PyGObject

Release

Sep 08, 2017

Contents

To get things started we will try to run a very simple GTK+ based GUI application using the *PyGObject* provided Python bindings. First create a small Python script called `hello.py` with the following content and save it somewhere:

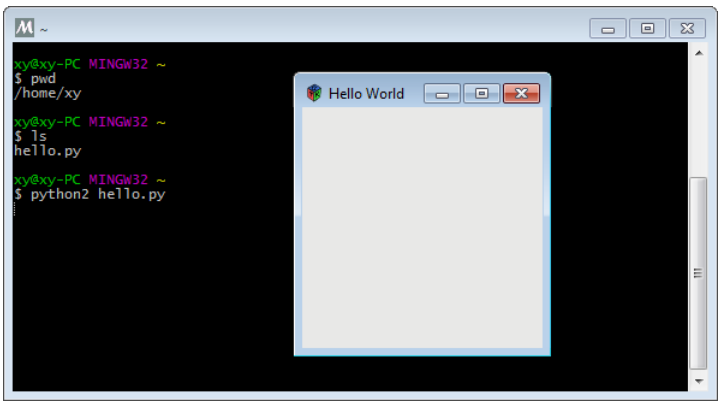
```
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

window = Gtk.Window(title="Hello World")
window.show()
window.connect("delete-event", Gtk.main_quit)
Gtk.main()
```

Before we can run the example application we need to install PyGObject, GTK+ and their dependencies. Follow the instructions for your platform below.

CHAPTER 1

Windows

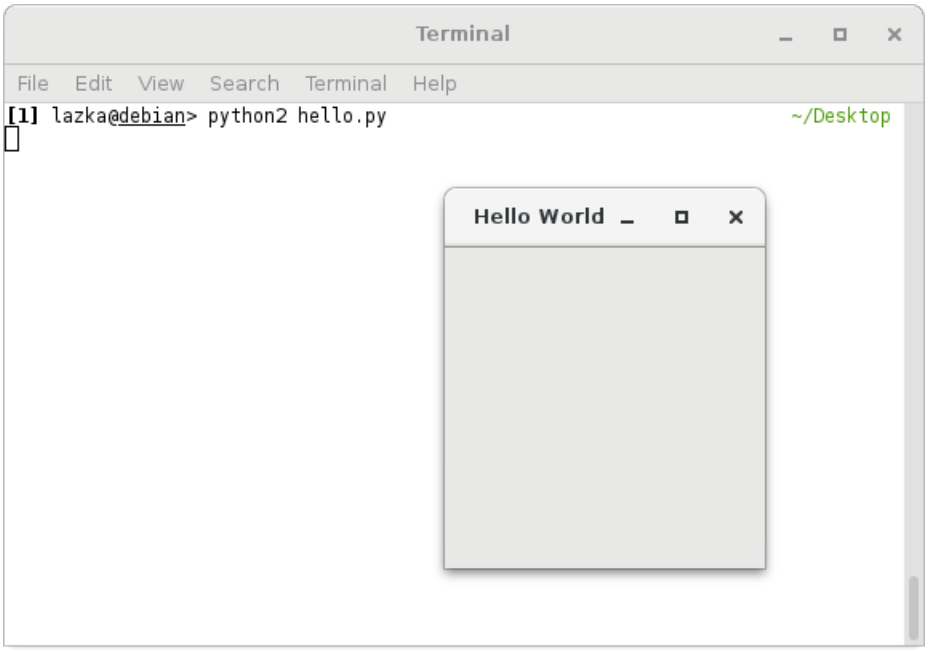
1. Go to https://msys2.github.io/ and download the x86_64 installer
2. Follow the instructions on the page for setting up the basic environment
3. Run `C:\msys64\mingw32.exe` - a terminal window should pop up
4. Execute `pacman -S mingw-w64-i686-gtk3 mingw-w64-i686-python2-gobject mingw-w64-i686-python3-gobject`
5. To test that GTK+3 is working you can run `gtk3-demo`
6. Copy the `hello.py` script you created to `C:\msys64\home\<username>`
7. In the mingw32 terminal execute `python2 hello.py` - a window should appear.



CHAPTER 2

Ubuntu / Debian

1. Open a terminal
2. Execute `sudo apt install python-gi python-gi-cairo python3-gi python3-gi-cairo gir1.2-gtk-3.0`
3. Change the directory to where your `hello.py` script can be found (e.g. `cd Desktop`)
4. Run `python2 hello.py`



CHAPTER 3

Fedora

1. Open a terminal
2. Execute `sudo dnf install pygobject3 python3-gobject gtk3`
3. Change the directory to where your `hello.py` script can be found (e.g. `cd Desktop`)
4. Run `python2 hello.py`

CHAPTER 4

Arch Linux

1. Open a terminal
2. Execute `sudo pacman -S python-gobject python2-gobject gtk3`
3. Change the directory to where your `hello.py` script can be found (e.g. `cd Desktop`)
4. Run `python2 hello.py`

CHAPTER 5

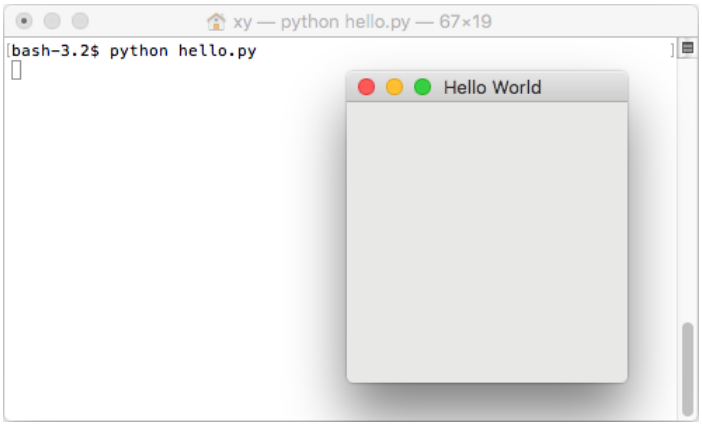
openSUSE

1. Open a terminal
2. Execute `sudo zypper install python-gobject python3-gobject gtk3`
3. Change the directory to where your `hello.py` script can be found (e.g. `cd Desktop`)
4. Run `python2 hello.py`

CHAPTER 6

macOS

1. Go to https://brew.sh/ and install homebrew
2. Open a terminal
3. Execute `brew install pygobject3 --with-python3 gtk+3` to install for both python2 and python3
4. Change the directory to where your `hello.py` script can be found (e.g. `cd Desktop`)
5. Run `python2 hello.py`



CHAPTER 7

User Guide

GI API

This is the API provided by the toplevel "gi" package.

gi.**require_version**(*namespace*, *version*)

Parameters

- **namespace** (*str*) – The namespace
- **version** (*str*) – The version of the namespace which should be loaded

Raises ValueError

Ensures the namespace gets loaded with the given version. If the namespace was already loaded with a different version or a different version was required previously raises ValueError.

```
import gi
gi.require_version('Gtk', '3.0')
```

gi.**require_foreign**(*namespace*, *symbol=None*)

Parameters

- **namespace** (*str*) – Introspection namespace of the foreign module (e.g. "cairo")
- **symbol** (str or None) – Optional symbol typename to ensure a converter exists.

