Python Desktop Handbook

Building Desktop Applications with GTK 4 and Qt 6

Peter Gill

Version 1.0-draft

Contents

Changelog **1**
Version 1.0 — in progress . . . 1
Brought up to GNOME 50 . . . 2
Earlier editions . . . 2

I GTK 4 with PyGObject **3**

1 Getting Started with GTK 4 **5**
1.1 Introduction . . . 5
1.1.1 What you need installed . . . 5
1.1.2 What this book targets . . . 6
1.1.3 This book assumes Wayland . . . 6
1.1.4 The two lines at the top of every program . . . 6
1.2 GTK 4 basics . . . 7
1.2.1 Widgets — what are they? . . . 7
1.2.2 Creating your first GTK 4 application . . . 7
1.2.3 Layout — boxes . . . 8
1.2.4 Callbacks — reacting to program events . . . 9
1.3 Widgets . . . 10
1.3.1 Buttons . . . 10
1.3.2 Toggle buttons, check buttons and radio groups . . . 11
1.3.3 Labels . . . 11
1.3.4 Text entries . . . 12
1.3.5 Numbers: spin buttons and scales . . . 13
1.3.6 Choices: drop downs . . . 13
1.3.7 Menus and actions . . . 14
1.3.8 Message dialogs . . . 15
1.3.8.1 In a libadwaita application, use libadwaita's . . . 16
1.3.9 Feedback: toasts instead of a status bar . . . 17
1.4 Libadwaita . . . 17
1.4.1 Adaptive layouts . . . 19
1.4.2 Keyboard shortcuts . . . 19
1.5 Summary . . . 20

2 GObject: Properties, Signals and Bindings **21**
2.1 Introduction . . . 21
2.2 Subclassing . . . 21
2.3 Properties . . . 22
2.3.1 Naming . . . 22
2.3.2 Computed and validating properties . . . 22
2.3.3 Watching for changes . . . 23

2.3.4 How they fail . . . 23
2.4 Signals . . . 23
2.4.1 The default handler runs first . . . 23
2.4.2 Return values and vetoes . . . 24
2.4.3 Disconnecting . . . 24
2.5 Bindings . . . 24
2.5.1 Not everything is a property . . . 25
2.5.2 Keeping the binding . . . 25
2.6 Why the list widgets needed all this . . . 25
2.7 Summary . . . 25

3 More GTK 4 **27**
3.1 Introduction . . . 27
3.2 Lists and tables . . . 27
3.2.1 The item type . . . 27
3.2.2 A list . . . 28
3.2.3 A sortable table . . . 29
3.2.4 Choosing between the views . . . 30
3.3 File dialogs . . . 31
3.3.1 You get a GFile, not a path . . . 31
3.4 Drag and drop . . . 31
3.5 Copy and paste . . . 33
3.6 Images and pictures . . . 33
3.7 Tooltips . . . 34
3.8 Building interfaces from UI files . . . 34
3.8.1 GtkTemplate . . . 35
3.9 Notifications, not status icons . . . 36
3.10 Summary . . . 36

4 Threads and Asynchronous Work **39**
4.1 Introduction . . . 39
4.2 The one rule . . . 39
4.3 Asynchronous I/O, without threads . . . 40
4.3.1 Cancellation . . . 40
4.4 Threads . . . 41
4.4.1 About the GIL . . . 42
4.5 Breaking work up . . . 42
4.6 asyncio . . . 42
4.6.1 Gio's async methods become awaitable . . . 43
4.6.2 A coroutine may touch widgets . . . 43
4.6.3 Two traps . . . 43
4.6.4 The older forms . . . 44
4.7 Choosing . . . 44
4.8 Summary . . . 45

5 Drawing with Cairo **47**
5.1 Introduction . . . 47
5.2 Cairo basics . . . 47
5.2.1 Surface formats . . . 48
5.2.2 The same drawing, different surfaces . . . 49
5.3 Drawing in a widget . . . 49
5.4 Paths . . . 50
5.4.1 Stroking . . . 51
5.4.2 Filling . . . 51

5.4.3 Colour and patterns . . . 51
5.4.4 Transformations and state . . . 51
5.5 Text . . . 53
5.6 Antialiasing . . . 54
5.7 Making it interactive . . . 54
5.8 Beyond Cairo: GtkSnapshot . . . 55
5.9 Summary . . . 56

6 Custom Widgets **57**
6.1 Introduction . . . 57
6.2 Compose first . . . 57
6.3 A widget that lays out children . . . 58
6.3.1 Unparent your children, or GTK complains . . . 59
6.3.2 Measuring . . . 59
6.3.3 The rule that catches everyone . . . 59
6.3.4 Allocating . . . 60
6.3.5 Drawing . . . 60
6.4 Layout managers . . . 60
6.5 Styling and accessibility . . . 61
6.6 Summary . . . 62

7 Printing **63**
7.1 Introduction . . . 63
7.2 The shape of a print job . . . 63
7.2.1 One line you need before any of it works . . . 64
7.3 Exporting a PDF . . . 64
7.4 Drawing a page . . . 64
7.5 Pagination . . . 66
7.6 Page setup and settings . . . 66
7.7 Under a sandbox . . . 67
7.8 Summary . . . 67

8 Desktop Integration **69**
8.1 Introduction . . . 69
8.2 The application id . . . 69
8.3 Settings . . . 69
8.3.1 The schema . . . 70
8.3.2 Reading and writing . . . 71
8.3.3 Binding, which is the good part . . . 71
8.4 Where files go . . . 71
8.5 Desktop files . . . 72
8.5.1 Notifications need one . . . 73
8.6 Passwords . . . 73
8.7 Opening things . . . 74
8.8 Portals and the sandbox . . . 74
8.9 Summary . . . 75

9 Audio and Video with GStreamer **77**
9.1 Introduction . . . 77
9.2 A video player . . . 78
9.3 Pipelines . . . 78
9.3.1 Building a pipeline from a string . . . 79
9.3.2 States . . . 80
9.3.3 The bus . . . 80
9.4 What is in a file . . . 81

9.5 Missing codecs . . . 81
9.6 Converting a file . . . 82
9.6.1 uridecodebin, not uridecodebin3 . . . 82
9.7 Video in your own widget . . . 83
9.7.1 Let the compositor do the work . . . 83
9.8 Summary . . . 84

10 D-Bus **85**
10.1 Introduction . . . 85
10.2 The vocabulary . . . 85
10.3 Looking before you write . . . 86
10.4 Calling a service . . . 86
10.4.1 Calling asynchronously . . . 87
10.5 Listening . . . 87
10.6 Being a service . . . 88
10.6.1 Owning the name . . . 89
10.7 Testing without a desktop . . . 89
10.8 Portals are D-Bus . . . 90
10.9 Summary . . . 90

