name, then its
# L, a, and b values separated by tabs. If the Lab data uses commas
# for the decimal separator, they will be converted to decimal points.
# A dialog asks for a heading for the pages.
"""
import scribusimport re, os, sys, datetime

def SelectAllText(textframe):
    texlen = scribus.getTextLength(textframe)
    scribus.selectText(0,texlen,textframe)
    return

colors = []
L = []
a = []
b = []
index = 0
ypos = -5 # this was used as a correction factor for placement
```

```
scribus.messageBox("About",'This script will take the Lab colour data
           \nfrom a clf colour file, create those colours\nin Scribus, and
           in addition create a \ndocument displaying each of these
           colours.')
colorfile = scribus.fileDialog('Select a color file', 'CLF files(*.clf)')
headingparts = os.path.split(colorfile)
heading = headingparts[-1]
heading = heading[:-4]
header = scribus.valueDialog("Title",'Enter a title for your color chart',
           heading)
for line in open(colorfile, 'rU').xreadlines():
    content = line.split('\t')
    fields = len(content)
    if (fields == 4):
    colors.append(content[0])
    Lval = content[fields -3]
    Lval = re.sub(r',','.',Lval)
    Lval = float(Lval)
    L.append(Lval)
    aval = content[fields -2]
    aval = re.sub(r',','.',aval)
    aval = float(aval)
    a.append(aval)
    bval = content[fields -1]
    bval = re.sub(r',','.',bval)
    bval = float(bval)
    b.append(bval)
size = len(colors)
docsize = int(size/49)
plural = 's'
if ((size % 49) > 0):
    docsize = docsize + 1
if docsize == 1:
    plural = ''
if size > 500:
    response = scribus.messageBox('Query', 'You are about to create '
           +str(size) + ' colors,\n anda document with '+str(docsize)+
           ' page' + plural + '.\nThis may take several minutes.\nDo you
           want to do this?', icon=scribus.ICON_WARNING,
                button1=scribus.BUTTON_YES, button2=scribus.BUTTON_NO)
else:
    response = scribus.messageBox('Query', 'You are about to create '
           +str(size) + ' colors,\n anda document with '+str(docsize)+
           ' page' + plural + '.\nDo you want to do this?',
           icon=scribus.ICON_NONE, button1=scribus.BUTTON_YES,
             button2=scribus.BUTTON_NO)
if response != 16384:
    sys.exit(-1)
#start = datetime.datetime.now().replace(microsecond=0)
scribus.progressReset()
scribus.progressTotal(docsize)
currpage = 1
if scribus.newDocument(scribus.PAPER_A4, (0,0,0,0),scribus.PORTRAIT, 1,
           scribus.UNIT_POINTS, scribus.NOFACINGPAGES,
                scribus.FIRSTPAGERIGHT,1):
    scribus.setUnit(scribus.UNIT_MILLIMETERS)
    scribus.setRedraw(False)
    scribus.createCharStyle("titlechar", "DejaVu Sans Condensed Bold", 18)
    scribus.createParagraphStyle("titlestyle",1,0,0,0,0,0,0,0,0,0,0,
           "titlechar")
    scribus.createCharStyle("keychar", "DejaVu Sans Condensed Bold",
           7.0,'','Black',1.0,'',0,0,0,0,0,0,0,0,0,1,1,-50)
    scribus.createParagraphStyle("keystyle",0,7.0,1,0,0,0,0,0,0,0,0,
           "keychar")
    scribus.createCharStyle("keyLabchar", "DejaVu Sans Book", 6.0)
    scribus.createParagraphStyle("keyLabstyle",0,7.0,1,0,0,0,0,0,0,0,0,
           "keyLabchar")
    scribus.createCharStyle("footerchar", "DejaVu Sans Book", 7)
    scribus.createParagraphStyle("footerstyle",0,12.0,2,0,0,0,0,0,0,0,0,
           "footerchar")
    scribus.createMasterPage('attribution')
    scribus.editMasterPage('attribution')
    footer = scribus.createText(97,285.3,99,5.2)
    scribus.setText('Scribus script created by Gregory Pittman &
           freieFarbe.de 2017', footer)
    scribus.setStyle("footerstyle", footer)
    leftfooter = scribus.createText(15.5, 281, 80, 9.5)
    scribus.setText(heading + '\nNo. of colours: ' + str(size) + ' ' +
           u'\u00b7' + ' Page \x1e of \x17', leftfooter)
    SelectAllText(leftfooter)
    scribus.setStyle("footerstyle", leftfooter)
    scribus.setTextAlignment(scribus.ALIGN_LEFT, leftfooter)
    scribus.closeMasterPage()
    scribus.gotoPage(1)
    titleframe = scribus.createText(30, 15, 90, 15)
    scribus.setText(heading, titleframe)
    scribus.setStyle("titlestyle", titleframe)
    key = scribus.createText(15.5, 43.5, 8, 18)
    scribus.setText("Name\n\nL\na\nb", key)
    SelectAllText(key)
    scribus.setStyle('keystyle', key)
    textlen = scribus.getTextLength(key)
    scribus.selectText(textlen-6, 6, key)
    scribus.setStyle('keyLabstyle', key)
    while (index < len(colors)):
      if ypos < 230:
        ypos = ypos + 35
      else:
        ypos = 30
        scribus.newPage(-1)
        currpage = currpage + 1
      for xpos in range(25, 180, 25):
        scribus.defineColorLab(colors[index], float(L[index]),
               float(a[index]), float(b[index]))
        newrectangle = scribus.createRect(xpos,ypos,22,12)
        scribus.setLineColor("None",newrectangle)
        scribus.setFillColor(colors[index], newrectangle)
        newtext = scribus.createText(xpos, ypos + 13.5, 22, 15)
        if (len(colors[index])<18):
           spacer = '\n'
        else:
           spacer = ''

        data = str(L[index]) + '\n' + str(a[index]) + '\n' + str(b[index])
        scribus.setText(colors[index]+spacer+'\n'+ data, newtext)
        SelectAllText(newtext)
```