Raises ImportError

Ensure the given foreign marshaling module is available and loaded.

Example:

```
import gi
import cairo
gi.require_foreign('cairo')
gi.require_foreign('cairo', 'Surface')
```

gi.**check_version**(*version*)

Parameters **version** (*tuple*) – A version tuple

Raises ValueError

Compares the passed in version tuple with the gi version and does nothing if gi version is the same or newer. Otherwise raises ValueError.

gi.**get_required_version**(*namespace*)

Returns The version successfully required previously by *gi.require_version()* or None

Return type str or None

gi.**version_info** = **(3, 18, 1)**
The version of PyGObject

class gi.**PyGIDeprecationWarning**
The warning class used for deprecations in PyGObject and the included Python overrides. It inherits from DeprecationWarning and is hidden by default.

class gi.**PyGIWarning**
Like *gi.PyGIDeprecationWarning* but visible by default.

Basic Types

PyGObject will automatically convert between C types and Python types. In cases where it's appropriate it will use default Python types like int, list, and dict.

Number Types

All glib integer types get mapped to int, long and float. Since the glib integer types are always range limited, conversions from Python int/long can fail with OverflowError:

```
>>> GLib.random_int_range(0, 2**31-1)
1684142898
>>> GLib.random_int_range(0, 2**31)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: 2147483648 not in range -2147483648 to 2147483647
>>>
```

Text Types

In case you use Python 2 then text is utf-8 encoded str, in case of Python 3 str is used.

Platform String Types

- Windows + Python 2: utf-8 encoded str
- Windows + Python 3: str
- Unix + Python 2: str
- Unix + Python 3: str

On Python 3 there is currently no support for `bytes`, see bug 746564 for more details.

Other Types

- GList <-> `list`
- GSList <-> `list`
- GHashTable <-> `dict`
- arrays <-> `list`

Flags & Enums

Flags are subclasses of `GObject.GFlags` and represent bit fields where some bits also have names:

```
>>> Gtk.DialogFlags.MODAL
<flags GTK_DIALOG_MODAL of type Gtk.DialogFlags>
>>> Gtk.DialogFlags.MODAL | Gtk.DialogFlags.DESTROY_WITH_PARENT
<flags GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT of type Gtk.DialogFlags>
>>> int(_)
3
>>> Gtk.DialogFlags(3)
<flags GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT of type Gtk.DialogFlags>
>>> isinstance(Gtk.DialogFlags.MODAL, Gtk.DialogFlags)
True
>>>
```

Bitwise operations on them will produce a value of the same type.

Enums are subclasses of `GObject.GEnum` and represent a list of named constants:

```
>>> Gtk.Align.CENTER
<enum GTK_ALIGN_CENTER of type Gtk.Align>
>>> int(Gtk.Align.CENTER)
3
>>> int(Gtk.Align.END)
2
>>> Gtk.Align(1)
<enum GTK_ALIGN_START of type Gtk.Align>
>>> isinstance(Gtk.Align.CENTER, Gtk.Align)
True
```

GObject.Object

Compare to other types, `GObject.Object` has the best integration between the GObject and Python type system.

1. It is possible to subclass a `GObject.Object`. Subclassing creates a new `GObject.GType` which is connected to the new Python type. This means you can use it with API which takes `GObject.GType`.
2. The Python wrapper instance for a `GObject.Object` is always the same. For the same C instance you will always get the same Python instance.

In addition `GObject.Object` has support for *signals* and *properties*

Signals

GObject signals are a system for registering callbacks for specific events.

To find all signals of a class you can use the GObject.signal_list_names() function:

```
>>> GObject.signal_list_names(Gio.Application)
('activate', 'startup', 'shutdown', 'open', 'command-line', 'handle-local-options')
>>>
```

To connect to a signal, use GObject.Object.connect():

```
>>> app = Gio.Application()
>>> def on_activate(instance):
...     print("Activated:", instance)
...
>>> app.connect("activate", on_activate)
17L
>>> app.run()
('Activated:', <Gio.Application object at 0x7f1bbb304320 (GApplication at␣
↪0x5630f1faf200)>)
0
>>>
```

It returns number which identifies the connection during its lifetime and which can be used to modify the connection.

For example it can be used to temporarily ignore signal emissions using GObject.Object.handler_block():

```
>>> app = Gio.Application(application_id="foo.bar")
>>> def on_change(*args):
...     print(args)
...
>>> c = app.connect("notify::application-id", on_change)
>>> app.props.application_id = "foo.bar"
(<Gio.Application object at 0x7f1bbb304550 (GApplication at 0x5630f1faf2b0)>,
↪<GParamString 'application-id'>)
>>> with app.handler_block(c):
...     app.props.application_id = "no.change"
...
>>> app.props.application_id = "change.again"
(<Gio.Application object at 0x7f1bbb304550 (GApplication at 0x5630f1faf2b0)>,
↪<GParamString 'application-id'>)
>>>
```

You can define your own signals using the *GObject.Signal* decorator:

GObject.**Signal**(*name=''*, *flags=GObject.SignalFlags.RUN_FIRST*, *return_type=None*, *arg_types=None*, *accumulator=None*, *accu_data=None*)

Parameters

- **name** (*str*) – The signal name
- **flags** (*GObject.SignalFlags*) – Signal flags
- **return_type** (*GObject.GType*) – Return type
- **arg_types** (*list*) – List of GObject.GType argument types
- **accumulator** (*GObject.SignalAccumulator*) – Accumulator function
- **accu_data** (*object*) – User data for the accumulator

```
class MyClass(GObject.Object):

    @GObject.Signal(flags=GObject.SignalFlags.RUN_LAST, return_type=bool,
                    arg_types=(object,),
                    accumulator=GObject.signal_accumulator_true_handled)
    def test(self, *args):
        print("Handler", args)

    @GObject.Signal
    def noarg_signal(self):
        print("noarg_signal")

instance = MyClass()

def test_callback(inst, obj):
    print "Handled", inst, obj
    return True

instance.connect("test", test_callback)
instance.emit("test", object())

instance.emit("noarg_signal")
```

Properties

Properties are part of a class and are defined through a `GObject.ParamSpec`, which contains the type, name, value range and so on.