11 Animation and Transitions **91**
11.1 Introduction . . . 91
11.2 Transitions you get for free . . . 91
11.3 CSS . . . 92
11.4 AdwAnimation . . . 93
11.5 The frame clock . . . 94
11.6 Moving a whole widget . . . 95
11.7 Respecting the user . . . 95
11.8 Colour scheme, accent and contrast . . . 96
11.8.1 What this means for anything you drew yourself . . . 96
11.9 Summary . . . 96

12 Embedding Web Content **99**
12.1 Introduction . . . 99
12.1.1 Get the version right . . . 99
12.2 A browser . . . 100
12.3 Deciding what the page may do . . . 100
12.4 Running JavaScript . . . 101
12.5 Letting the page call back . . . 101
12.5.1 Treat the page as untrusted . . . 102
12.6 Loading your own HTML . . . 102
12.7 Settings worth changing . . . 103
12.8 The sandbox . . . 103
12.9 Summary . . . 103

13 Internationalization **105**
13.1 Introduction . . . 105
13.2 Marking strings . . . 105
13.2.1 Plurals . . . 106
13.2.2 Context . . . 106
13.2.3 Formatting . . . 106
13.2.4 Comments for translators . . . 106
13.3 The GTK-specific part . . . 106
13.4 Strings in .ui files . . . 107
13.5 The pipeline . . . 108

13.6 Testing it . . . 108
13.7 More than words . . . 109
13.8 Summary . . . 110

14 Testing **111**
14.1 Introduction . . . 111
14.2 Setting up . . . 111
14.3 Testing the logic . . . 112
14.3.1 Testing that a property notifies . . . 112
14.4 Testing widgets . . . 113
14.4.1 Pressing things . . . 113
14.4.2 Lists build their rows lazily . . . 113
14.5 Running without a display . . . 114
14.6 Smoke tests . . . 114
14.7 Driving the whole application . . . 115
14.8 Types and static checks . . . 115
14.9 What to test . . . 116
14.10Summary . . . 116

15 Packaging and Distribution **119**
15.1 Introduction . . . 119
15.2 What has to be installed . . . 119
15.3 Meson . . . 120
15.3.1 The generated launcher . . . 120
15.3.2 GResource . . . 121
15.3.3 Validate at build time . . . 121
15.4 AppStream metadata . . . 122
15.5 Flatpak . . . 122
15.6 The other routes . . . 123
15.7 Try it . . . 123
15.8 Summary . . . 124

A Book Text Licenses **125**
A.1 Attribution-ShareAlike 4.0 International . . . 125
A.1.1 Section 1 – Definitions . . . 125
A.1.2 Section 2 – Scope . . . 126
A.1.3 Section 3 – License Conditions . . . 127
A.1.4 Section 4 – Sui Generis Database Rights . . . 128
A.1.5 Section 5 – Disclaimer of Warranties and Limitation of Liability . . . 128
A.1.6 Section 6 – Term and Termination . . . 128
A.1.7 Section 7 – Other Terms and Conditions . . . 129
A.1.8 Section 8 – Interpretation . . . 129
A.2 About Creative Commons . . . 129

B Source Code License **131**
B.1 The MIT License . . . 131
B.2 What this does not cover . . . 131

C Migrating from PyGTK **133**
C.1 The shape of a program . . . 133
C.2 Imports and versions . . . 134
C.3 Containers and visibility . . . 134
C.4 Widgets that were renamed or replaced . . . 134
C.5 Dialogs stopped blocking . . . 134
C.6 Signals . . . 135

C.7 Drawing . . . 135
C.8 Threads . . . 136
C.9 Things with no replacement . . . 136

D Icon Names **137**
D.0.1 The -symbolic suffix . . . 137
D.0.2 Checking a name exists . . . 137
D.1 Files and documents . . . 137
D.2 Editing . . . 138
D.3 Text formatting . . . 138
D.4 Navigation . . . 139
D.5 View and zoom . . . 139
D.6 Media . . . 139
D.7 Dialogs and status . . . 140
D.8 Names with no stock ancestor . . . 140
D.9 Stock items with no replacement . . . 140

Further Reading **141**
Reference documentation . . . 141
Guidance rather than API . . . 141
Things to read the source of . . . 141
Tools worth having installed . . . 142
Where the previous edition went . . . 142

List of Figures

1.1 The first window . . . 8
1.2 A column containing a row and a button . . . 9
1.3 Text, icon, icon-and-label, suggested and destructive buttons . . . 10
1.4 A toggle button, a check button, a radio group made of check buttons, and a switch . . . 12
1.5 An entry, a password entry, a search entry and a text view . . . 13
1.6 A spin button, a scale and a drop down . . . 14
1.7 One menu model driving both a menu bar and a header bar button . . . 15
1.8 The same widgets as libadwaita rows, with no hand-written spacing . . . 18

3.1 A list view over a Gio.ListStore . . . 29
3.2 A column view, sortable by header and filtered by the search entry . . . 30
3.3 Drag sources on the left, a drop target on the right . . . 32
3.4 A themed icon above a photograph in a Gtk.Picture . . . 34
3.5 A window whose layout came from a .ui file . . . 36

4.1 Progress reported from a worker thread through GLib.idle_add . . . 42

5.1 Two lines drawn to an image surface . . . 48
5.2 Cairo drawing inside a Gtk.DrawingArea . . . 50
5.3 Paths, fills, gradients and transforms . . . 52
5.4 Cairo's text API next to a Pango layout . . . 54
5.5 The same shapes with antialiasing on and off . . . 54
5.6 A widget drawn from render nodes rather than with Cairo . . . 56

6.1 A composed widget: one entry, one drop down and one button, presented as a single thing 58
6.2 Children wrapped onto three rows by a custom container . . . 61

7.1 A page produced by the export example . . . 65

8.1 Widgets bound to GSettings keys, remembered between runs . . . 72

9.1 Gtk.Video playing the generated test clip . . . 79

11.1 A stack with a switcher, and a revealer below it . . . 92
11.2 The swatch mid-transition, after the round class was added . . . 93

12.1 A page posting a message to Python, and Python reading a value back . . . 102

13.1 The greeter in its original English . . . 109
13.2 The same window under LANGUAGE=de . . . 109

Changelog

Version 1.0 — in progress

The book is being rewritten. The previous edition taught PyGTK and GTK 2, a stack that has had no release since 2011; this one teaches GTK 4 with PyGObject in Part I and Qt 6 with PySide6 in Part II.

