Writing GTK 2.0 and GNOME 2.0 Applications with Python

James Henstridge
james@daa.com.au

Overview

- Why you should use PyGTK.
- Differences compared to PyGTK 0.6.x.
- New features in PyGTK.
- What is left to do for PyGTK 2.0?

Why Python?

- A simple, readable language
- Comprehensive standard library
- Many extensions -- a good "glue" language
- Garbage collection
- Much nicer to program than Perl

Why use PyGTK for Gnome apps?

- Write shorter programs
- Object oriented API
- No need to worry about casting or reference counting
- Can be used for prototyping C applications.
- Rapid application development (with glade)
- Gets rid of the type safety problems with C varargs APIs

A Simple PyGTK Program

```
import gtk

def hello_cb(button):
    print "Hello World"
    print "Button's label is: ", button.get_property("label")

def destroy_window(window):
    gtk.main_quit()

win = gtk.Window()
win.connect("destroy", destroy_window)
win.set_border_width(10)

button = gtk.Button("Click me")
button.connect("clicked", hello_cb)
button.foo = 'bar'

win.add(button)
win.show_all()

gtk.main()
```

A Simple PyGTK Program (continued)

- Simple obvious mapping from C API to object oriented API.
- Can connect any callable python object as a signal handler
- Widgets act as normal python class instances
  - can set/get attributes on the widget objects.
  - can even subclass widgets (see next slide)
- No casting

C to Python Mapping

- Most of the GTK C API maps to Python fairly easily:
  - gtk_foo_* (functions) -> gtk.Foo.* (methods)
  - gtk_foo_new -> gtk.Foo.__init__ (constructor)
  - gtk_* -> gtk.* (functions)
- Similar for Pango, ATK and GDK APIs:
  - pango_* -> pango module
  - atk_* -> atk module
  - gdk_* -> gtk.gdk module
  - glade_* (libglade) -> gtk.glade module
  - Pango functions in pango module
  - ATK functions in atk module
  - GDK functions in gtk.gdk module
  - libglade functions in gtk.glade module

A More Python Like Version

```
import gtk

class MyWindow(gtk.Window):
    def __init__(self, button_msg="Click me"):
        gtk.Window.__init__(self)
        self.set_border_width(10)

        self.button = gtk.Button(button_msg)
        self.add(self.button)
        self.show()

        self.connect("destroy", self.destroy_cb)
        self.button.connect("clicked", self.button_cb)
    def destroy_cb(self, window):
        gtk.main_quit()
    def button_cb(self, button):
        print "Hello World"

win = MyWindow()
win.show()

gtk.main()
```

Building Interfaces with Libglade

- Write interface in glade
- Write program logic in Python
- Use libglade to hook up program logic to GUI:

```
import gtk.glade

xml = gtk.glade.XML('filename.glade')
xml.signal_autoconnect({'funcname': func, ...})
gtk.main()
```

- See Christian Egli's talk for more details

Incompatibilities with Stable PyGTK

- No procedural API
- All types are implemented as Python 2.2 new style types.
- Gtk and Gdk prefixes removed from types (see next slide)
- Threading support not quite working

Mapping From Old to New Names

- gtk.Gtk* -> gtk.*
- gtk.Gdk* -> gtk.gdk.*
- GTK.* -> gtk.* (constants)
- GDK.* -> gtk.gdk.* (constants)
- libglade.GladeXML -> gtk.glade.XML

New Features

- Support for new text widget
  - Based on Tk text widget, so provides a useful upgrade path from Tkinter
  - Doesn’t suck like GtkText
- New Tree widget
  - Full support for use of GtkTreeView, GtkTreeStore and GtkListStore
  - Limited support for defining new tree models.
- Unicode support
  - Text handling in GTK now handled by Pango. UTF-8 used as internal encoding.
  - Python unicode strings get automatically recoded to UTF-8

New Features (continued)

- Single wrapper per GObject.
  - Setting attributes on a python wrapper will actually work as expected.
- New defs format used to generate bindings
  - Common format used by many gtk 2.0 bindings, including C++, Guile
  - Provides more information
- start of reference documentation
  - currently uses structure from defs files, and content from C reference docs.
- Can define new types in Python
  - create and override signals
  - create object properties

Creating a New Type

```
import gobject, gtk

class MyWidget(gtk.Widget):
    def __init__(self):
        # Call __gobject_init__ instead of parent
        # constructor.  This ensures that the
        # correct type of GObject gets created.
        self.__gobject_init__()


    ...

gobject.type_register(MyWidget)
```

Adding Signals

```
import gobject, gtk

class MyWidget(gtk.Widget):
    __gsignals__ = {
        'mysignal': (gobject.SIGNAL_RUN_FIRST,
                     None, (int,))
    }

    ...

    def do_mysignal(self, intarg):
        print 'Integer argument:', intarg

gobject.type_register(MyWidget)

w = MyWidget()
w.connect('mysignal', function)
w.emit('mysignal', 42)
```

Overriding Signals

```
import gobject, gtk

class MyWidget(gtk.Widget):
    __gsignals__ = {
        'expose_event': 'override'
    }

    ...

    def do_expose_event(self, event):
        print "got expose at (%d, %d) %dx%d" % \
            (event.x,event.y, event.width,event.height)
        # call parent implementation of expose_event
        return self.chain(event)

gobject.type_register(MyWidget)

w = MyWidget()

...
```

Properties

```
import gobject

class MyObject(gobject.GObject):
    __gproperties__ = {
        'spam': (str, 'spam property', 'a string', 'default',
                 gobject.PARAM_READWRITE)
    }
    def __init__(self):
        self.__gobject_init__()
        self.__spam = 'default'
    def do_set_property(self, pspec, value):
        if pspec.name == 'spam':
            self.__spam = value
        else:
            raise AttributeError, pspec.name
    def do_get_property(self, pspec):
        if pspec.name == 'spam':
            return self.__spam
        else:
            raise AttributeError, pspec.name
gobject.type_register(MyObject)

obj = MyObject()
obj.set_property('spam', 'eggs')
```

Generators

- Rewriting tasks to be run by the main loop as idle tasks is tedious and error prone.
  - Use generators!

```
from __future__ import generators
import gtk

def task_generator():
    for i in range(100):
        print i
        yield gtk.TRUE # yield control to main loop
    yield gtk.FALSE    # task complete

gtk.idle_add(task_generator().next)
```

- State saved automatically between invocations of idle function.

What Still Needs To Be Done For 2.0 Release

- Finish off gtk bindings
  - Some infrastructure issues need to be solved (threading, etc)
  - Check coverage
  - Parity with stable
- GNOME library bindings
  - ORBit2 bindings (Johan Dahlin)
  - Wrap more GNOME libraries