To find all the registered properties of a class you can use the `GObject.Object.list_properties()` class method.

```
>>> Gio.Application.list_properties()
[<GParamString 'application-id'>, <GParamFlags 'flags'>, <GParamString
'resource-base-path'>, <GParamBoolean 'is-registered'>, <GParamBoolean
'is-remote'>, <GParamUInt 'inactivity-timeout'>, <GParamObject
'action-group'>, <GParamBoolean 'is-busy'>]
>>> param = Gio.Application.list_properties()[0]
>>> param.name
'application-id'
>>> param.owner_type
<GType GApplication (94881584893168)>
>>> param.value_type
<GType gchararray (64)>
>>>
```

The `GObject.Object` contructor takes multiple properties as keyword arguments. Property names usually contain "-" for seperating words. In Python you can either use "-" or "_". In this case variable names don't allow "-", so we use "_".

```
>>> app = Gio.Application(application_id="foo.bar")
```

To get and set the property value see `GObject.Object.get_property()` and `GObject.Object.set_property()`.

```
>>> app = Gio.Application(application_id="foo.bar")
>>> app
```

```
<Gio.Application object at 0x7f7499284fa0 (GApplication at 0x564b571e7c00)>
>>> app.get_property("application_id")
'foo.bar'
>>> app.set_property("application_id", "a.b")
>>> app.get_property("application-id")
'a.b'
>>>
```

Each instance also has a `props` attribute which exposes all properties as instance attributes:

```
>>> from gi.repository import Gtk
>>> button = Gtk.Button(label="foo")
>>> button.props.label
'foo'
>>> button.props.label = "bar"
>>> button.get_label()
'bar'
>>>
```

To track changes of properties, `GObject.Object` has a special `notify` signal with the property name as the detail string. Note that in this case you have to give the real property name and replacing "-" with "_" wont work.

```
>>> app = Gio.Application(application_id="foo.bar")
>>> def my_func(instance, param):
...     print("New value %r" % instance.get_property(param.name))
...
>>> app.connect("notify::application-id", my_func)
11L
>>> app.set_property("application-id", "something.different")
New value 'something.different'
>>>
```

You can define your own properties using the *GObject.Property* decorator, which can be used similarly to the builtin Python `property` decorator:

GObject.**Property**(*type=None*, *default=None*, *nick=''*, *blurb=''*, *flags=GObject.ParamFlags.READWRITE*, *minimum=None*, *maximum=None*)

Parameters

- **type** (*GObject.GType*) – Either a GType, a type with a GType or a Python type which maps to a default GType
- **default** (*object*) – A default value
- **nick** (*str*) – Property nickname
- **block** (*str*) – Short description
- **flags** (*GObject.ParamFlags*) – Property configuration flags
- **minimum** (*object*) – Minimum value, depends on the type
- **maximum** (*object*) – Maximum value, depends on the type

```
class AnotherObject(GObject.Object):
    value = 0

    @GObject.Property
    def prop_pyobj(self):
        """Read only property."""
```

```
        return object()

    @GObject.Property(type=int)
    def prop_gint(self):
        """Read-write integer property."""

        return self.value

    @prop_gint.setter
    def prop_gint(self, value):
        self.value = value
```

Examples

Subclassing:

```
>>> from gi.repository import GObject
>>> class A(GObject.Object):
...     pass
...
>>> A()
<__main__.A object at 0x7f9113fc3280 (__main__+A at 0x559d9861acc0)>
>>> A.__gtype__
<GType __main__+A (94135355573712)>
>>> A.__gtype__.name
'__main__+A'
>>>
```

In case you want to specify the GType name we have to provide a `__gtype_name__`:

```
>>> from gi.repository import GObject
>>> class B(GObject.Object):
...     __gtype_name__ = "MyName"
...
>>> B.__gtype__
<GType MyName (94830143629776)>
>>>
```

`GObject.Object` only supports single inheritance, this means you can only subclass one `GObject.Object`, but multiple Python classes:

```
>>> from gi.repository import GObject
>>> class MixinA(object):
...     pass
...
>>> class MixinB(object):
...     pass
...
>>> class MyClass(GObject.Object, MixinA, MixinB):
...     pass
...
>>> instance = MyClass()
```

Here we can see how we create a `Gio.ListStore` for our new subclass and that we get back the same Python instance we put into it:

```
>>> from gi.repository import GObject, Gio
>>> class A(GObject.Object):
...     pass
...
>>> store = Gio.ListStore.new(A)
>>> instance = A()
>>> store.append(instance)
>>> store.get_item(0) is instance
True
>>>
```

Cairo Integration

Despite cairo not being a GObject based library, PyGObject provides special cairo integration through pycairo. Functions returning and taking cairo data types get automatically converted to pycairo objects and vice versa.

Some distros ship the PyGObject cairo support in a separate package. If you've followed the instructions on "*Getting Started*" you should have everything installed.

If your application requires the cairo integration you can use *gi.require_foreign()*:

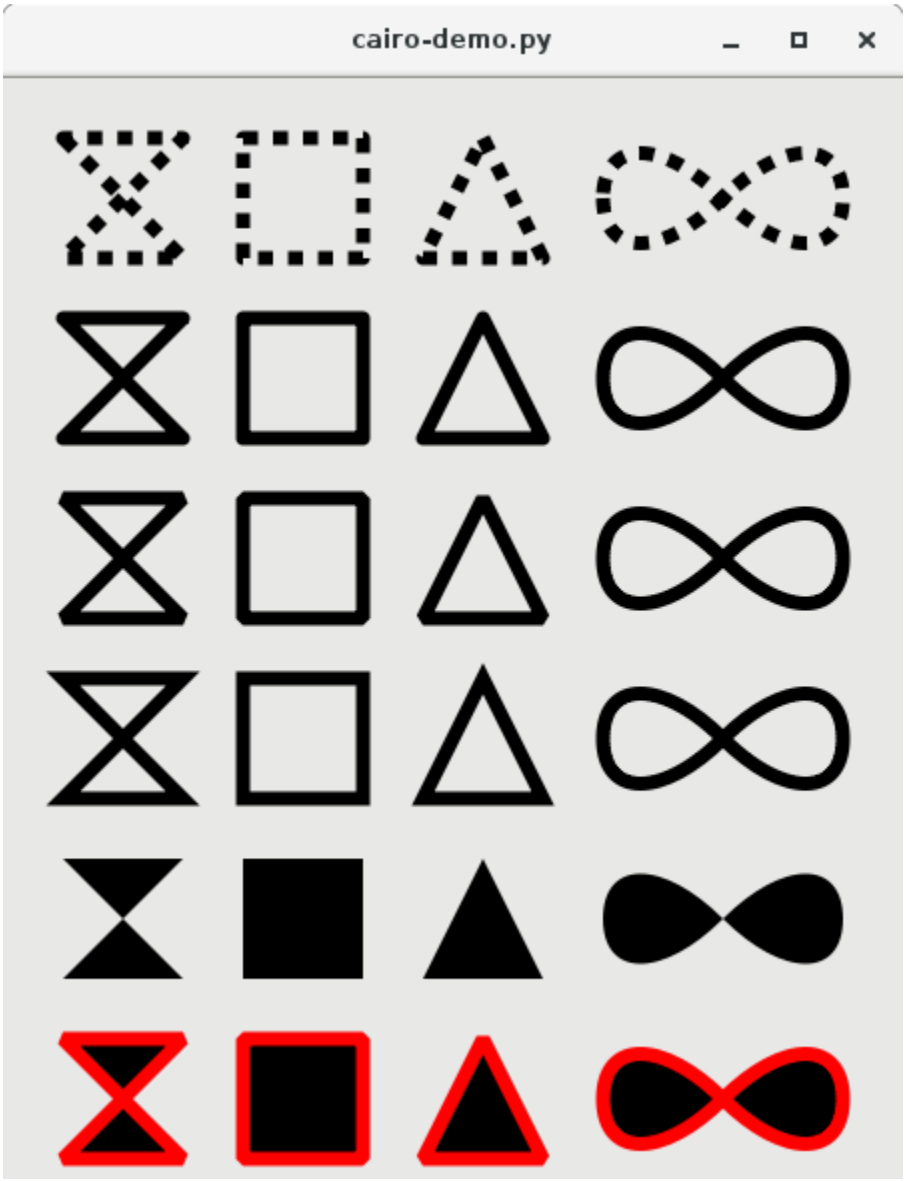
```
try:
    gi.require_foreign("cairo")
except ImportError:
    print("No pycairo integration :(")
```