Three things changed about how the book is made, as well as what is in it:

- The source moved from LyX to Markdown. The site is built with Jekyll and the PDF with pandoc, from the same files.
- Every listing is a file under `examples/`, and every example is started and shut down again on each build. Code that stops working fails the build.
- Figures that can be generated are generated, by running the example that draws them, rather than being screenshots that slowly stop matching the text.

Part I — GTK 4 with PyGObject — is complete:

- **Getting Started with GTK 4** — `Gtk.Application`, boxes without packing arguments, grouped check buttons in place of radio buttons, `Gtk.DropDown`, menus as models plus actions, asynchronous dialogs, and libadwaita.
- **GObject** — new. Properties, signals and bindings: the layer the rest of Part I is built on and the previous edition never explained.
- **More GTK 4** — list and column views over `Gio.ListStore`, `Gtk.FileDialog`, drag and drop through controllers, `Gtk.Picture`, `Gtk.Template`, notifications.
- **Threads and Asynchronous Work** — new. Gio's async APIs, worker threads, cancellation, and `async`/`await` on the GLib main loop through PyGObject's own asyncio integration. `gdk_threads_enter()` is gone and has no replacement.
- **Drawing with Cairo** — Cairo through `Gtk.DrawingArea`, Pango for text, `GtkSnapshot` for widget drawing.
- **Custom Widgets** — new, and finally written: the previous edition's chapter of this name contained one line. Composition, measure and allocate, and layout managers.
- **Printing** — `Gtk.PrintOperation`, pagination, and exporting a PDF so printing is testable without a printer.
- **Desktop Integration** — GSettings in place of GConf, desktop files, libsecret in place of the gnome-keyring API, portals.
- **Audio and Video with GStreamer** — GStreamer 1.0, `Gtk.MediaFile`, pipelines and the bus, discovery, transcoding.
- **D-Bus** — GDBus in place of dbus-python, proxies, signals, and exporting a service.
- **Animation and Transitions** — replaces Clutter with container transitions, CSS, `Adw.Animation` and the frame clock.
- **Embedding Web Content** — WebKitGTK 6.0 in place of gtkmozembed, and the JavaScript bridge.
- **Internationalization** — gettext without intltool, and the two text-domain bindings GTK needs.
- **Testing** — new. Keeping the rules out of the widgets, testing that properties notify, widget tests, headless runs, smoke tests and type checking.

- **Packaging and Distribution** — replaces IronPython and Gtk#. Meson, GResource, AppStream metadata and Flatpak.
- **Migrating from PyGTK** — a translation table from the previous edition's idioms.

The appendices were rewritten too: *Icon Names* is now a stock-item to icon-name mapping rather than the GTK 2 stock list, and the bibliography is now *Further Reading*, pointing at documentation that still exists.

Brought up to GNOME 50

The text targets the GNOME 50 stack — GTK 4.22, libadwaita 1.9, PyGObject 3.56 — and the Flatpak manifest builds against the GNOME 50 runtime. GNOME 50 removed its X11 session, so the book is now written for Wayland only, and Getting Started says what that rules out.

Sections added for the APIs that became the idiomatic answer in the meantime:

- Adaptive layout with `Adw.Breakpoint`, in Getting Started.
- `Adw.Dialog` and `Adw.AlertDialog`, which adapt to a bottom sheet at phone width, in Getting Started.
- Keyboard shortcuts — accelerators, `Gtk.ShortcutController` and `Adw.ShortcutsDialog` — in Getting Started.
- Copy and paste with `Gdk.Clipboard`, in More GTK 4.
- Colour scheme, accent colour and high contrast through `Adw.StyleManager`, in Animation and Transitions.
- `Gtk.GraphicsOffload`, in Audio and Video.

Every example is now annotated with type hints.

Part II — Qt 6 with PySide6 — is not written yet.

Dropped, because the technology was retired rather than replaced: the IronPython and Gtk# chapter, the PyGTK-on-Windows appendix, and the unfinished Telepathy, Geoclue and custom-widget chapters.

Both licences changed as well. The text moved from CC BY-SA 3.0 to 4.0, and the sample code from the LGPL v3 to MIT.

Earlier editions

Versions 0.03 to 0.13, from December 2008 to October 2012, covered PyGTK and GTK 2: widgets and layout, Glade and libglade, Cairo, printing, GConf and desktop integration, GStreamer, D-Bus, Clutter, embedded Mozilla and Internet Explorer, internationalization, and IronPython with Gtk#.

That edition is preserved in the git history of this repository, and the last PDF built from the LyX source is pygtk-notebook-latest-0.13.pdf.

Part I

GTK 4 with PyGObject

Chapter 1

Getting Started with GTK 4

> Every listing in this chapter is a file under `examples/gtk4/`. They are run on each build, so if one of them stops working the build says so.

1.1 Introduction

GTK 4 is the toolkit behind GNOME, and PyGObject is how Python talks to it. There is no separate binding to install and no wrapper library to learn: PyGObject reads GTK's introspection data at import time, so the Python API and the C API are the same API with different punctuation. When the GTK documentation says `gtk_widget_set_visible()`, you write `widget.set_visible()`.

If you last wrote GUI code with PyGTK, the shape of this chapter will be familiar and most of the details will not be. GTK 4 removed a lot: no more `gtk.RadioButton`, no more `show_all()`, no more packing arguments on every `pack_start` call, no more `gtk.main()`. What replaced them is generally smaller, and this chapter introduces each replacement where the old idiom used to sit.

Alongside GTK 4 sits **libadwaita**, GNOME's widget library. GTK gives you buttons and boxes; libadwaita gives you the rows, dialogs and adaptive layouts that make an application look like it belongs on the desktop. You can write GTK 4 without it, and the first half of this chapter does, but the last section shows why you probably do not want to.

1.1.1 What you need installed

GTK 4 and PyGObject come from your distribution, not from PyPI. Installing PyGObject with `pip` builds it against whatever GTK you already have and fails confusingly when you have none, so start with the system packages.

```
# Debian, Ubuntu
sudo apt install python3-gi python3-gi-cairo gir1.2-gtk-4.0 gir1.2-adw-1

# Fedora
sudo dnf install python3-gobject gtk4 libadwaita

# Arch
sudo pacman -S python-gobject gtk4 libadwaita
```

Check the result. This prints the running GTK and libadwaita versions and exits:

```
python3 -c "import gi; gi.require_version('Gtk','4.0'); gi.require_version('Adw','1'); \
  from gi.repository import Gtk, Adw; \
```

```
  print('GTK', Gtk.get_major_version(), Gtk.get_minor_version()); \
  print('Adw', Adw.MAJOR_VERSION, Adw.MINOR_VERSION)"
```

1.1.2 What this book targets

Everything here was written and tested against the GNOME 50 stack:

Component	Version	Where it comes from
GTK	4.22	your distribution
libadwaita	1.9	your distribution
PyGObject	3.56	your distribution
Python	3.12 or later	your distribution

That is what shipped in March 2026, and what Ubuntu 26.04 LTS and Fedora 44 carry. The book does not chase the newest API for its own sake, and most of it works several releases back — but where a section needs something recent it says which version, like this:

> **libadwaita 1.4.** `Adw.Breakpoint` needs it. On older versions the window still works; it just does not adapt.