```
            scribus.setStyle('keystyle',newtext)
            textlen = scribus.getTextLength(newtext)
            datalen = len(data)
            overflow = scribus.textOverflows(newtext)
            if overflow == 1:
              scribus.sizeObject(22,20, newtext)
            scribus.selectText(textlen -datalen, datalen, newtext)
            scribus.setStyle('keyLabstyle', newtext)
            index = index + 1
            if index >= len(colors):
              break
          scribus.progressSet(currpage)

    pages = scribus.pageCount()
    for p in range(1, pages + 1):
        scribus.applyMasterPage('attribution',p)
    scribus.progressReset()
    scribus.deselectAll()
    #finish = datetime.datetime.now().replace(microsecond=0)
    #interval = finish -start
    #scribus.messageBox("Interval",'This took ' + str(interval) + ' to
    finish',button1=scribus.BUTTON_OK)
    scribus.setRedraw(True)
    scribus.redrawAll()
```

At the top, in addition to some packages we've seen before, we also import re and datetime. More on these later.
The script otherwise begins with a function **`SelectAllText()`**, which is discussed in the chapter **Creating Text Styles**. After this, we initialize some variables and lists and then gives a message explaining what this script is about. We then ask for a color file using a **`fileDialog()`**, and since we're only interested in these color files, we sift out those ending with **.clf**. I then sift out the name of the file to potentially be used as the default header for the document we will create later, but a **`valueDialog()`** allows the user to make the heading whatever is desired.

Now we're ready to read this file, line by line:

```
    for line in open(colorfile, 'rU').xreadlines():
```

The part I wanted to point out is the 'rU' parameter for the file. What I found as I was writing this script was that some of the **.clf** files had been created and saved on a Mac computer. The importance of this is that a text file saved on a Mac does not use a newline character to end each line, and therefore Python running on UTF-8 will read the entire file as one line. For this and other reasons I put in the conditional, **`if (fields ==4):`** later to identify these files. I thought I was going to have to convert these Mac files, but then I discovered this '**rU**' parameter, which covers Mac and Windows methods of indicating new lines. I left the conditional in just in case some file had some other kind of error, and I did find a file that had a missing tab.

Next comes parsing our values, and here the re package comes in to change the decimal separators to periods. At the end of parsing, we have four lists for our color names and the Lab values to create them. Some of these files were quite large, with thousands of colors, so in order to give the user some feedback about this, we make note of the number of colors and also the number of pages of document that 49 colors per page will create. Next a messageBox() displays this information, and gives the user an option to bail out and quit the script. For whatever reason, if you click an OK button, this returns the integer **16384**, thus the reason for this check.

I've commented out the line beginning with **`#start = ...`** along with two others later – I'll discuss these below.

The next two lines

```
scribus.progressReset()
scribus.progressTotal(docsize)
```

set up a Progress Bar in the right lower edge of the Scribus window for some feedback about the progress of the script – we're using docsize as the denominator to check progress.

We're finally ready to begin the document, A4 size, with units in millimeters. Among the preparatory work is that of creating a number of Character and Paragraph Styles to be used in the document, and then a Master Page, which will have some basic information which will be on all pages. Look at what's in the footer:

```
scribus.setText(heading + '\nNo. of colours: ' + str(size) + ' ' +
          u'\u00b7' + ' Page \x1e of\x17', leftfooter)
```

We're indicating the number of colors in the document, then Unicode **00b7**, which is a middle dot. But what are '**\x1e**' and '**\x17**'? These turn out to be the hexadecimal notation in Scribus for the current page and the total number of pages. Since this is a Master Page, we want the current page to change accordingly. After this we close our Master Page editing and begin the first page of the document. Only the first page will show a header and also a key which explains the Lab values we willdisplay under each color swatch.

As I said, each page will contain a maximum of 49 color swatches, 7 rows of 7 swatches, the size and placement of which was decided in some to and fro discussions with freieFarbe. Underneath each swatch is a small text frame showing the color name, then a line each for the **L**, the **a**, and **b** values. Since we are going through our lists with an indexing method, it seemed efficient to create each color, then its swatch, then the swatch's key, then move to the next color. We have to know when we've filled a page so we can make a new one, and know when all the colors are created. After the first row of a page iscreated, we reset our Progress Bar with **`scribus.progressSet(currpage)`**. When we finish the document, then we are ready to apply the Master Page to all pages.

```
pages = scribus.pageCount()
for p in range(1, pages + 1):
    scribus.applyMasterPage('attribution',p)
```

This shows a peculiarity of the range command in Python. If you set a range of (1, 10) the conditional actually uses 1 through 9, so this is why I needed to add one to the pages variable.

I did not cover the application of the styles to the labels since this is handled in Creating Text Styles.

Now let's talk about those lines commented out:

```
#start = datetime.datetime.now().replace(microsecond=0)