Note that PyGObject currently does not support cairocffi, only pycairo.

Demo

The following example shows a Gtk.Window with a custom drawing in Python using pycairo.

```
#!/usr/bin/env python
"""
Based on cairo-demo/X11/cairo-demo.c
"""

import cairo
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

SIZE = 30


def triangle(ctx):
    ctx.move_to(SIZE, 0)
    ctx.rel_line_to(SIZE, 2 * SIZE)
    ctx.rel_line_to(-2 * SIZE, 0)
    ctx.close_path()


def square(ctx):
    ctx.move_to(0, 0)
    ctx.rel_line_to(2 * SIZE, 0)
```


cairo-demo.py

```
    ctx.rel_line_to(0, 2 * SIZE)
    ctx.rel_line_to(-2 * SIZE, 0)
    ctx.close_path()


def bowtie(ctx):
    ctx.move_to(0, 0)
    ctx.rel_line_to(2 * SIZE, 2 * SIZE)
    ctx.rel_line_to(-2 * SIZE, 0)
    ctx.rel_line_to(2 * SIZE, -2 * SIZE)
    ctx.close_path()


def inf(ctx):
    ctx.move_to(0, SIZE)
    ctx.rel_curve_to(0, SIZE, SIZE, SIZE, 2 * SIZE, 0)
    ctx.rel_curve_to(SIZE, -SIZE, 2 * SIZE, -SIZE, 2 * SIZE, 0)
    ctx.rel_curve_to(0, SIZE, -SIZE, SIZE, - 2 * SIZE, 0)
    ctx.rel_curve_to(-SIZE, -SIZE, - 2 * SIZE, -SIZE, - 2 * SIZE, 0)
    ctx.close_path()


def draw_shapes(ctx, x, y, fill):
    ctx.save()

    ctx.new_path()
    ctx.translate(x + SIZE, y + SIZE)
    bowtie(ctx)
    if fill:
        ctx.fill()
    else:
        ctx.stroke()

    ctx.new_path()
    ctx.translate(3 * SIZE, 0)
    square(ctx)
    if fill:
        ctx.fill()
    else:
        ctx.stroke()

    ctx.new_path()
    ctx.translate(3 * SIZE, 0)
    triangle(ctx)
    if fill:
        ctx.fill()
    else:
        ctx.stroke()

    ctx.new_path()
    ctx.translate(3 * SIZE, 0)
    inf(ctx)
    if fill:
        ctx.fill()
    else:
        ctx.stroke()

    ctx.restore()
```

```
82
83
84  def fill_shapes(ctx, x, y):
85      draw_shapes(ctx, x, y, True)
86
87
88  def stroke_shapes(ctx, x, y):
89      draw_shapes(ctx, x, y, False)
90
91
92  def draw(da, ctx):
93      ctx.set_source_rgb(0, 0, 0)
94
95      ctx.set_line_width(SIZE / 4)
96      ctx.set_tolerance(0.1)
97
98      ctx.set_line_join(cairo.LINE_JOIN_ROUND)
99      ctx.set_dash([SIZE / 4.0, SIZE / 4.0], 0)
100     stroke_shapes(ctx, 0, 0)
101
102     ctx.set_dash([], 0)
103     stroke_shapes(ctx, 0, 3 * SIZE)
104
105     ctx.set_line_join(cairo.LINE_JOIN_BEVEL)
106     stroke_shapes(ctx, 0, 6 * SIZE)
107
108     ctx.set_line_join(cairo.LINE_JOIN_MITER)
109     stroke_shapes(ctx, 0, 9 * SIZE)
110
111     fill_shapes(ctx, 0, 12 * SIZE)
112
113     ctx.set_line_join(cairo.LINE_JOIN_BEVEL)
114     fill_shapes(ctx, 0, 15 * SIZE)
115     ctx.set_source_rgb(1, 0, 0)
116     stroke_shapes(ctx, 0, 15 * SIZE)
117
118
119 def main():
120     win = Gtk.Window()
121     win.connect('destroy', lambda w: Gtk.main_quit())
122     win.set_default_size(450, 550)
123
124     drawingarea = Gtk.DrawingArea()
125     win.add(drawingarea)
126     drawingarea.connect('draw', draw)
127
128     win.show_all()
129     Gtk.main()
130
131
132 if __name__ == '__main__':
133     main()
```

Threads & Concurrency

Operations which could potentially block should not be executed in the main loop. The main loop is in charge of input processing and drawing and blocking it results in the user interface freezing. For the user this means not getting any feedback and not being able to pause or abort the operation which causes the problem.

Such an operation might be:

- Loading external resources like an image file on the web
- Searching the local file system
- Writing, reading and copying files
- Calculations where the runtime depends on some external factor

The following examples show

- how Python threads, running in parallel to GTK+, can interact with the UI
- how to use and control asynchronous I/O operations in glib

Threads

The first example uses a Python thread to execute code in the background while still showing feedback on the progress in a window.

```
import threading
import time

from gi.repository import GLib, Gtk, GObject


def app_main():
    win = Gtk.Window(default_height=50, default_width=300)
    win.connect("delete-event", Gtk.main_quit)

    progress = Gtk.ProgressBar(show_text=True)
    win.add(progress)

    def update_progess(i):
        progress.pulse()
        progress.set_text(str(i))
        return False

    def example_target():
        for i in range(50):
            GLib.idle_add(update_progess, i)
            time.sleep(0.2)

    win.show_all()

    thread = threading.Thread(target=example_target)
    thread.daemon = True
    thread.start()


if __name__ == "__main__":
```

```
    app_main()
    Gtk.main()
```

The example shows a simple window containing a progress bar. After everything is set up it constructs a Python thread, passes it a function to execute, starts the thread and the GTK+ main loop. After the main loop is started it is possible to see the window and interact with it.