Two version numbers are worth committing to memory, because they are where the current idioms arrived: **GTK 4.10**, which deprecated the blocking dialogs and introduced the asynchronous ones, and **libadwaita 1.5**, which introduced `Adw.Dialog`. Code older than those two will look noticeably different from everything in this book.

Python 3.12 is the floor for the examples, which use `type` statements and modern generics in their annotations. PyGObject 3.50 is the floor for the `async`/`await` integration in Threads and Asynchronous Work; the rest of the book does not need it.

1.1.3 This book assumes Wayland

GNOME 50 removed its X11 session. GNOME now runs on Wayland only, and so does the advice in this book. Nothing here is X11-specific, but a few habits carried over from X11 are worth unlearning now rather than debugging later:

- **A window cannot position itself.** There is no `move()`, no way to ask for screen coordinates, and no way to place a window under the pointer. The compositor decides. Dialogs get placed correctly only because you told them which window is their parent — which is why every dialog call in this book passes one.
- **A window does not know where it is**, only how big it is.
- **You cannot read the screen or other windows.** Screenshots, screen sharing and global shortcuts go through portals, which ask the user. That is Desktop Integration.
- **Scaling can be fractional.** Do not assume the scale factor is a whole number, and draw in logical units rather than pixels wherever you can. The Cairo chapter comes back to this.

XWayland still runs X11 clients, so an old application keeps working; the point is that new code should not be written against those assumptions. If you are testing on a headless machine, the examples run under `xvfb-run` — that is XWayland's territory rather than a recommendation, and it is only how this book's own build takes its screenshots.

1.1.4 The two lines at the top of every program

```
import gi
```

```
gi.require_version("Gtk", "4.0")
from gi.repository import Gtk
```

`gi.require_version` picks which version of the GTK typelib to load, and it has to run *before* the import. Many systems have GTK 3 and GTK 4 installed side by side; without that call you get whichever one PyGObject finds first, along with a warning and, sooner or later, a confusing crash. Do the same for every library that has more than one version in circulation — `Adw`, `Gst`, `WebKit` all need it.

1.2 GTK 4 basics

1.2.1 Widgets — what are they?

A widget is a piece of a user interface. Labels, buttons, menus, text entries, sliders, whole scrolling panes — all widgets. If you came from .NET or Qt you would call them controls; the idea is the same.

Two things about GTK 4 widgets are worth knowing before you write any code.

Every widget is a container. In GTK 3, a `GtkButton` was a `GtkContainer` that happened to hold a label. In GTK 4, every widget can have children, and the ones that hold exactly one child expose it as a property:

```
button = Gtk.Button()
button.set_child(Gtk.Label(label="Save"))
```

Widgets are visible by default. GTK 3 required `show()` on every widget and `show_all()` on the window. GTK 4 dropped both. You create widgets, you add them to a window, and you call `present()` on the window. Anything you want hidden, you hide explicitly with `set_visible(False)`.

1.2.2 Creating your first GTK 4 application

A GTK 4 program is a `Gtk.Application`. It owns the main loop, handles single-instance behaviour, holds your actions, and knows when the last window has closed so it can exit.

```
import sys

import gi

gi.require_version("Gtk", "4.0")
from gi.repository import Gtk


def on_activate(app):
    window = Gtk.ApplicationWindow(application=app, title="Hello World")
    window.set_default_size(320, 120)
    window.set_child(Gtk.Label(label="Hello World!"))
    window.present()


app = Gtk.Application(application_id="com.example.HelloWorld")
app.connect("activate", on_activate)
sys.exit(app.run(sys.argv))
```

Running it opens a window with a label in it, and closing the window ends the program. Four things are doing work here.

The **application id** is a reverse-DNS name that identifies your program to the desktop. It is not decoration: D-Bus activation, the desktop file, notifications, and the single-instance check all key off it.

Figure 1.1: The first window

Use a domain you control; use `com.example.Something` while you are experimenting.

The **activate** signal fires when the application is asked to show itself — on startup, and again if someone launches it a second time. Build your window there, not at import time.

`Gtk.ApplicationWindow` is a window that knows which application it belongs to. Passing `application=app` is what ties the two together; without it the application has no windows, decides it has nothing to do, and exits immediately.

`app.run(sys.argv)` starts the main loop and returns an exit status when the last window closes. This is where `gtk.main()` went.

> **Where did `gtk.main()` go?** You can still get at the loop directly through `GLib.MainLoop`, and a few of the examples later in this book do when there is no window involved. For anything with a user interface, `Gtk.Application` is the supported path and gives you actions, accelerators and session integration for free.

1.2.3 Layout — boxes

A window holds one child. To get more than one widget on screen you need a layout container, and the workhorse is `Gtk.Box`: a row or a column of widgets.

```
outer = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=12)
outer.set_margin_top(12)
outer.set_margin_bottom(12)
outer.set_margin_start(12)
outer.set_margin_end(12)

row = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=6)
row.append(Gtk.Label(label="Hello World!"))
row.append(Gtk.Label(label="Still in the row"))

outer.append(row)
outer.append(Gtk.Button(label="This button is below the row"))

window.set_child(outer)
```

Boxes nest, and nesting is how you build a layout: one box per row or column, each holding widgets or further boxes, with a single top-level box in the window.

The GTK 3 call was `box.pack_start(child, expand, fill, padding)`. GTK 4 has `append()` and `prepend()`, and the three extra arguments moved onto the child widget itself:

```
label.set_hexpand(True)          # take any spare horizontal space
label.set_halign(Gtk.Align.START)  # but sit at the start of it
label.set_margin_start(6)        # padding is now a margin
```

That split takes a moment to get used to and then stops being confusing: `expand` asks for space, `align`

says what to do with the space you were given, and margins are the widget's own business. Fill is the default — a widget that is given space uses it unless its alignment says otherwise.

`spacing` on the box is the gap *between* children. Margins on the box are the gap around the outside. Setting four margins by hand gets old, so most real code either uses libadwaita's rows, which have their own spacing, or a small helper.