#finish = datetime.datetime.now().replace(microsecond=0)
#interval = finish -start
#scribus.messageBox("Interval",'This took ' + str(interval) + ' to
        finish',button1=scribus.BUTTON_OK)
```

When I began trying out this script on the large number of .clf files which I had, I quickly found that the larger files would run for an hour or more! This is part of what led to the warning messages, but I wanted to get some objective measure of this, and found this method using the datetime package. This sets a start time somewhere before the bulk of the work, then after the document is finished a finish time so we can calculate the interval, followed by a simple message about this. This also helped understand the huge benefit of creating these styles, since originally the script simply applied text features to each label frame as they were created. Somehow this ends up being very time-consuming. Using styles instead meant that only minutes were needed for even the largest files.

If you're going to try out this script, make sure you're using Scribus 1.5.4svn or better. There should be an official release of 1.5.4 later this year. You can download a collection of .clf files from freieFarbe's website.

CIE-HLC

freieFarbe

Name	L	a	b
HLC 000 10 00	10.0	0.0	0.0
HLC 000 20 00	20.0	0.0	0.0
HLC 000 30 00	30.0	0.0	0.0
HLC 000 40 00	40.0	0.0	0.0
HLC 000 50 00	50.0	0.0	0.0
HLC 000 60 00	60.0	0.0	0.0
HLC 000 70 00	70.0	0.0	0.0
HLC 000 80 00	80.0	0.0	0.0
HLC 000 90 00	90.0	0.0	0.0
HLC 010 20 10	20.0	9.9	1.7
HLC 010 20 20	20.0	19.7	3.5
HLC 010 20 30	20.0	29.5	5.2
HLC 010 30 10	30.0	9.9	1.7
HLC 010 30 20	30.0	19.7	3.5
HLC 010 30 30	30.0	29.5	5.2
HLC 010 30 40	30.0	39.4	7.0
HLC 010 30 50	30.0	49.2	8.7
HLC 010 40 10	40.0	9.9	1.7
HLC 010 40 20	40.0	19.7	3.5
HLC 010 40 30	40.0	29.5	5.2
HLC 010 40 40	40.0	39.4	7.0
HLC 010 40 50	40.0	49.2	8.7
HLC 010 40 60	40.0	59.1	10.4
HLC 010 50 10	50.0	9.9	1.7
HLC 010 50 20	50.0	19.7	3.5
HLC 010 50 30	50.0	29.5	5.2
HLC 010 50 40	50.0	39.4	7.0
HLC 010 50 50	50.0	49.2	8.7
HLC 010 50 60	50.0	59.1	10.4
HLC 010 50 70	50.0	68.9	12.2
HLC 010 60 10	60.0	9.9	1.7
HLC 010 60 20	60.0	19.7	3.5
HLC 010 60 30	60.0	29.5	5.2
HLC 010 60 40	60.0	39.4	7.0
HLC 010 60 50	60.0	49.2	8.7
HLC 010 70 10	70.0	9.9	1.7
HLC 010 70 20	70.0	19.7	3.5
HLC 010 70 30	70.0	29.5	5.2
HLC 010 70 40	70.0	39.4	7.0
HLC 010 80 10	80.0	9.9	1.7
HLC 010 80 20	80.0	19.7	3.5
HLC 010 90 10	90.0	9.9	1.7
HLC 020 20 10	20.0	9.4	3.4
HLC 020 20 20	20.0	18.8	6.8
HLC 020 20 30	20.0	28.2	10.3
HLC 020 30 10	30.0	9.4	3.4
HLC 020 30 20	30.0	18.8	6.8
HLC 020 30 30	30.0	28.2	10.3
HLC 020 30 40	30.0	37.6	13.7

TEMP_GRAPHING.PY

This script is a modification of another that is depicted in the Scribus wiki in the page Advanced Graphing. That project came about after I wanted to make a script which depicted daily events that could occur at some specific time of day or night. I lumped these together in occurrences per hour of time.

From this I wanted to make something a little more flexibile, since in the original example the Y axis, the number of events, by definition were always zero or more, so the bottom of Y origin was always zero. I looked around trying to find some data which I could use to depict in a graph. Eventually, I decided to accumulate my own data, based on hourly temperatures where I live. At the end of this chapter I explain how I did this in a semi-automatic fashion.

Conceptually what I needed to do in Scribus was to import the lines from the file, parse out the temperatures, create an appropriate graphical structure, then create the line that depicts these measurement. Along the way there needed to be some decision points about the graph to make sure the Y axis is appropriate for the data.

```
#!/usr/bin/env python
# -*-coding: utf-8 -*
# File: temp_graphing.py
# creates graph with axes and plots Y values
# X values are fixed