In the background `example_target()` gets executed and calls `GLib.idle_add()` and `time.sleep()` in a loop. In this example `time.sleep()` represents the blocking operation. `GLib.idle_add()` takes the `update_progess()` function and arguments that will get passed to the function and asks the main loop to schedule its execution in the main thread. This is needed because GTK+ isn't thread safe; only one thread, the main thread, is allowed to call GTK+ code at all times.

Threads: FAQ

- I'm porting code from pygtk (GTK+ 2) to PyGObject (GTK+ 3). Has anything changed regarding threads?

  Short answer: No.

  Long answer: `gtk.gdk.threads_init()`, `gtk.gdk.threads_enter()` and `gtk.gdk.threads_leave()` are now `Gdk.threads_init()`, `Gdk.threads_enter()` and `Gdk.threads_leave()`. `gobject.threads_init()` can be removed.

- I'm using `Gdk.threads_init()` and want to get rid of it. What do I need to do?

  - Remove any `Gdk.threads_init()`, `Gdk.threads_enter()` and `Gdk.threads_leave()` calls. In case they get executed in a thread, move the GTK+ code into its own function and schedule it using `GLib.idle_add()`. Be aware that the newly created function will be executed some time later, so other stuff can happen in between.
  - Replace any call to `Gdk.threads_add_*()` with their GLib counterpart. For example `GLib.idle_add()` instead of `Gdk.threads_add_idle()`.

- What about signals and threads?

  Signals get executed in the context they are emitted from. In which context the object is created or where `connect()` is called from doesn't matter. In GStreamer, for example, some signals can be called from a different thread, see the respective signal documentation for when this is the case. In case you connect to such a signal you have to make sure to not call any GTK+ code or use `GLib.idle_add()` accordingly.

- What if I need to call GTK+ code in signal handlers emitted from a thread?

  In case you have a signal that is emitted from another thread and you need to call GTK+ code during and not after signal handling, you can push the operation with an `threading.Event` object to the main loop and wait in the signal handler until the operation gets scheduled and the result is available. Be aware that if the signal is emitted from the main loop this will deadlock. See the following example

```
# [...]

toggle_button = Gtk.ToggleButton()

def signal_handler_in_thread():

    def function_calling_gtk(event, result):
        result.append(toggle_button.get_active())
        event.set()

    event = threading.Event()
    result = []
```

```
    GLib.idle_add(function_calling_gtk, event, result)
    event.wait()
    toggle_button_is_active = result[0]
    print(toggle_button_is_active)

# [...]
```

- What about the Python GIL ?

  Similar to I/O operations in Python, all PyGObject calls release the GIL during their execution and other Python threads can be executed during that time.

Asynchronous Operations

In addition to functions for blocking I/O glib also provides corresponding asynchronous versions, usually with the same name plus a `_async` suffix. These functions do the same operation as the synchronous ones but don't block during their execution. Instead of blocking they execute the operation in the background and call a callback once the operation is finished or got canceled.

The following example shows how to download a web page and display the source in a text field. In addition it's possible to abort the running operation.

```
import time

from gi.repository import Gio, GLib, Gtk


class DownloadWindow(Gtk.Window):

    def __init__(self):
        super(DownloadWindow, self).__init__(
            default_width=500, default_height=400, title="Async I/O Example")

        self.cancellable = Gio.Cancellable()

        self.cancel_button = Gtk.Button(label="Cancel")
        self.cancel_button.connect("clicked", self.on_cancel_clicked)
        self.cancel_button.set_sensitive(False)

        self.start_button = Gtk.Button(label="Load")
        self.start_button.connect("clicked", self.on_start_clicked)

        textview = Gtk.TextView()
        self.textbuffer = textview.get_buffer()
        scrolled = Gtk.ScrolledWindow()
        scrolled.add(textview)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6,
                      border_width=12)
        box.pack_start(self.start_button, False, True, 0)
        box.pack_start(self.cancel_button, False, True, 0)
        box.pack_start(scrolled, True, True, 0)

        self.add(box)

    def append_text(self, text):
        iter_ = self.textbuffer.get_end_iter()
```

```
        self.textbuffer.insert(iter_, "[%s] %s\n" % (str(time.time()), text))

    def on_start_clicked(self, button):
        button.set_sensitive(False)
        self.cancel_button.set_sensitive(True)
        self.append_text("Start clicked...")

        file_ = Gio.File.new_for_uri(
            "http://python-gtk-3-tutorial.readthedocs.org/")
        file_.load_contents_async(
            self.cancellable, self.on_ready_callback, None)

    def on_cancel_clicked(self, button):
        self.append_text("Cancel clicked...")
        self.cancellable.cancel()

    def on_ready_callback(self, source_object, result, user_data):
        try:
            succes, content, etag = source_object.load_contents_finish(result)
        except GLib.GError as e:
            self.append_text("Error: " + e.message)
        else:
            content_text = content[:100].decode("utf-8")
            self.append_text("Got content: " + content_text + "...")
        finally:
            self.cancellable.reset()
            self.cancel_button.set_sensitive(False)
            self.start_button.set_sensitive(True)


if __name__ == "__main__":
    win = DownloadWindow()
    win.show_all()
    win.connect("delete-event", Gtk.main_quit)

    Gtk.main()
```

The example uses the asynchronous version of `Gio.File.load_contents()` to load the content of an URI pointing to a web page, but first we look at the simpler blocking alternative:

We create a `Gio.File` instance for our URI and call `Gio.File.load_contents()`, which, if it doesn't raise an error, returns the content of the web page we wanted.

```
file = Gio.File.new_for_uri("http://python-gtk-3-tutorial.readthedocs.org/")
try:
    status, contents, etag_out = file.load_contents(None)
except GLib.GError:
    print("Error!")
else:
    print(contents)
```

In the asynchronous variant we need two more things:

- A `Gio.Cancellable`, which we can use during the operation to abort or cancel it.
- And a `Gio.AsyncReadyCallback()` callback function, which gets called once the operation is finished and we can collect the result.

The window contains two buttons for which we register `clicked` signal handlers:

- The `on_start_clicked()` signal handler calls `Gio.File.load_contents_async()` with a `Gio.Cancellable` and `on_ready_callback()` as `Gio.AsyncReadyCallback()`.
- The `on_cancel_clicked()` signal handler calls `Gio.Cancellable.cancel()` to cancel the running operation.