For a grid of widgets rather than a line of them, `Gtk.Grid` attaches children at a column and row with a column and row span:

```
grid = Gtk.Grid(column_spacing=6, row_spacing=6)
grid.attach(Gtk.Label(label="Name"), 0, 0, 1, 1)
grid.attach(Gtk.Entry(), 1, 0, 2, 1)
```

The full example is `examples/gtk4/boxes.py`.

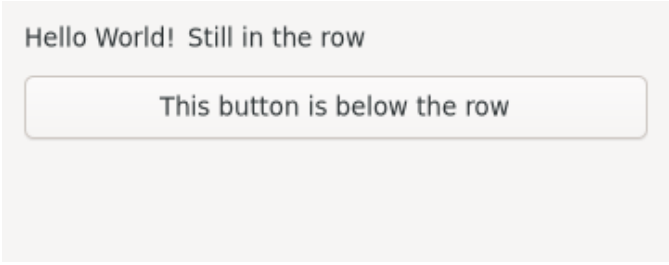

Figure 1.2: A column containing a row and a button

1.2.4 Callbacks — reacting to program events

A program that ignores its user is not much of a program. Widgets emit **signals** when something happens to them, and you connect a Python function to the signals you care about.

```
def on_button_clicked(button):
    button.set_label("Clicked!")


plain = Gtk.Button(label="Click me")
plain.connect("clicked", on_button_clicked)
```

The first argument to a callback is always the widget that emitted the signal. Anything you pass to `connect()` after the callback is handed back to it unchanged, which is how you get your own data into the handler:

```
def on_counter_clicked(button, state):
    state["count"] += 1
    button.set_label(f"Clicked {state['count']} times")


counter = Gtk.Button(label="Clicked 0 times")
counter.connect("clicked", on_counter_clicked, {"count": 0})
```

Two details save time later.

`connect()` returns a handler id. Keep it if you will ever need `widget.disconnect(handler_id)` — usually because the handler outlives the thing it updates.

Some things that feel like signals are **property notifications**. A `Gtk.Switch` has no `toggled` signal; it has an `active` property, and you watch it with `notify::active`:

```
switch.connect("notify::active", lambda sw, pspec: print(sw.get_active()))
```

Property-notify handlers take the object and a `GParamSpec` — that second argument is nearly always ignored, but it has to be in the signature. Any property on any GObject can be watched this way, which is more useful than it sounds: it is how you react to a window being resized, a list selection changing, or a media player reaching the end of a file.

Callbacks run on the main thread, and while one is running the interface is frozen. Anything slow belongs on a worker thread or in an asynchronous call — see D-Bus and asynchronous work for the patterns.

The full example is `examples/gtk4/callbacks.py`.

1.3 Widgets

The rest of this chapter is a tour of the widgets almost every program uses. Each one is a runnable file under `examples/gtk4/widgets/`.

1.3.1 Buttons

A button carries a label, an icon, or both:

```
text = Gtk.Button(label="Save")

icon = Gtk.Button(icon_name="document-save-symbolic")
icon.set_tooltip_text("Save")

content = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=6)
content.append(Gtk.Image.new_from_icon_name("document-save-symbolic"))
content.append(Gtk.Label(label="Save As"))
both = Gtk.Button(child=content)
```

Icon names come from the icon theme, and the `-symbolic` suffix asks for the monochrome variant that recolours itself to match the text around it. GTK 3's stock items (`gtk.STOCK_SAVE` and friends) are gone; see Icon names for the modern names.

GTK ships CSS classes for the two buttons that need to stand out:

```
suggested = Gtk.Button(label="Confirm")
suggested.add_css_class("suggested-action")

destructive = Gtk.Button(label="Delete")
destructive.add_css_class("destructive-action")
```

Use them sparingly — one suggested action per dialog — and never use the destructive class for something that is merely annoying to undo.

The full example is `examples/gtk4/widgets/buttons.py`.

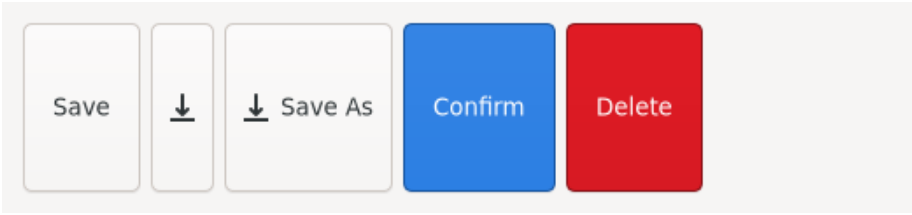

Figure 1.3: Text, icon, icon-and-label, suggested and destructive buttons

1.3.2 Toggle buttons, check buttons and radio groups

A `Gtk.ToggleButton` stays pressed. A `Gtk.CheckButton` is the same idea drawn as a tick box. Both emit `toggled`:

```
toggle = Gtk.ToggleButton(label="Toggle me")
toggle.connect("toggled", lambda b: print(b.get_active()))

check = Gtk.CheckButton(label="Check me")
check.connect("toggled", lambda b: print(b.get_active()))
```

There is no `Gtk.RadioButton` in GTK 4. A radio group is several check buttons joined with `set_group()`; a check button that belongs to a group draws itself as a radio button:

```
first = None
for name in ("Small", "Medium", "Large"):
    option = Gtk.CheckButton(label=name)
    if first is None:
        first = option
        option.set_active(True)
    else:
        option.set_group(first)
    option.connect("toggled", on_toggled)
    box.append(option)
```

Every member joins the *same* first button, not the one before it. Chaining them pairwise looks like it works and produces two groups.

Watch out for the double signal: turning one option on turns another off, so a group of three emits `toggled` twice per click. If you only care about the option that became active, check `get_active()` first and ignore the rest.

A `Gtk.Switch` is for settings that take effect immediately, and it reports through its `active` property rather than a `toggled` signal:

```
switch = Gtk.Switch()
switch.connect("notify::active", lambda sw, _p: print(sw.get_active()))
```

The rule of thumb GNOME uses: a switch turns something on now, a check button records a choice you will confirm later with a button.

The full example is `examples/gtk4/widgets/toggle-and-check.py`.

1.3.3 Labels

Labels display text and, given a little markup, do rather more:

```
label = Gtk.Label()
label.set_markup("<b>Bold</b> and <i>italic</i> and <tt>monospace</tt>")
label.set_wrap(True)
label.set_selectable(True)
label.set_xalign(0)   # left-align within whatever space it has
```

The markup is Pango markup, not HTML — a small XML-ish dialect with `<b>`, `<i>`, `<tt>`, `<span foreground="red">` and a few others. **Never build markup by concatenating strings you did not write.** A stray `&` or `<` in a filename raises a parse error and prints nothing; run untrusted text through `GLib.markup_escape_text()` first, or use `set_text()` and skip markup entirely.