import scribus, os, sys

xorigin = 80 # x-origin of graph
yorigin = 520 # y-origin of graph
xaxis=637 # x axis length
yaxis=408 # y axis length
color="Black"
colors = ''
linecolor = 'Black'
colorcount = 0
a = 1.5 # width of lines
l = 2.0 # width of plotted line
t = 0.8 # width of scale markers
xvalue = []
polyvalue = []
i=0
yvalue = []
yvalues = ''
time = ['00','01','02','03','04','05','06','07','08','09','10','11','12',
           '13','14','15','16','17','18','19]
deg = u"\u00b0" # unicode for degree sign
if scribus.haveDoc():
    tempfile = scribus.fileDialog('Select Your Data File','Text Files
           (*.txt)')
    for line in open(tempfile).xreadlines():
      content = line.split()
      yvalue.append(float(content[1]))
      yvalues = yvalues + content[1] + ' '
      i += 1
    colornames = scribus.getColorNames()
    for coloritem in colornames:
      colors = colors + coloritem + ' '
      colorcount += 1
    if (colorcount == 4):
      colorcount = 0
      colors = colors + '\n'
    linecolor = scribus.valueDialog('Enter a Line Color Name',
            colors,'Black')
    scribus.setRedraw(1)
    scribus.setUnit(0)
    xtick = xorigin
    xscale = xaxis/i # this gives you typographic points per data unit
    yrange = float(scribus.valueDialog('Y Axis Maximum', 'Enter Top Value
            for Y Axis\n'+yvalues+'\n(Here is your data)'))
    ytick = yoriginyoffset = float(scribus.valueDialog('Y Axis
            Origin','Enter Y value of graph origin', '0'))
    yscale = float(yaxis/(yrange -yoffset))
# this gives you typographic points per data unit
    yCelscale = float(yscale * 9.0 / 5.0) # typographic points per Celsius
    unitcreateaxes = scribus.valueDialog('Create Axes?','Do you want to
            create axes for this graph? \nChanging to anything else means
            No','Yes')
    nticks = 0
    while (nticks < i):
      xtick = xtick + xscale
      xlabel = xtick -9.75
      xvalue.append(xtick)
      if (createaxes == 'Yes'):
        xt = scribus.createLine(xtick, yorigin, xtick, yorigin + 5)
# creating X axis tick lines
        scribus.setLineWidth(t, xt)
        scribus.setLineColor(color, xt)
        scribus.setFillColor(color, xt)
        xbox = scribus.createText(xlabel, yorigin + 10, 20, 18)
        scribus.setText(time[nticks], xbox)
        scribus.setFont("DejaVu Sans Book", xbox)
        scribus.setFontSize(12, xbox)
        scribus.setTextAlignment(scribus.ALIGN_CENTERED, xbox)
        if (nticks == 0): # only need to do these once
          d = scribus.createLine(xorigin,yorigin,xorigin+xaxis,yorigin)
# create X axis
          scribus.setLineWidth(a, d)
          scribus.setLineColor(color, d)
          scribus.setFillColor(color, d)
          e = scribus.createLine(xorigin,yorigin,xorigin,yorigin-yaxis)
# create Y axis
          scribus.setLineWidth(a, e)
          scribus.setLineColor(color, e)
          scribus.setFillColor(color, e)
          f = scribus.createLine(xorigin + 637,yorigin,xorigin +
              637 ,yorigin-yaxis)
# create secondary Y axis
          scribus.setLineWidth(a, f)
          scribus.setLineColor(color, f)
          scribus.setFillColor(color, f)
          ymark = int(scribus.valueDialog('Tick Mark Interval', 'Enter Y
              Interval for Tick Marks\n(Fahrenheit)', '10'))
          ytick = ytick -ymark * yscale
          while (ytick >= (yorigin -yaxis -1)):
            yt = scribus.createLine(xorigin, ytick, xorigin -5, ytick)
# creating Y axis tick lines
            scribus.setLineWidth(t, yt)
            scribus.setLineColor(color, yt)
            scribus.setFillColor(color, yt)
            ytick = ytick -ymark * yscale
          Celsorigin = float((yoffset -32) * 5.0 / 9.0)
          firstCelstick = int(Celsorigin) + 1
          while 1:
            if firstCelstick % 5 == 0:
              break
            else:
              firstCelstick += 1
          yCentries = ''
          yCtickstart = yorigin -(float(firstCelstick -Celsorigin)) *
                  yCelscale
          while yCtickstart > yorigin -yaxis:
            yCentries = str(firstCelstick) + '\n\n' + yCentries
            yC = scribus.createLine(xorigin + xaxis, yCtickstart, xorigin
                  + xaxis + 5, yCtickstart)
            yCtickstart = yCtickstart -5.0 * yCelscale
            firstCelstick += 5
          yentries = ''
          while yrange > yoffset:
            yentries = yentries + str(int(yrange)) + '\n\n'
            yrange = yrange-ymark
          ylabel = scribus.createText(xorigin -40,yorigin -412,27.75, 415)
          scribus.setText(yentries, ylabel)
          scribus.setFont("DejaVu Sans Book", ylabel)
          scribus.setFontSize(12, ylabel)
          scribus.setTextAlignment(scribus.ALIGN_RIGHT, ylabel)
          yClabel = scribus.createText(xorigin + xaxis + 5, yorigin -403,
                  21.75, 415)
          scribus.setText(yCentries, yClabel)
          scribus.setFont("DejaVu Sans Book", yClabel)
          scribus.setFontSize(12, yClabel)
          scribus.setTextAlignment(scribus.ALIGN_RIGHT, yClabel)
          flabel = scribus.createText(xorigin -30, yorigin -438, 30, 22)
          scribus.setText(deg + 'F', flabel)
          scribus.setFont("DejaVu Sans Condensed Bold", flabel)
          scribus.setFontSize(20, flabel)
          clabel = scribus.createText(xorigin + 633, yorigin -438, 30, 22)
          scribus.setFont("DejaVu Sans Condensed Bold", clabel)
          scribus.setFontSize(20, clabel)
      nticks += 1
    item = 0
    while (item < i):
# this interleaves the X and Y data for createPolyLine
      polyvalue.append(xvalue[item])
      polyvalue.append((yorigin + yscale * yoffset -yvalue[item] * yscale))
      item += 1
    poly = scribus.createPolyLine(polyvalue) # this is the actual plotting
    scribus.setLineWidth(l, poly)
    scribus.setLineColor(linecolor, poly)
    scribus.setFillColor("None",poly)
    scribus.selectObject(poly)



else:
    if scribus.newDocument(scribus.PAPER_LETTER, (20,20,20,20),
             scribus.LANDSCAPE, 1, scribus.UNIT_POINTS, scribus.PAGE_1, 0,
             1):
    scribus.messageBox('Message', 'You did not have a document open, so \n
             one has been createdfor you.\n\nNow you can run the script
             again.', scribus.ICON_NONE, scribus.BUTTON_OK)
scribus.redrawAll()
```

At the beginning, a number of variables are initialized, with the X, Y origin and length of the axes hardcoded. Every element of the graph is referenced to the variables xorigin and yorigin. Because of this, the placement of the graph on the page can be adjusted by changing these values and nothing else. The script is designed to work with a US Letter page in landscape orientation, but actually could be used with A4 paper by just adjusting these two variables.

Something I have done here is to allow for the user to have already created a document with **`if scribus.haveDoc()`**:, but if not, the alternative else: clause creates one, even though the script then ends and must be started again. This ends up being more convenient than just ending the script and having the user manually create a new document.

Next, using **`fileDialog()`**, the user must choose a file, an individual line of which looks like this:

```
Temperature: 60.1 F (15.6 C) Tue Mar 6 14:01:18 EST 2018
```

The split method works well here, dividing this line according to where spaces are. The second string, the Fahrenheit temperature, is what we're wanting, obtained with **`content[1]`**, remembering that the first, **`content[0]`** would be **Temperature:**. I don't really need the date or time, since individual files cover a 24-hour period, with hour 00 the first line and hour 23 the last line.

Since you have a choice as to the color of the graph line, we need to get a list of colors available from Scribus, then ask the user which to choose, with a default of Black. The next step is for the user to decide what the top value of the Y axis should be, and in the **`valueDialog()`** he is shown all the data imported from the file. Then, one chooses the value for the Y origin, which might be something other than zero, like 20 or even -10. With these maximum and minimum data values for the Y axis, a scaling factor, yscale, can then be calculated. There should always be 24 data points, so the X axis scaling factor, **xscale**, needs no user input.

The next **`valueDialog()`** wants to know if axes should be created. If you have already created a graph and just want to make another graph line on it, then change the default **Yes** to anything else. I like this method since not only is coding easier, but user input is easier as well. The last place for user input comes after we've begun making the axes and tick marks. If you have chosen not to create axes, then this **`valueDialog()`** is skipped, along with all the other axes and tick-creating code.

At long last, we graph the data with this relatively small part of the script:

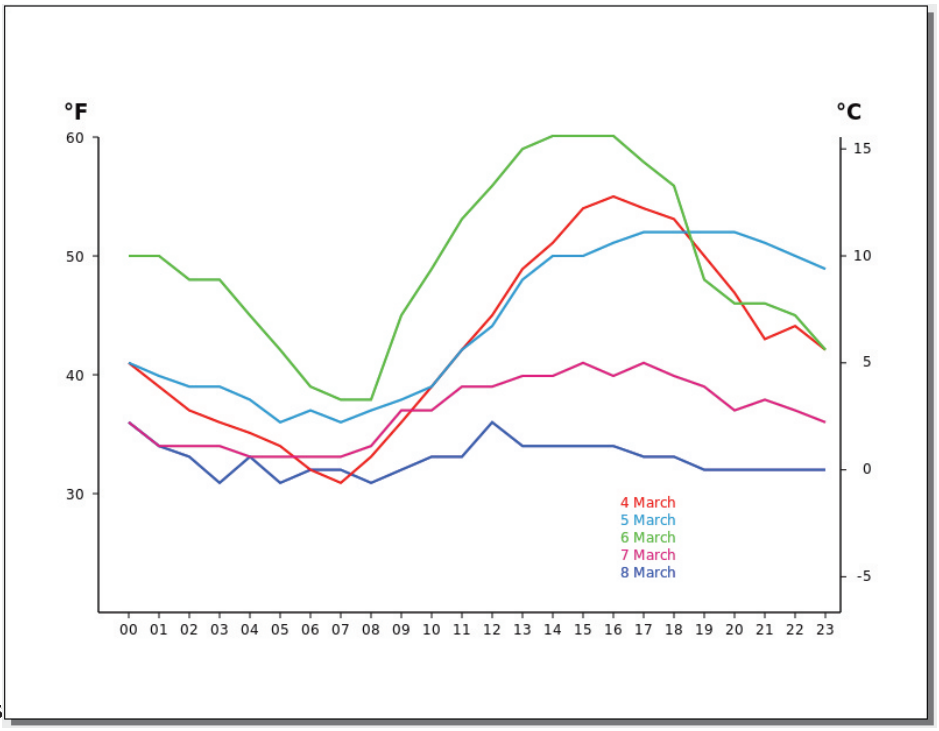
```
item = 0
while (item < i): # this interleaves the X and Y data for createPolyLine
    polyvalue.append(xvalue[item])
    polyvalue.append((yorigin + yscale * yoffset -yvalue[item] * yscale))
    item += 1
poly = scribus.createPolyLine(polyvalue) # this is the actual plotting
scribus.setLineWidth(1, poly)
scribus.setLineColor(linecolor, poly)
scribus.setFillColor("None",poly)
scribus.selectObject(poly)
```

The hardest part to get right is the calculation of the **Y-Pos** values, particularly considering that higher data values translate to smaller **Y-Po**s values, since a **Y-Pos** of zero is the top of the page.

Here is an even trickier task:

```
Celsorigin = float((yoffset -32) * 5.0 / 9.0)
firstCelstick = int(Celsorigin) + 1
  while 1:
    if firstCelstick % 5 == 0:
      break
    else:
      firstCelstick += 1
  yCentries = ''
  yCtickstart = yorigin -(float(firstCelstick -Celsorigin)) * yCelscale
  while (yCtickstart > yorigin -yaxis):
    yCentries = str(firstCelstick) + '\n\n' + yCentries
    yC = scribus.createLine(xorigin + xaxis, yCtickstart, xorigin +
        xaxis + 5, yCtickstart)
    yCtickstart = yCtickstart -5.0 * yCelscale
    firstCelstick += 5
```

When this script was originally written, it only made a left Y axis and ticks for Fahrenheit values, but I wanted to also create marks for Celsius. The problem starts with figuring out what the value of the bottom tick should be and where it should be. I start out by taking the Fahrenheit value of the Y origin and converting to Celsius, very likely to be an odd decimal value. I truncate this with **int**, chopping off the decimal, then add 1. Next I need to find the next value divisible by 5, since I decided to space the Celsius ticks every 5 degrees Celsius. Once I have this first tick, then placing the rest is fairly straightforward. With a few more embellishments, like scales and labels, we get something like this as the result:



Here I show 5 cons way the script works, I can even save this document, open it, then run some time later to add more data, as long as I pay attention to the limits of the Y axis, 60 and 20 in this case, since these will need to be entered as the script runs. Any additional labels, such as a titlefor the graph or a key to indicate which color corresponds to a particular date, would be manual operations in Scribus, and I don't see a reason to add these to the script. As you can see, I have added a text frame and put in a key for the different graph line colors. Something else which is a manual operation is the adjustment of the linespacing for the Fahrenheit and Celsius scales. The easiest way to do this is to first make sure the top values is lined up with the topmost tick (by moving the frame as needed), then adjust the linespacing with the mouse wheel – remember if you hold down Shift while scrolling, the changes in the spinbox are by tenths of a point.

SOME LIMITATIONS

Something that doesn't work well is if you might, for example, set the value of the Y origin to 25, and the tick interval to 10. The ticks probably come out alright, but the markers should say 55, 45, 35... and they will instead be 60, 50, 40... If you have to extend your Y axis because new data would be off the chart, you are probably better off starting all over rather than trying to edit the graph, then run the script only with the new data. It might be tempting to make a new page, then copy and paste the polylines with careful shrinkage vertically to match up to the new scale.

Because of the automated way that the X axis works, you couldn't run this where there were gaps in the data, but what you could do in that case would be to add lines to the data file where there is a gap, then in Scribus afterward edit the polyline shape to snip out the artificial data, then move the accurate parts horizontally if needed.

MAKING THE DATA FILE

I suppose it's possible to find historical weather data somewhere, but I thought it would be more interesting to do this with a cron task. I have a command line program called **weather** (the latest version is **weather-2.3**). Once I had this downloaded, unzipped and untarred, I can go to its directory and type something like:

```
./weather "Louisville East CCD"
```

and the output looks like this:

```
Searching via name...
[using result Louisville East CCD, KY]
Current conditions at Bowman Field, KY
Last updated Mar 07, 2018 -10:53 AM EST / 2018.03.07 1553 UTC
Temperature: 39.0 F (3.9 C)
Relative Humidity: 59%
Wind: from the NW (320 degrees) at 10 MPH (9 KT) gusting to 26 MPH (23 KT)
Sky conditions: overcast
```

This led then to the following shell script (notice the pipe to the grep command):

```
#! /usr/bin/env sh
# weather.sh -created to be a cron job
cd /home/gregp/Downloads/weather-2.3
./weather "Louisville East CCD" | grep Temperature >> /home/gregp/temp.txt
'date' >> /home/gregp/temp.txt
```

and its output looks like this, which is appended to a file temp.txt:

```
Temperature: 39.0 F (3.9 C)
Wed Mar 7 11:32:03 EST 2018
```

Notice that I did not try to **grep** out the date and time, but simply used the shell command date. I then placed an executable copy of this shell script in **/etc/cron.hourly** in one of my computers that is always running and connected to the internet. As long as that computer is on, it accumulates these readings hourly. The last steps to make this usable by the script is to open the temp.txt file into Kwrite, so that I can do a Replace using regex to convert '\)\n' to '\) ', converting each newline character after the closing parentheses to a space, so now each line looks like this:

```
Temperature: 39.0 F (3.9 C) Wed Mar 7 11:32:03
```

which is parsable by my script. The last step is to make an individual file of 24 lines for each date's data, and this is the file I look for with the **`fileDialog()`**.
Note: you can obtain **weather** from **http://fungi.yuggoth.org/weather/src/**

CREATING TEXT STYLES

We certainly encourage the use of Paragraph and Character Styles in Scribus. Aside from easily applying a number of font and paragraph features all at once, you also can subsequently edit your style and have any changes applied document-wide at the same time. It makes sense, then, that we may want to create these styles on the fly as we go through a script. It is possible to import styles from one document to another, but you may need to refresh your memory about where some particular style is located.

So, the short answer is that yes, we can create styles in Scripter. At the start, it's helpful to know that for most purposes there are two components you must create. The first is a Character Style, which denotes the font, its size, and then a large number of features, including its fill color and stroke color.

After that, you then create a Paragraph Style, which includes the name of the Character Style you wish to use, then such things as linespacing, alignment, spacing between paragraphs, indentation, and so on.

This is about as simple as it can be:

```
scribus.createCharStyle("titlechar", "DejaVu Sans Condensed Bold", 18)
scribus.createParagraphStyle("titlestyle",1,0,0,0,0,0,0,0,0,0,0,
          "titlechar")
```

For our Character Style we have only specified its name, the font, and the font size. For the Paragraph Style we really only wanted to specify its name, the linespacing mode (in this case automatic), and the associated Character Style, but we can't truncate the elements between the linespacing mode and the name of the Character Style.

Recently, someone wanted a script with created styles, and specified that he wanted the kerning or tracking to be -5.0%. Easy enough to say, but... Let's look at what the online manual says.