Once the operation is finished, either because the result is available, an error occurred or the operation was canceled, `on_ready_callback()` will be called with the `Gio.File` instance and a `Gio.AsyncResult` instance which holds the result.

To get the result we now have to call `Gio.File.load_contents_finish()` which returns the same things as `Gio.File.load_contents()` except in this case the result is already there and it will return immediately without blocking.

After all this is done we call `Gio.Cancellable.reset()` so the `Gio.Cancellable` can be re-used for new operations and we can click the "Load" button again. This works since we made sure that only one operation can be active at any time by deactivating the "Load" button using `Gtk.Widget.set_sensitive()`.

CHAPTER 8

Frequently Asked Questions

Can I use PyGObject with virtualenv?

To use PyGObject in a virtualenv you have to install it globally and use `--system-site-packages`.

```
virtualenv --system-site-packages --python=python2 venv
source venv/bin/activate
python -c "import gi"
deactivate
```

You can also symlink “gi” and “cairo” in the virtualenv, but this is not supported.

What about the PyGObject package on PyPI?

The PyGObject on PyPI is the old PyGObject 2 and should not be used in new projects.

Can I install PyGObject through pip somehow?

You can install directly from git:

```
virtualenv --system-site-packages --python=python2 venv
source venv/bin/activate
pip install "git+https://git.gnome.org/browse/pygobject@3.22.0"
python -c "import gi"
deactivate
```

Note that this uses autotools internally and not distutils.

How can I use PyGObject with the official CPython builds on Windows?

https://sourceforge.net/projects/pygobjectwin32 provides binaries which should be ABI compatible with the official CPython binaries. I'd recommend using msys2 if at all possible, since there are more people involved and it's easier to fix/patch things yourself.

CHAPTER 9

Application Deployment

There is currently no nice deployment story, but it's not impossible. This is a list of random notes and examples.

Linux

On Linux there is no single strategy. Quod Libet uses distutils, MyPaint uses SCons. Gramps uses distutils.

macOS

On OSX you can use gtk-osx which is based on jhbuild and then gtk-mac-bundler for packaging things up and making libraries relocatable. With macOS bundles you generally have a startup shell script which sets all the various env vars relative to the bundle, similar to jhbuild.

Windows

On Windows things are usually build to be relocatable by default, so no env vars are needed. You can build/install through MSYS2, copy the bits you need and you are done. For GUI application you'll also need an exe launcher that links against the python dll.

Example Deployments

- Quod Libet provides a Windows installer based on MSYS2 and NSIS3. On macOS, jhbuild is used for building, gtk.mac-bundler for packing things up and dmgbuild for creating a dmg. distutis is used for building/installing the application into the final environment. Most of this is automated and scripts can be found in the git repo.
- MyPaint provides a Windows installer based on MSYS2 and Inno Setup. It uses SCons for building/installing the application.

- ...?

Other options

- PyInstaller is a program that freezes (packages) Python programs into stand-alone executables, under Windows, Linux, Mac OS X, and more. PyInstaller's packager has built-in support for automatically including PyGObject dependencies with your application without requiring additional configuration.

CHAPTER 10

Testing and Continuous Integration

To get automated tests of GTK+ code running on a headless server use Xvfb (virtual framebuffer X server). It provides the `xvfb-run -a` command which creates a temporary X server without the need for any real display hardware.

```
xvfb-run -a python my_script.py
```

Continuous Integration using Travis CI / CircleCI

Travis CI uses a rather old Ubuntu and thus the supported GTK+ is at 3.10 and PyGObject is at 3.12. If that's enough for you then have a look at our Travis CI example project:

https://github.com/pygobject/pygobject-travis-ci-examples

To get newer PyGObject, GTK+, etc. working on Travis CI or CircleCI you can use Docker with an image of your choosing. Have a look at our Docker example project which runs tests on various Debian, Ubuntu and Fedora versions:

https://github.com/pygobject/pygobject-travis-ci-docker-examples

CHAPTER 11

Debugging & Profiling

Things can go wrong, these tools may help you find the cause. If you know any more tricks please share them.

GObject Instance Count Leak Check

Requires a development (only available in debug mode) version of glib. Jhbuild recommended.

```
jhbuild shell
GOBJECT_DEBUG=instance-count GTK_DEBUG=interactive ./quodlibet.py
```

- In the GTK+ Inspector switch to the "Statistics" tab
- Sort by "Cumulative" and do the action which you suspect does leak or where you want to make sure it doesn't repeatedly. Like for example opening and closing a window or switching between media files to present.
- If something in the "Cumulative" column steadily increases there probably is a leak.

cProfile Performance Profiling

- https://docs.python.org/2/library/profile.html
- bundled with python

```
python -m cProfile -s [sort_order] quodlibet.py > cprof.txt
```

where `sort_order` can one of the following: calls, cumulative, file, line, module, name, nfl, pcalls, stdname, time

Example output:

```
         885311 function calls (866204 primitive calls) in 12.110 seconds

   Ordered by: cumulative time
```

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002   12.112   12.112 quodlibet.py:11(<module>)
        1    0.007    0.007   12.026   12.026 quodlibet.py:25(main)
19392/13067    0.151    0.000    4.342    0.000 __init__.py:639(__get__)
        1    0.003    0.003    4.232    4.232 quodlibetwindow.py:121(__init__)
        1    0.000    0.000    4.029    4.029 quodlibetwindow.py:549(select_browser)
        1    0.002    0.002    4.022    4.022 albums.py:346(__init__)
        ...
        ...
```

SnakeViz - cProfile Based Visualization

- https://jiffyclub.github.io/snakeviz/
- `pip install snakeviz`

```
python -m cProfile -o prof.out quodlibet.py
snakeviz prof.out
```

Sysprof - System-wide Performance Profiler for Linux

- http://sysprof.com/

```
sysprof-cli -c "python quodlibet/quodlibet.py"
sysprof capture.syscap
```

GDB

```
gdb --args python quodlibet/quodlibet.py
# type "run" and hit enter
```

Debugging Wayland Issues

```
mutter --nested --wayland
# start your app, it should show up in the nested mutter
```

```
weston
# start your app, it should show up in the nested weston
```

Debugging HiDPI Issue