A label that might contain a URL can turn it into a link on its own:

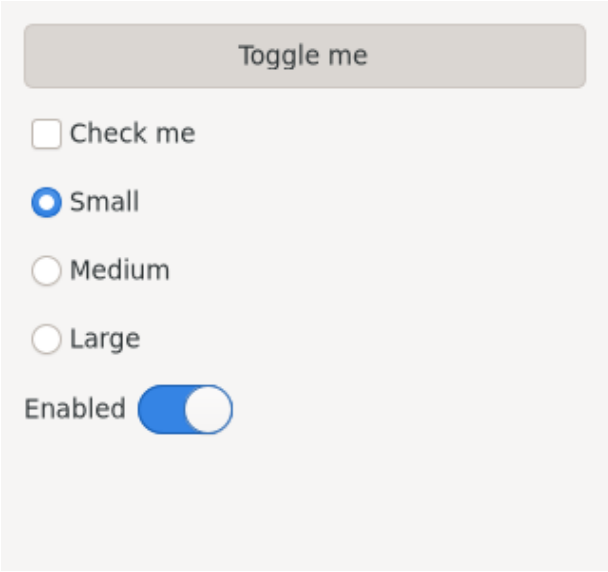

Figure 1.4: A toggle button, a check button, a radio group made of check buttons, and a switch

```
label.set_markup('<a href="https://gtk.org">The GTK website</a>')
```

1.3.4 Text entries

```
entry = Gtk.Entry(placeholder_text="Your name")
entry.connect("activate", lambda e: print("entered:", e.get_text()))
entry.connect("changed", lambda e: print("now:", e.get_text()))
```

`activate` fires when the user presses Enter; `changed` fires on every keystroke, which is right for live filtering and wrong for anything expensive.

There are specialised entries for the common cases, and they are worth using because they carry the right behaviour with them:

```
password = Gtk.PasswordEntry(show_peek_icon=True)
search = Gtk.SearchEntry()
search.connect("search-changed", lambda e: print(e.get_text()))
```

`Gtk.SearchEntry` debounces — `search-changed` waits for a pause in typing rather than firing per keystroke, which is exactly what you want before hitting a database.

Multi-line text is a different widget with a different shape. `Gtk.TextView` displays a `Gtk.TextBuffer`, and the text lives in the buffer:

```
view = Gtk.TextView(wrap_mode=Gtk.WrapMode.WORD)
view.get_buffer().set_text("Several lines of text\ngo in a GtkTextView.")

scroller = Gtk.ScrolledWindow(vexpand=True)
scroller.set_child(view)
```

Getting the text back needs the bounds, because a buffer can hand you any range:

```
buffer = view.get_buffer()
start, end = buffer.get_bounds()
text = buffer.get_text(start, end, False)
```

The last argument is whether to include invisible characters; you want `False`. A text view does not scroll on its own — put it in a `Gtk.ScrolledWindow` or it will grow until it pushes everything else off the window.

The full example is `examples/gtk4/widgets/entries.py`.

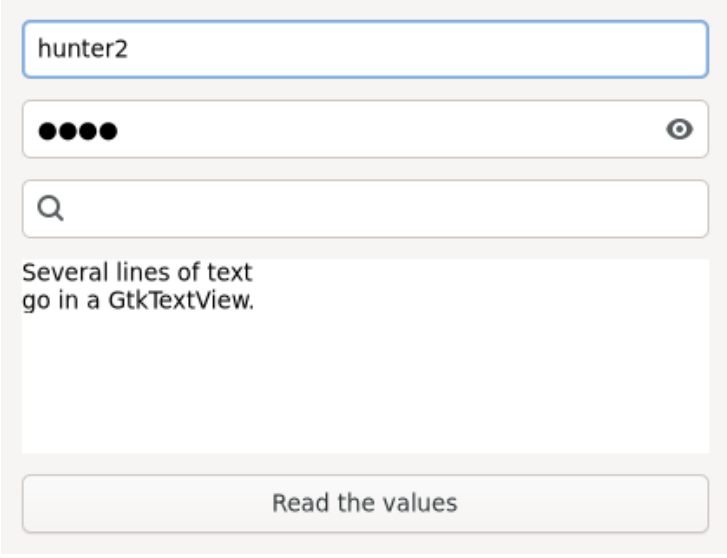

Figure 1.5: An entry, a password entry, a search entry and a text view

1.3.5 Numbers: spin buttons and scales

```
spin = Gtk.SpinButton.new_with_range(0, 100, 1)    # min, max, step
spin.set_value(25)
spin.connect("value-changed", lambda s: print(s.get_value_as_int()))

scale = Gtk.Scale.new_with_range(Gtk.Orientation.HORIZONTAL, 0, 100, 1)
scale.set_draw_value(True)
```

`get_value()` returns a float; `get_value_as_int()` saves you the rounding. Both widgets are views onto a `Gtk.Adjustment`, and sharing one adjustment between a scale and a spin button keeps them in step with no code at all:

```
adjustment = Gtk.Adjustment(lower=0, upper=100, step_increment=1, value=25)
spin = Gtk.SpinButton(adjustment=adjustment)
scale = Gtk.Scale(orientation=Gtk.Orientation.HORIZONTAL, adjustment=adjustment)
```

1.3.6 Choices: drop downs

`GtkComboBox` and `GtkComboBoxText` are deprecated. `Gtk.DropDown` replaces them, and for a fixed list of strings it is a one-liner:

```
flavours = ["Vanilla", "Chocolate", "Strawberry"]
dropdown = Gtk.DropDown.new_from_strings(flavours)
dropdown.connect(
    "notify::selected",
```

```
    lambda d, _p: print("chose:", flavours[d.get_selected()]),
)
```

`get_selected()` returns a position, not a value, and returns `Gtk.INVALID_LIST_POSITION` when nothing is selected. For anything richer than strings — a list of objects, a custom row layout, a searchable list — a drop down is backed by a list model and a factory, the same machinery that drives list views. That is More GTK 4.

The full example is `examples/gtk4/widgets/numbers-and-choices.py`.

