Scripter: Extension Scripts

Author: Craig Ringer

The Scribus Python plugin offers some additional features for extending Scribus with new capabilities and features, as opposed to automating tasks. In particular, it is possible to use 'extension scripts' to create new palettes and dockable windoes that can be used just like they were a part of Scribus.

Extension scripts

Extension scripts are mostly like normal Python scripts for Scribus. They're written somewhat differently so that they can be run with the "extension script" feature, giving them access to PyQt support and some other special features of the scripter. The most fundamental differences between normal scripts and extensions scripts are that:

- Extension scripts can create Python objects that continue to exist after the script exits. Objects will only continue to exist if a reference to them still exists, most commonly because there's a name in the global namespace that refers to them. This means that Python functions can be called by Scribus - for example, as PyQt slots, or as callback functions on an event.
- Extension scripts can create code that runs without blocking the execution of scribus, so you can create floating palettes that can be present while the user works normally with Scribus (ie non-modal palettes).
- PyQt works correctly in extension scripts, unlike normal scipts.
- Extension scripts can make changes to the Python environment that will affect scripts run after it. Modules imported by one script can be seen by another, settings changed by one may stay changed, etc. This means you have to be somewhat more careful when writing extension scripts. In particular, global names bound by one extension script can be overwritten by another, causing the objects associated with those names to be cleaned up by the interpreter. In other words, you can have name conflicts and interactions between scripts, which you can't with normal scripts.

The technical bit

Normal scripts get run in a new Python sub-interepter that's used just for that script then cleaned up. This means that whatever Python objects they create and whatever Python settings they change get automatically reset when the script exits. Because Scribus takes care of cleaning up your script, you don't have to worry about memory, conflicting with other scripts, etc and can just get on with the business of writing the script.

Extension scripts, by contrast, are run in a single Python interpreter that is started up when the script plugin loads. This interpreter keeps on running until the script plugin is unloaded when Scribus quits. When Scribus runs an extension script, it loads it into the running interpreter - much like `execfile` loads a Python script in another Python script. This lets extension scripts create new objects as they run, then exit to return control to Scribus without having the objects they created destroyed. If another script is then run, it can see all the objects created by the first script.

There are several situations where being able to create objects from Python that hang around after the script exits is useful. The most signficiant is graphical programming with PyQt, where PyQt objects are created when the script runs and become functional only after the script exits, returning control to the Scribus event loop. Another possible use is for macros, event callbacks, and timers, where Scribus would need to be able to call Python code. You can do all these with PyQt right now, but there is no direct support for timers and callbacks in Scribus yet.

There are some downsides to having objects persist after the script exits. Scripts can potentially interact in ways not thought of by their authors, which is often good but can also cause unexpected and surprising bugs. Script authors must also consider the effect of their code on the memory consumption of Scribus.

Building graphical add-ons in Python

Building new palettes and dialogs in PyQt is a fairly easy way to extend Scribus's user interface and provide extra facilities for advanced scripts. Python is well suited to getting data into and out of databases, content management systems, and other external repositories, and being able to build your own interfaces for this can be very helpful.

In most ways, PyQt works the same when running within Scribus as it does when used in a stand-alone Python interpreter. There are differences, however, and it is important to understand these.

- An instance of `QApplication` already exists, and attempting to create one will have undesired consequences. You can access the existing `QApplication` instance as `qt.qApp` if you need it.
- Scribus runs the Qt event loop. Entering the Qt event loop in PyQt will probably prevent further execution of Scribus until your code exists, and may have other undesired behaviour. The following documentation will cover the correct approach to integrating your code into the event loop. In brief, however, you simply create all your instances, show your windows, and let your script exit. Qt will automatically integrate your windows into its event loop and everything will "just work" - even Python slots and Python widgets. In general, anything you want to keep around should be put in the global namespace (as explained above).

The Basics - Converting Hello World

The first PyQt tutorial is the classic Hello World application. As an example of the differences between PyQt and Scribus, we'll convert that program to run in Scribus. Here's the original:

```
#!/usr/bin/env python
import sys
import qt
```

```
a = qt.QApplication(sys.argv)

hello = qt.QPushButton("Hello world!", None)
hello.resize(100, 30)

a.setMainWidget(hello)
hello.show()
sys.exit(a.exec_loop())
```

First, we need to disable the creation of `QApplication` since in Scribus a `QApplication` instance already exists, and only one is allowed per application. PyQt provides us with access to the `QApplication` that was created by Scribus when it starts up as `qt.qApp`. So, we simply replace:

```
a = qt.QApplication(sys.argv)
```

with

```
a = qt.qApp
```

and we're done with that change.

Next, we need to prevent the script from trying to run its own event loop. Because Scribus has an event loop, if the script starts its own it will disrupt Scribus until it exits. Qt is smart enough to hook any windows you create into the existing event loop, so there's not much to do. While your script is running, Scribus is under the control of Python, so what we need to do is do all our setup (in this case, create a simple window and show it) but then *let our script exit* instead of running the event loop. Because all extension scripts run in the same Python interpreter, the objects you create are not destroyed when your script exits. It's a bit like loading a module. When your script exists, Scribus takes control and resumes running the Qt event loop. Because your windows are Qt widgets, the Scribus event loop looks after them and they work like a normal part of Scribus. When a Python slot is triggered or a Python function is called, PyQt automatically takes care of running the Python function then returning to Scribus.