```
GDK_SCALE=2 ./quodlibet/quodlibet.py
```

```
MUTTER_DEBUG_NUM_DUMMY_MONITORS=2 MUTTER_DEBUG_DUMMY_MONITOR_SCALES=1,2 mutter --
↪nested --wayland
# start your app, it should show up in the nested mutter
```

CHAPTER 12

Porting from Static Bindings

Before PyGObject 3, bindings where not generated automatically through gobject introspection and where provided as separate Python libraries like pygobject, pygtk, pygst etc. We call them static bindings.

If your code contains imports like `import gtk`, `import gst`, `import glib` or `import gobject` you are using the old bindings and you should upgrade.

Note that using old and new bindings in the same process is not supported, you have to switch everything at once.

Static Bindings Library Differences

pygtk supported GTK+ 2.0 and Python 2 only. PyGObject supports GTK+ >=3.0 and Python 2/3. If you port away from pygtk you also have to move to GTK+ 3.0 at the same time. **pygtkcompat** described below can help you with that transition.

pygst supports GStreamer 0.10 and Python 2 only. Like with GTK+ you have to move to PyGObject and GStreamer 1.0 at the same time.

pygobject 2 supports glib 2.0 and Python 2. The new bindings also support glib 2.0 and Python 2/3.

General Porting Tips

PyGObject contains a shell script which can help you with the many naming differences between static and dynamic bindings:

https://git.gnome.org/browse/pygobject/plain/pygi-convert.sh

```
./pygi-convert.sh mymodule.py
```

It just does basic text replacement. It reduces the amount of naming changes you have to make in the beginning, but nothing more.

1. Run on a Python module

2. Check/Verify the changes made (e.g. using `git diff`)
3. Finish porting the module by hand
4. Continue to the next module...

Porting Tips for GTK+

While PyGObject theoretically supports GTK+ 2.0 it is not really usable. It will be easier to port to GTK+ 3.0 right away.

For some general advice regarding the migration from GTK+ 2.0 to 3.0 see the offical migration guide. If you need to know how a C symbol is exposed in Python have a look at the symbol mapping listing.

Using the pygtkcompat Compatibility Layer

PyGObject ships a compatibility layer for pygtk which partially emulates the old interfaces:

```
from gi import pygtkcompat
pygtkcompat.enable()
pygtkcompat.enable_gtk(version='3.0')

import gtk
```

`enable()` has to be called once before the first `gtk` import.

Note that pygtkcompat is just for helping you through the transition by allowing you to port one module at a time. Only a limited subset of the interfaces are emulated correctly and you should try to get rid of it in the end.

Default Encoding Changes

Importing `gtk` had the side effect of changing the default Python encoding from ASCII to UTF-8 (check `sys.getdefaultencoding()`) and that no longer happens with PyGObject. Since text with pygtk is returned as utf-8 encoded str, your code is likely depending auto-decoding in many places and you can change it manually by doing:

```
# Python 2 only
import sys
reload(sys)
sys.setdefaultencoding("utf-8")
# see if auto decoding works:
assert '\xc3\xb6' + u'' ==  u'\xf6'
```

While this is not officially supported by Python I don't know of any downsides. Once you are sure that you explicitly decode in all places or you move to Python 3 where things are unicode by default you can remove this again.

CHAPTER 13

Development Guide

Creating a Development Environment

This describes how to work on PyGObject itself. Please follow the instructions on "*Getting Started*" first, as they are a pre-requirement.

Ubuntu / Debian

```
sudo apt build-dep pygobject
git clone https://git.gnome.org/browse/pygobject
cd pygobject
./autogen.sh
make
make check
```

Building & Testing

Building

Building for Python 2:

```
./autogen.sh --with-python=python2
make
```

Building for Python 3:

```
./autogen.sh --with-python=python3
make
```

Testing

To run the test suite:

```
make check
```

To test only a specific class:

```
make check TEST_NAMES=test_everything.TestEverything
```

Contribute

Reporting Bugs

You can search through the GNOME Bugzilla for existing bug reports: https://bugzilla.gnome.org/page.cgi?id=browse.html&product=pygobject

You can also file a new bug report: https://bugzilla.gnome.org/enter_bug.cgi?product=pygobject

Submitting Patches

Make your changes and commit them. Use the following to export the, for example, the last 3 commits:

```
git format-patch -3
```

Open a new bug report and attach the resulting files.

Style Guidelines

Python Code

- Generally follow Python's PEP8 style guidelines. We run the pep8 command to verify this during unittest runs.
- Break up logical blocks of related code with a newline. Specifically add a blank newline after conditional or looping blocks.
- Don't comment what is obvious. Instead prefer meaningful names of functions and variables:

```
# Get the functions signal annotations <-- this comment is unnecessary
return_type, arg_types = get_signal_annotations(func)
```

- Use comments to explain non-obvious blocks and conditionals, magic, workarounds (with bug references), or generally complex pieces of code. Good examples:

```
# If a property was defined with a decorator, it may already have
# a name; if it was defined with an assignment (prop = Property(...))
# we set the property's name to the member name
if not prop.name:
    prop.name = name
```

```
# Python causes MRO's to be calculated starting with the lowest
# base class and working towards the descendant, storing the result
# in __mro__ at each point. Therefore at this point we know that
# we already have our base class MRO's available to us, there is
# no need for us to (re)calculate them.
if hasattr(base, '__mro__'):
    bases_of_subclasses += [list(base.__mro__)]
```

Python Doc Strings

- Doc strings should generally follow PEP257 unless noted here.
- Use reStructuredText (resST) annotations.
- Use three double quotes for doc strings (""").
- Use a brief description on the same line as the triple quote.
- Include function parameter documentation (including types, returns, and raises) between the brief description and the full description. Use a newline with indentation for the parameters descriptions.

```
def spam(amount):
    """Creates a Spam object with the given amount.

    :param int amount:
        The amount of spam.
    :returns:
        A new Spam instance with the given amount set.
    :rtype: Spam
    :raises ValueError:
        If amount is not a numeric type.

    More complete description.
    """
```

- For class documentation, use the classes doc string for an explanation of what the class is used for and how it works, including Python examples. Include `__init__` argument documentation after the brief description in the classes doc string. The class `__init__` should generally be the first method defined in a class putting it as close as possible (location wise) to the class documentation.

```
class Bacon(CookedFood):
    """Bacon is a breakfast food.

    :param CookingType cooking_type:
        Enum for the type of cooking to use.
    :param float cooking_time:
        Amount of time used to cook the Bacon in minutes.