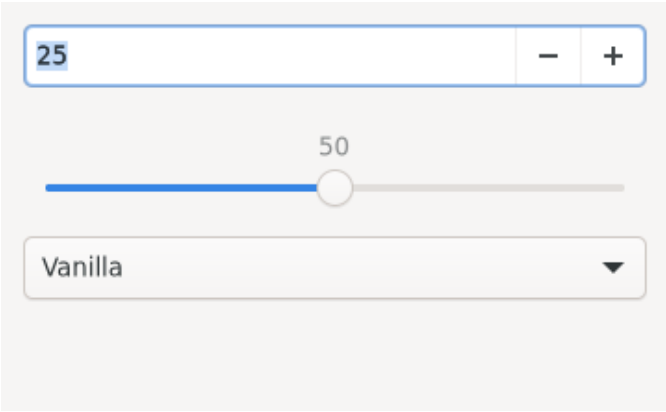

Figure 1.6: A spin button, a scale and a drop down

1.3.7 Menus and actions

Menus are the part of GTK 4 that has changed most, and the change is worth understanding because it pays off everywhere else.

In GTK 3 you built a menu out of widgets and connected each item to a callback. In GTK 4 you describe the menu as a **model** and implement the behaviour as **actions**. Nothing in the model names a function; items name actions by string.

```
def add_action(app, name, callback):
    action = Gio.SimpleAction.new(name, None)
    action.connect("activate", callback)
    app.add_action(action)


add_action(app, "new", lambda *_: print("New"))
add_action(app, "quit", lambda *_: app.quit())
app.set_accels_for_action("app.quit", ["<Control>q"])
```

Then the model:

```
menu = Gio.Menu()

file_menu = Gio.Menu()
file_menu.append("New", "app.new")
file_menu.append("Open", "app.open")

quit_section = Gio.Menu()
quit_section.append("Quit", "app.quit")
file_menu.append_section(None, quit_section)   # draws a separator
```

```
menu.append_submenu("File", file_menu)
```

Because the model is just data, the same menu can appear in more than one place:

```
app.set_menubar(menu)          # a traditional menu bar
window.set_show_menubar(True)

header = Gtk.HeaderBar()       # and a hamburger button, from the same model
header.pack_end(Gtk.MenuButton(icon_name="open-menu-symbolic", menu_model=menu))
window.set_titlebar(header)
```

The `app.` prefix on an action name says where the action lives. Actions added to the application are `app.something`; actions added to a window are `win.something`. Get the prefix wrong and the menu item is simply greyed out — GTK has no way to tell you that you meant a different scope, so a permanently insensitive menu item is nearly always a missing or misnamed action.

Set the menu bar in the **startup** handler, before any window exists:

```
app.connect("startup", on_startup)   # actions and menubar
app.connect("activate", on_activate) # windows
```

Actions carry state too, which is how you get checkboxes and radio items in a menu:

```
action = Gio.SimpleAction.new_stateful(
    "show-sidebar", None, GLib.Variant.new_boolean(True)
)
action.connect("change-state", on_change_state)
```

The payoff for all this indirection is that one action definition serves the menu bar, the popover, the keyboard shortcut, a toolbar button and D-Bus remote control at once. It is more ceremony than `connect("activate", ...)` for a single menu item and considerably less for a real application.

The full example is `examples/gtk4/widgets/menus.py`.

Figure 1.7: One menu model driving both a menu bar and a header bar button

1.3.8 Message dialogs

`GtkDialog` and `GtkMessageDialog` are deprecated as of GTK 4.10. `Gtk.AlertDialog` replaces them, and it is **asynchronous**: you hand it a callback and get control back immediately instead of running a nested main loop and blocking.

```
def ask_to_delete(window, label):
    dialog = Gtk.AlertDialog()
    dialog.set_message("Delete this file?")
```

```
    dialog.set_detail("Once it is gone it is gone. There is no undo.")
    dialog.set_buttons(["Cancel", "Delete"])
    dialog.set_cancel_button(0)     # chosen by Escape
    dialog.set_default_button(1)    # chosen by Enter
    dialog.choose(window, None, on_choice, label)


def on_choice(dialog, result, label):
    try:
        button = dialog.choose_finish(result)
    except GLib.Error:
        label.set_text("Dismissed")    # closed without answering
        return
    label.set_text("Deleted" if button == 1 else "Cancelled")
```

`choose_finish()` returns the **index** into the button list you set, and **raises `GLib.Error`** if the dialog was dismissed rather than answered. That `try` is not optional: a user pressing Escape or closing the dialog is the normal case, not an error case, and an unhandled exception in a callback will not stop your program but will fill the terminal with tracebacks.

For a notice with a single button there is nothing to wait for:

```
dialog = Gtk.AlertDialog()
dialog.set_message("Nothing happened")
dialog.show(window)
```

Passing the parent window matters. It makes the dialog modal for that window, positions it correctly, and — on Wayland — is the only way the compositor knows which window the dialog belongs to.

The same asynchronous shape covers the other dialogs: `Gtk.FileDialog`, `Gtk.ColorDialog`, `Gtk.FontDialog` all take a callback and a matching `*_finish()` that raises on dismissal. File dialogs are in More GTK 4.

The full example is `examples/gtk4/widgets/dialogs.py`.

1.3.8.1 In a libadwaita application, use libadwaita's

libadwaita 1.5. `Adw.Dialog` and `Adw.AlertDialog` need it.

`Gtk.AlertDialog` is correct and stays correct. But in an application that is already using libadwaita, `Adw.AlertDialog` is the better one, for a reason that only shows up at phone width: instead of opening a separate floating window it slides up from the bottom as a sheet inside the parent. Same code, two shapes, chosen by the space available — the breakpoints idea again, this time with nothing to configure.

```
dialog = Adw.AlertDialog(heading="Delete this file?",
                         body="Once it is gone it is gone. There is no undo.")
dialog.add_response("cancel", "Cancel")
dialog.add_response("delete", "Delete")
dialog.set_response_appearance("delete", Adw.ResponseAppearance.DESTRUCTIVE)
dialog.set_default_response("cancel")   # Enter
dialog.set_close_response("cancel")     # Escape, or dismissal
dialog.choose(parent, None, on_choice, label)
```

Two improvements on the GTK version are worth naming:

Responses are strings, not indices. `choose_finish()` gives you `"delete"`, not `1`. Insert a button and nothing silently starts meaning something else.

Dismissal is not an error. `Gtk.AlertDialog.choose_finish()` raises `GLib.Error` when the user presses Escape; `Adw.AlertDialog.choose_finish()` returns the close response you already named. There is no `try` to forget.

For a dialog with your own content in it rather than a message and buttons, subclass `Adw.Dialog`: set a title, set a child, and `present(parent)` it. `present()` takes any widget, not only a window — libadwaita walks up the tree to find where the dialog belongs. `Adw.PreferencesDialog` and `Adw.AboutDialog` are the two you get pre-built, and they replace the `Adw.PreferencesWindow` and `Adw.AboutWindow` you will see in older code.