```
createCharStyle(...)
Creates a character style. This function takes the following keyword
  parameters:
"name" [required] -> name of the char style to create
"font" [optional] -> name of the font to use
fontsize [optional] -> font size to set (double)
"features" [optional] -> nearer typographic details can be defined by a
  string that might contain the following phrases
comma-separated (without spaces!):
    inherit
    bold
    italic
    underline
    underlinewords
    strike
    superscript
    subscript
    outline
    shadowed
    allcaps
    smallcaps
"fillcolor" [optional], "fillshade" [optional] -> specify fill options
"strokecolor" [optional], "strokeshade" [optional] -> specify stroke
  options
baselineoffset [optional] -> offset of the baseline
shadowxoffset [optional], shadowyoffset [optional] -> offset of the shadow
  if used
outlinewidth [optional] -> width of the outline if used
underlineoffset [optional], underlinewidth [optional] -> underline options
  if used
```

```
strikethruoffset [optional], strikethruwidth [optional] -> strikethru
  options if used
scaleh [optional], scalev [optional] -> scale of the chars
tracking [optional] -> tracking of the text
"language" [optional] -> language code
```

As you can see, tracking is next to last in the options. I thought I could just count down the bullets, maybe skipping the white bullets to see which one was tracking. I thought it was 13th. After days of trial and error, mostly error, I finally looked at this list and considered that, for example, fillcolor and fillshade are actually two options, and so going through the list again, maybe tracking is 19th? This turned out to be correct, but then, how do you set it for -5%? Answer: by playing with numbers. Meanwhile, there was also the trick of deciding what value to put for an option that you really didn't want to set.

The end result of all this consternation was this:

```
scribus.createCharStyle("keychar", "DejaVu Sans Condensed Bold",
          7.0,'','Black',1.0,'',0,0,0,0,0,0,0,0,0,1,1,-50)
scribus.createParagraphStyle("keystyle",0,7.0,1,0,0,0,0,0,0,0,0,"keychar")
```

So the 4th and 7th elements of **`createCharStyle()`** needed to be null strings. The sixth, fillshade was 1.0 for 100%. The 17th and 18th are 1 for 100% (horizontal and vertical scales), and the others could be zeros. But look at number 19, the kerning/tracking – somehow -50 translates to -5.0%!

From what I can tell, if I wanted something like underlining, that 4th element null string would be 'underline'.

APPLYING A PARAGRAPH STYLE

I had thought that applying a style to a frame would be easy, something like on the main canvas, where you select a frame, apply the style, and you are done! When I tried this in a script, the Paragraph Style was only applied to the last paragraph, or in other words, from the end of the text to the first newline character that was encountered.

The workaround, since I was making a number of styles and applying them, was to create a function like this, to not just select the frame but select all the text in it:

```
def SelectAllText(textframe):
    texlen = scribus.getTextLength(textframe)
    scribus.selectText(0,texlen,textframe)
    return
```

Functions go at the top of your script, right after all the import statements. So then down in the script, after creating the styles I could then just:

```
SelectAllText(key)
scribus.setStyle('keystyle', key)
```

Something I haven't tried is to see what would happen if I applied a style to a frame BEFORE it had content – would this work?

AN UNANTICIPATED BONUS

To my happy surprise, there was a side benefit to creating and using styles. If you recall, I mentioned that some of these color files contained thousands of colors. What the script does is to create a list of the color names, and corresponding lists of the values used to create these colors in Scribus. After the colors are created in Scribus, a document is created with small color swatches for each color, 49 per page, with a text frame key underneath showing these color values that made each color. Some of these files made a document over 100 pages, and consequently might take over an hour to finish.

Once I had these styles, instead of applying the individual font features to the labels, I applied a style. The result was that even the largest color files only took a number of minutes to complete. The smaller ones took only a few seconds. So here is a clear additional reason to use styles in scripts. I should add, too, that you don't have to create the styles with Python. You might create your styles on the main canvas, or import them from some document, then in your script simply use them.

PYTHON AND STRINGS

As you may notice in this book, there are a number of useful ways to work with strings in Python. In Scripter, **`valueDialog()`** and **`fileDialog()`** both return a string. The message you put into a **`messageBox()`** must be a string. If what you actually need from the user is an integer, you need to do something like this:

```
value = scribus.valueDialog('Input Needed', 'How many times to do this
            task?', '1')
value = int(value)
```

and in reverse, for a messageBox():

```
counter = 7

scribus.messageBox('Results', 'The current counter reading is ' +
          str(counter) + '\nGood Job!', button = scribus.BUTTON_OK)
```

With this message you can also see how using '**\n**' (this is a string denoting a newline), or maybe '**\t**' (for tab) can improve the appearance of the message.

In many cases we may need to manipulate our string input even though it will remain a string. There are a number of built-in string methods in Python, which I will make use of. One of these is **split**.

```
values = '20 34 55 61 23 7'
pages = values.split()
```

The variable name pages denotes a list of the individual pages, though these are still string values, and if we wanted to use them as integers, they would need to be converted. Sometimes, we either want a string to be in upper case or in lower case, so these are easily accomplished.

```
response = 'yes'
response.upper() # response is now 'YES'
Response = 'No'
Response.lower() # Response is now 'no'
```

A reason for doing this is to simplify some conditional argument, and not need to cover all possible entries. A recent script that I wrote began by taking lines from a file. Each line consisted of a name of a color, and then separated by tabs were 3 floating point numbers, so a line might look like this:

```
30-478 39,52 60,42 38,39
```

The name of the color here is 30-478, but after that we can see that these numbers use a comma as the decimal separator, which Python will have trouble managing as a number.