    Use Bacon in combination with other breakfast foods for
    a complete breakfast. For example, combine Bacon with
    other items in a list to make a breakfast:

    .. code-block:: python

        breakfast = [Bacon(), Spam(), Spam(), Eggs()]

    """
```

```
    def __init__(self, cooking_type=CookingType.BAKE, cooking_time=15.0):
        super(Bacon, self).__init__(cooking_type, cooking_time)
```

Python Override Guidelines

This document serves as a guide for developers creating new PyGObject overrides or modifying existing ones. This document is not intended as hard rules as there may always be pragmatic exceptions to what is listed here. It is also a good idea to study the Zen of Python by Tim Peters.

In general, overrides should be minimized and preference should always be placed on updating the underlying API to be more bindable, adding features to GI to support the requirement, or adding mechanical features to PyGObject which can apply generically to all overrides (#721226 and #640812).

If a GI feature or more bindable API for a library is in the works, it is a good idea to avoid the temptation to add temporary short term workarounds in overrides. The reason is this can creaste unnecessary conflicts when the bindable API becomes a reality (#707280).

- Minimize class overrides when possible.

  Reason: Class overrides incur a load time performance penalty because they require the classes GType and all of the Python method bindings to be created. See #705810

- Prefer monkey patching methods on repository classes over inheritance.

  Reason: Class overrides add an additional level to the method resolution order (mro) which has a performance penalty. Since overrides are designed for specific repository library APIs, monkey patching is reasonable because it is utilized in a controlled manner by the API designer (as opposed to monkey patching a third-party library which is more fragile).

- Avoid overriding `__init__` *Reason*: Sub-classing the overridden class then becomes challenging and has the potential to cause bugs (see #711487 and reasoning listed in https://wiki.gnome.org/Projects/PyGObject/InitializerDeprecations).

- Unbindable functions which take variadic arguments are generally ok to add Python implementations, but keep in mind the prior noted guidelines. A lot of times adding bindable versions of the functions to the underlying library which take a list is acceptable. For example: #706119. Another problem here is if an override is added, then later a bindable version of the API is added which takes a list, there is a good chance we have to live with the override forever which masks a working version implemented by GI.

- Avoid side effects beyond the intended repositories API in function/method overrides.

  Reason: This conflates the original API and adds a documentation burden on the override maintainer.

- Don't change function signatures from the original API and don't add default values.

  Reason: This turns into a documentation discrepancy between the libraries API and the Python version of the API. Default value work should focus on bug #558620, not cherry-picking individual Python functions and adding defaults.

- Avoid implicit side effects to the Python standard library (or anywhere).
  - Don't modify or use sys.argv

    Reason: sys.argv should only be explicitly controlled by application developers. Otherwise it requires hacks to work around a module modifying or using the developers command line args which they rightfully own.

```
saved_argv = sys.argv.copy()
sys.argv = []
from gi.repository import Gtk
sys.argv = saved_argv
```

  - Never set Pythons default encoding.

    Reason: Read or watch Ned Batchelders "Pragmatic Unicode"

- For PyGTK compatibility APIs, add them to PyGTKCompat not overrides.
- Prefer adapter patterns over of inheritance and overrides.

  Reason: An adapter allows more flexibility and less dependency on overrides. It allows application developers to use the raw GI API without having to think about if a particular typelibs overrides have been installed or not.

Packaging

Some notes on how to package PyGObject

Existing Packages:

- https://www.archlinux.org/packages/extra/x86_64/python-gobject
- https://packages.qa.debian.org/p/pygobject.html
- https://github.com/Alexpux/MINGW-packages/tree/master/mingw-w64-pygobject

Building:

```
./configure --with-python=${PYTHON} --prefix="${PREFIX}"
make check # if you want to run the test suite
make DESTDIR="${PKGDIR}" install
```

Runtime dependencies:

- glib
- libgirepository (gobject-introspection)
- libffi
- Python 2 or 3

The overrides directory contains various files which includes various Python imports mentioning gtk, gdk etc. They are only used when the corresponding library is present, they are not needed.

Build dependencies:

- The runtime dependencies
- cairo (optional)
- pycairo (optional)
- pkg-config

If autotools is used:

- gnome-common for PyGObject < 3.26
- autoconf-archive for PyGObject >= 3.26

Test Suite dependencies:

- The runtime dependencies
- GTK+ 3 (optional)
- pango (optional)
- pycairo (optional)

CHAPTER 14

Maintainer Guide

Making a Release

1. Make sure configure.ac has the right version number
2. Update NEWS file (use `make release-news` target and then edit as you see fit)
3. Run `make distcheck`, fix any issues and commit.
4. Upload tarball: `scp pygobject-3.X.Y.tar.gz master.gnome.org:`
5. Install tarball: `ssh master.gnome.org 'ftpadmin install pygobject-3.X.Y.tar.gz'`
6. Commit NEWS as `"release 3.X.Y"` and push
7. Tag with: `git tag -s 3.X.Y -m "release 3.X.Y"`
8. Push tag with: `git push origin 3.X.Y`
9. Commit post-release version bump to configure.ac
10. Send release announcements to gnome-announce-list@gnome.org; pygtk@daa.com.au; python-hackers-list@gnome.org; python-announce-list@python.org
11. Blog about it (include the HTMLized NEWS that `make release-news` prints)

Based on http://live.gnome.org/MaintainersCorner/Releasing

Branching

Each cycle after the feature freeze, we create a stable branch so development can continue in the master branch unaffected by the freezes.

1. Create the branch locally with: `git checkout -b pygobject-3-2`
2. Push new branch: `git push origin pygobject-3-2`
3. In master, update configure.ac to what will be the next version number (3.3.0)

4. Announce the branching, send email telling people to continue development in master and cherry-picking the changes that are appropriate for the stable branch

CHAPTER 15

Further Resources

Python GTK+ 3 Tutorial Many examples showing how to build an application using PyGObject and GTK+.

Python GI API Reference Auto generated API documentation for many libraries accessible through PyGObject.

PyGObject Wiki on gnome.org Contains various information collected over time.

CHAPTER 16

Contact

IRC #python on irc.gnome.org

Logs for the channel: http://quodlibet.duckdns.org/irc/pygobject

Mailinglist https://mail.gnome.org/mailman/listinfo/python-hackers-list

PyGObject is a Python package which provides bindings for GObject based libraries such as GTK+, GStreamer, WebKitGTK+, GLib, GIO and many more.

If you want to write a Python application for GNOME or a Python GUI application using GTK+, then PyGObject is the way to go. Also check out the "Python GTK+ 3 Tutorial" and the "Python GI API Reference".

CHAPTER 17

How does it work?

PyGObject uses glib, gobject, girepository, libffi and other libraries to access the C library (libgtk-3.so) in combination with the additional metadata from the accompanying typelib file (Gtk-3.0.typelib) and dynamically provides a Python interface based on that information.

CHAPTER 18

Who Is Using PyGObject?

- D-Feet - an easy to use D-Bus debugger
- GNOME Music - a music player for GNOME
- GNOME Tweak Tool - a tool to customize advanced GNOME 3 options
- Gramps - a genealogy program
- Lollypop - a modern music player
- Meld - a visual diff and merge tool
- MyPaint - a nimble, distraction-free, and easy tool for digital painters
- Orca - a flexible and extensible screen reader
- Pithos - a Pandora Radio client
- Pitivi - a free and open source video editor
- Quod Libet - a music library manager / player
- Transmageddon - a video transcoder

The following applications or libraries use PyGObject for optional features, such as plugins or as optional backends:

- beets - a music library manager and MusicBrainz tagger
- gedit- a GNOME text editor
- matplotlib - a python 2D plotting library
- Totem - a video player for GNOME

Index

G