Red Hat Python Lab - Lesson 6: GUI, XML-RPC, The End.

Daniel Mach <dmach@redhat.com>
Martin Sivák <msivak@redhat.com>

Useful links

- Python Quick Reference: http://rgruet.free.fr/#QuickRef
- Python docs: http://www.python.org/doc/
- PEP 8: http://www.python.org/dev/peps/pep-0008/
- pep8.py: http://pypi.python.org/pypi/pep8/
- pylint: http://www.logilab.org/project/pylint
- Epydoc: http://epydoc.sourceforge.net/

Interesting links:

- Letajici cirkus on root.cz: http://www.root.cz/serialy/letajici-cirkus/
- py.cz: http://www.py.cz/
- Unladen swallow: http://code.google.com/p/unladen-swallow/

Don't forget

- Batteries included -> read docs, find existing function, use it
- Follow PEP 8
- Pylint is your friend
- Write unit tests
- Objects are not everything, don't make your code over-complicated
- Return from functions as soon as possible -> more readable code, better indentation

Gui programming with Python

As python is a language with strong introspection and has support for dynamically created behaviour, GUI programming is very easy.

There are many frameworks supporting abovementioned python features. Python itself contains bindings for Tk (the module is named Tkinter). But today, the most important frameworks are Gtk and Qt. We will concentrate on Gtk (or it's pythonized brother PyGTK), but very similar concepts should work in Qt, wxWidgets or Tk.

Gtk has moved to new API with version 3 (completely based on introspection features), but we will use version 2 to make it work on older systems. In version 3.x there is no pygtk module and everything is done using gi module (glib introspection).

Hello world in PyGtk

```
# Gtk2 pygtk version
import pygtk
pygtk.require('2.0')
import gtk

# Gtk3 version
# from gi.repository import gtk

class HelloWorld:
  def destroy(self, widget, data=None):
    gtk.main_quit()

  def __init__(self):
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.connect("destroy", self.destroy)

    self.label = gtk.Label("Hello World")
    self.label.show()
    self.window.add(self.label)
    self.window.show()
    gtk.main()

if __name__ == "__main__":
  HelloWorld()
```

PyGtk documentation

- Tutorial: http://www.pygtk.org/pygtk2tutorial/
- Reference manual: http://library.gnome.org/devel/pygtk/stable/

GTK project also has some useful abstractions over threads, streams and others. You can find the reference manual to Python GObject library at http://library.gnome.org/devel/pygobject/stable/

- Reference manual: http://developer.gnome.org/gtk3/3.4/

Gtk Builder

To avoid building everything manually and hardcoding the GUI layout into code, we have tools available, which will load the design from XML description and connect it to our application's logic.

- libglade - the older way, supports only widgets
- GtkBuilder - newer incarnation of glade, supports data model objects, accels and more..

Both these ways are very simmilar to use, consider following examples of importing sample layout and showing the window:

```
# Gtk2 - old glade
import gtk
import gtk.glade
gl = gtk.glade.XML("x.glade")
gl.get_widget("main_window").show()
gtk.main()

# Gtk2 - new glade
gtkb = gtk.Builder()
gtkb.add_from_file("x.xml")
gtkb.get_object("main_window").show()
gtk.main()

# Gtk3
from gi.repository import Gtk
gtkb = Gtk.Builder()
gtkb.add_from_file("x.xml")
gtkb.get_object("main_window").show()
Gtk.main()
```

Connecting actions in GUI to methods

When you have all the elements you need, it is time to assign actions to buttons and other pieces which may react to user's actions. There are (as usual) couple of ways to do it:

- manually - call widget.connect("signal-name", method_to_start)
- automatically - define method names in the glade file and then call *gtkb.connect_signals(some_object)* (gtkb is the gtk.Builder instance from previous slide). some_object is either dictionary or an actual object containing keys/methods with names corresponding to the definitions in loaded glade file. (Glade users will use *gl.signal_autoconnect(some_object)*).

```
class GUI(object):
  def __init__(self):
    self.gtkb = gtk.Builder()
    self.gtkb.add_from_file("x.xml")
    self.gtkb.connect_signals(self)
    self.mainwindow = self.gtkb.get_object("main_window")
    self.mainwindow.show()
    gtk.main()

  def on_window_destroy(self, w, data = None):
    gtk.main_quit()
```

Using Gtk in threaded app

Gtk is not thread safe, so to use it from multithreaded application, some precautions have to be made:

- Gtk will run in it's thread (the thread where you executed gtk.main)
- All gtk calls will be done in that thread using callbacks

```
# use gobject instead of Glib in Gtk2
# wait variant @ http://git.fedorahosted.org/cgit/anaconda.git/tree/pyanaconda/ui/gui/utils.py
# decorator documentation http://www.python.org/dev/peps/pep-0318/

def gtk_thread_nowait(func):
    """Decorator method which causes every call of the decorated function
       to be executed in the context of Gtk main loop. The new method does
       not wait for the callback to finish. """

    def _idle_method(args):
        """This method contains the code for the main loop to execute.
        """
        ret = func(*args)
        return False           # has to return False so it is executed only once

    def _call_method(*args):
        """The new body for the decorated method. """
        GLib.idle_add(_idle_method, args)

    return _call_method
```

Usage:

```
@gtk_thread_nowait
def my_gtk_touching_method(...):
  do gtk calls and return as soon as possible

# will be executed inside the Gtk mainloop
my_gtk_touching_method()
```

XML-RPC Server

Python has a XML-RPC Server implementation, all you need to do is to add threading support by using ThreadingMixin.

```
import SocketServer
import SimpleXMLRPCServer


class SimpleThreadedXMLRPCServer(SocketServer.ThreadingMixIn, SimpleXMLRPCServer.SimpleXMLRPCServer):
    pass
```

You can do the same thing using SocketServer and get a basic network server almost for free: http://docs.python.org/library/socketserver.html.

```
def foo_handler():
    return "foo"


class HandlerClass(object):
    def bar_handler(self):
        return "bar"


server = SimpleThreadedXMLRPCServer((address, port), allow_none=True)

# register functions to the server
server.register_introspection_functions()
server.register_instance(HandlerClass())
server.register_function(foo_handler)

# start server
server.serve_forever()
# server.shutdown() stops serving, but has to be started from a different thread (serve_forever blocks current thread)
```

XML-RPC Client

XML-RPC Client is really easy to use.

```
import xmlrpclib


client = xmlrpclib.ServerProxy("http://localhost:8001", allow_none=True)
print client.foo_handler()
print client.bar_handler()
```