```
content = line.split('\t')
fields = len(content)
if (fields == 4):
    colors.append(content[0])
    Lval = content[fields -3]
    Lval = re.sub(r',' , '.',Lval)
    Lval = float(Lval)
    L.append(Lval)
    aval = content[fields -2]
    aval = re.sub(r',' , '.',aval)
    aval = float(aval)
```

```
        a.append(aval)
        bval = content[fields -1]
        bval = re.sub(r',' , '.',bval)
        bval = float(bval)
        b.append(bval)
```

The first line is using the **`split()`** method, and notice how we indicate that the separator is a tab with '**\t**'. Next we take the precaution in these files, where there might be thousands of colors, that the number of bits of data is 4 as it should be, so **`len(content)`** returns the number of elements in the list content. A reason why this file used a tab as a separator is that some of the color names had spaces in them, so this would have been a tricky matter to deal with a space separator. For the float values then we start with the third from the last element of fields, and then using the Python **re** package, we use its **sub** method to replace (**r**) a comma with a period. This may be hard to decipher, since inside the parentheses we have an r, then a comma inside single quotes, then a comma separator, then a period inside single quotes. Once we make this switch, we can then convert this string to a float value. I will also add that at the top of this script we needed a line **`import re`** for the Python regex package to work.

The characters in a string can also be accessed in part or individually, since you can manage them like a list. The command **`len(string)`** may help you select some particular set of characters by finding out its length.

```
    value = 'Some lengthy and arbitrary string'
    value[0] # this is the first character in the string: S
    value[-1] # the last character: g
    value[-4] # the fourth from the last character: r
    value[2:] # all the characters from the third to the end: me lengthy and
            arbitrary string
    value[:4} # all the characters from the beginning up to the fifth: Some
            (includes the space after the word)
```

While you can't do this directly with numbers, you can, as we did in the script **postnet.py** convert the number **correctnum** to a string, so that we could see whether the last digit was a '0'. But we could have also done this:

```
    remainder = correctnum%10
```

where the '**%**' is the mathematical operator for modulus, in other words we find the remainder after dividing by 10. If the integer **correctnum** is divisible by 10, then remainder is 0.

TROUBLESHOOTING

As the saying goes, "An ounce of prevention is worth a pound of cure", so it is that putting some extra work into the construction of your script will make troubleshooting less likely to be needed. Make liberal use of methods like **`try:`** and **`if:`** to test the running conditions of the script, but equally important is that when these fail, give a useful message to the user. This is where a **`messageBox()`** can go a long way to helping the situation.

```
if scribus.selectionCount() == 0:
    scribus.messageBox('Scribus -Usage Error',"There is no object
           selected.\nPlease select a text frame and try again.",
           scribus.ICON_WARNING, scribus.BUTTON_OK)
    sys.exit(2)
```

If the script just quit or failed for some reason, the user might not have any idea what was wrong, but here he gets a specific message. Sometimes you may want to do this:

```
if scribus.haveDoc():

else:
    scribus.messageBox('Usage Error', 'You need a Document open',
           icon=scribus.ICON_NONE, button1=scribus.BUTTON_OK)
    sys.exit(2)
```

but as we saw in **temp_graphing.py**, you might do this:

```
if scribus.haveDoc():

else:
    if scribus.newDocument(scribus.PAPER_LETTER, (20,20,20,20),
           scribus.LANDSCAPE, 1, scribus.UNIT_POINTS, scribus.PAGE_1, 0, 1):
      scribus.messageBox('Message', 'You did not have a document open, so
           \none has been createdfor you.\n\nNow you can run the script
           again.', scribus.ICON_NONE, scribus.BUTTON_OK)
```

where you not only identify the issue, you also do something to help resolve it. Often, when you can't figure out what's going on, you can temporarily insert a **`messageBox()`** to give you some feedback. Here is something I had in my initial versions of **temp_graphing.py**, just after the importing of 24 hour data from the file:

```
scribus.messageBox('Your Data','Your data has '+str(i)+' data points',
           scribus.BUTTON_OK)
```

so that already at this time I had a warning of a flaw in the data file, if there was some number of data points other than 24. Something else I noticed in this script was that, in spite of intentions, the graph tick marks did not fill up the Y axis. When I put in a **`messageBox()`** to inspect the scaling value, I realized that I needed to create this as a **float**() value, or it would be represented as an integer.

We all make typographical errors as we type, and these are very troublesome in scripts. If you are lucky, you may get a syntax error message, but often there is some feedback that just doesn't make sense. In many of these cases, noting the line number identified as a problem is your best clue. Go there and see if you've forgotten a closing parenthesis, or put in too many, or maybe just spelled something wrong. As I said in the beginning, Python is very picky about indentations, and expects all the lines in a given section to line up with each other. This is where using an editor that has features to help coding is very useful. I use Kwrite, sometimes Emacs, and both of these automatically will do syntax highlighting of keywords, and manage indentations well, but what triggers this is that each must know that the file that is being edited is a Python script, because its name ends with .py.

The last piece of advice I would give is to make your scripts in an incremental way. Especially with all the error-checking that is more or less routine, I often begin with some script that I know works contains all of that, and strip it down to nothing but the error-checking, then begin to assemble the new tasks one at a time, periodically running them in Scribus to make sure I haven't broken something. This is especially important when I am using commands that I haven't used before, or use rarely. Sometimes, when you're adding some new feature to an already sizeable script, you have to pull out the new feature into a new, small one to just work on that aspect. This also means that you should keep scripts that aren't so useful anymore, and also that all your scripts contain comments that help you understand them years later. Don't forget to try out some old script to make sure that it still works. Over the years, there have been various changes in the Scripter commands, especially with parameters that are required. Many constants have changed.

There are a number of things you might have in a script that are not specific to Scripter, and therefore might be most easily worked out in some Python script that you run outside of Scribus.

Ultimately, you have to have a mindset that keeps you persistent at scripting.