The only hitch with this scheme is that when your script terminates, all objects you create inside a function or other local scope will be cleaned up by Python as the scope is left (eg when leaving main()). This means that you need to keep a reference to everything at the global level so that it's not cleaned up. Support for PyQt in Scribus is very new, and there's no clear "right" way to do this yet. Options include:

- Creating everything you want to keep in the global namespace. Caution is required if your script is run multiple times.
- Storing objects you wish to keep in a dictionary or class in the global namespace. Most of the same problems exist with this as with storing the objects directly as global names.
- Putting your script in a module, then having the script the user runs simply import the module and run a function in it. This looks like it'll be the favoured approach. Note that the module body is not reloaded with every import, so you should generally put any code you want run each time in a function inside the module instead of the module top level. Alternately, you could check if the module is already loaded and reload() it instead of importing it anew.

For now, because this script already creates everything as a global we're going to do things that way. Larger scripts should be written as modules.

Given that the objects we need will already hang around when the script exits, all we need to do is prevent the script from entering the event loop. That's easy - just comment out the last line:

```
# sys.exit(a.exec_loop())
```

and we're nearly done. The script will run now, but when you close it it'll have a rather unintended effect - it'll close down Scribus. That's probably not what you want. This happens because a Qt application normally exits when its main widget ('main window') is closed. We call `qt.setMainWidget(...)` to make our new window the main widget, so when it's closed, Scribus goes with it. To prevent this, just comment out `qt.setMainWidget`.

The new script looks like:

```
#!/usr/bin/env python
import sys
import qt

a = qt.qApp

hello = qt.QPushButton("Hello world!", None)
hello.resize(100, 30)

#a.setMainWidget(hello)
hello.show()
#sys.exit(a.exec_loop())
```

You'll find that script already saved as `pyqt_tut1.py` in the scripter examples directory. Try running it as an extension script. You should get a hello world button. Notice that you can keep working in Scribus as normal while it's there, and that when you close the hello world window it politely goes away without affecting Scribus.

If you have a look at the sample copy of this tutorial script, you'll notice it has a few small additions. They are accompanied by some explanatory comments, so they won't be explored more here.

Fun with global names and shared interpreters

You may remember that I mentioned 'issues' with storing objects you want to keep as globals earlier? Unsurprisingly, I was dodging something I didn't want to have to explain right away.

Storing objects as global names works fine ... until the user runs your script again, or runs another script that uses the same names. Python uses reference counting - an object continues to exist while one or more names refer to it. When the global name you created earlier is overwritten by another script or by another run of your script, there are no more references to that object (which might be a window the user is still using). Python does its job and helpfully deletes it for you - it doesn't know it might still be being displayed, or might be a slot that one of your windows is using. In many cases this will simply result in a window unexpectedly vanishing, but it can have nastier consequences too.

Try this now. Run the hello world script (using "Load Extension Script...") and without closing the "Hello world" window, run the script again. The original window should vanish and be replaced with the new one.

There aren't really any good solutions to this yet, and the right behavour depends on what exactly you want to do. I'd like to have some clearer recommendations to give, but for that I need use cases. If you're running into this issue, please post a description of your project on the Scribus mailing list and I or someone else will try to give you a few suggestions.

The best solution so far is to use a simple wrapper script to run your script and put your real script in a module. The wrapper script imports your module then runs a function from the module to show the windows. Since the module is only run the first time it's imported, the window(s) will be displayed if they're not already visible, but won't be disrupted if they are still visible. You can reload() the module if you really want to re-run it, possibly after running some cleanup code.

Better suggestions would be much appreciated. Feel free to post questions and ideas on the mailing list.

Other tricks

Even if you're not building a custom graphical user interface, it's possible to make use of extension scripts. For example, you can use PyQt to run a function on a timer. One use might be to check for updates to an article in a database and perhaps ask the user if they wish to update their document with the new text (or see the differences). You can find a very simple exmaple of setting up a timer with PyQt in the examples directory, called `pyqt_timer.py`.

Another idea, as suggested by someone on the mailing list, was to write an XML-RPC server to expose the scripter API to external programs. This should be possible by using the PyQt classes for networking and event handling.

Other sources of information

This document isn't a PyQt or Qt tutorial. Good sources of information on Qt and PyQt are:

- The PyQt tutorial and examples from the PyQt documentation
- Graphical Programming with Python - Qt Edition
- TrollTech's Qt documentation (C++)
- The Independent Qt Tutorial
- QtForum.org

Cleanly handling being run outside Scribus

```
try:
    import scribus
except ImportError:
    print "This script can only be run as an extension script from Scribus"
    sys.exit(1)
```

This tries to load the Scribus scripter interface, and if it fails assumes it is not running under Scribus and complains. It's a good idea to do this in all your scripts so that you don't confuse users who try to run them with the standalone Python interpreter. Try running the script with `python pyqt_tut1.py` and note how it tells the user why it won't work then exits. This is much nicer than an import error or bizarre behaviour.

Unanswered questions and missing features

Support for extending Scribus from Python is still very much a work in progress. Lots works fine, but there are lots of unexplored corners. Feedback, suggestions, requests, ideas and offers of help would be much appreciated and can be directed to the mailing list or to the author(s) of this document.

Note that in particular there is no support for:

- Using PyQt from normal scripts (as opposed to extension scripts)
- Using PyGtk or wxPython

- Threading (PyQt threads might work within the limits of Qt's threading support)
- Hooking into the right-click menu (yet!)
- Triggering scripts on certain events (being considered)
- Easily and reliably hooking into the menus
- Extending Scribus dialogs
- Using Scribus custom widgets and classes
- Anything that requires you to hand over control to its event loop without returning (these will work, but block Scribus).

Some of these are just not done yet, some are extremely difficult, and some we don't know how to do at all or don't plan to attempt.