The full example is `examples/gtk4/widgets/adw-dialogs.py`.

1.3.9 Feedback: toasts instead of a status bar

`GtkStatusbar` is deprecated and has no direct replacement, because the pattern it served — a strip of text along the bottom that nobody reads — is not one GNOME recommends any more.

For transient feedback, libadwaita has `Adw.Toast`: a message that slides in over the content, optionally with one button, and goes away on its own.

```
self.toasts = Adw.ToastOverlay()
self.toasts.set_child(content)

toast = Adw.Toast.new("Saved")
toast.set_button_label("Undo")
toast.connect("button-clicked", lambda _t: print("undo"))
self.toasts.add_toast(toast)
```

The overlay wraps your content; the toasts appear on top of it. For persistent status — a word count, a connection indicator — put a label in the header bar or a bottom bar, where it belongs to the layout rather than floating over it.

1.4 Libadwaita

Everything so far has been plain GTK 4. In practice you will write libadwaita, and the difference is mostly that you stop building layouts by hand.

```
import gi

gi.require_version("Gtk", "4.0")
gi.require_version("Adw", "1")
from gi.repository import Adw, Gtk


class Window(Adw.ApplicationWindow):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.set_title("Adwaita Window")
        self.set_default_size(420, 260)

        self.toasts = Adw.ToastOverlay()

        page = Adw.PreferencesPage()
        group = Adw.PreferencesGroup(title="Settings")
        group.add(Adw.SwitchRow(title="Enabled", subtitle="Flip me"))
        group.add(Adw.EntryRow(title="Your name"))
```

```
        page.add(group)
        self.toasts.set_child(page)

        toolbar = Adw.ToolbarView()
        toolbar.add_top_bar(Adw.HeaderBar())
        toolbar.set_content(self.toasts)

        self.set_content(toolbar)


app = Adw.Application(application_id="com.example.AdwaitaWindow")
app.connect("activate", lambda a: Window(application=a).present())
sys.exit(app.run(sys.argv))
```

That window has a proper header bar, correctly spaced rows with titles and subtitles, and a layout that adapts to a phone-sized window — with no margins, alignments or box nesting written by hand. `Adw.SwitchRow` is a label, a subtitle and a switch that already agree about spacing.

Three differences to watch for when you switch:

`Adw.Application` instead of `Gtk.Application` — it initialises the libadwaita stylesheet. Using `Gtk.Application` with libadwaita widgets gives you widgets that work and look wrong.

`Adw.ApplicationWindow` uses `set_content()`, not `set_child()`, and it has no separate titlebar area — the header bar goes inside the content, usually in an `Adw.ToolbarView`. This trips up everyone once.

Subclassing is the normal style here. Once a window owns state — a toast overlay, a list model, a file being edited — a class is easier to follow than a nest of closures, and it is what the GNOME examples look like.

The full example is `examples/gtk4/adwaita-window.py`.

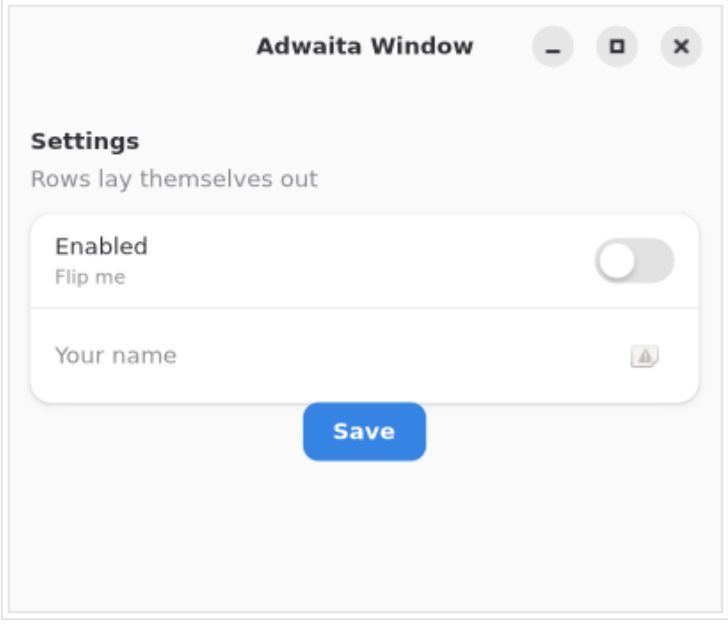

Figure 1.8: The same widgets as libadwaita rows, with no hand-written spacing

1.4.1 Adaptive layouts

libadwaita 1.4. `Adw.Breakpoint` needs it.

A window is not one width. The user drags it narrow, tiles it beside something else, or runs your application on a phone, and a layout built for 1200 pixels has to become a layout that works in 400. Doing that by hand — a `notify::width` handler, some arithmetic, a pile of `set_visible()` calls, and a bug where you forget to put something back — is the wrong approach, and libadwaita replaces it with a declarative one.

A **breakpoint** is a condition plus a list of properties to set while it holds:

```
narrow = Adw.Breakpoint.new(Adw.BreakpointCondition.parse("max-width: 600sp"))
narrow.add_setter(self.split, "collapsed", True)
narrow.add_setter(self.show_sidebar_button, "visible", True)
self.add_breakpoint(narrow)
```

There is no handler and no arithmetic. While the window is 600sp or narrower the split view is collapsed and the button is visible; when it stops being narrow libadwaita **puts the old values back**. That last part is what you get wrong writing it yourself, because it means remembering what every property was before you changed it.

Two details in that snippet earn their place:

`sp`, not `px`. A scalable pixel is a pixel at the user's text scaling factor. Someone running at 125% text size gets a breakpoint that fires at a proportionally larger real width, which is what you want — the layout is tight because the *content* does not fit, and their content is bigger. Conditions can also be written on height, and combined: `max-width: 600sp and max-height: 400sp`.

600sp is the conventional divide between a phone-shaped window and a desktop-shaped one. Use it unless your content has an opinion.

The widget being collapsed here is `Adw.OverlaySplitView`, which is a sidebar beside the content when there is room and a sidebar sliding over the content when there is not. Its siblings are worth knowing by name, because between them they cover most of what a window needs to do at two sizes:

`Adw.OverlaySplitView` Sidebar and content. Collapses to an overlay.

`Adw.NavigationSplitView` The same, but collapses into a navigation stack — the sidebar becomes a page you go *back* from, which is the right model for a list that leads to a detail view.

`Adw.ViewSwitcher` and `Adw.ViewSwitcherBar` Tabs in the header bar when wide; a bar along the bottom, thumb-reachable, when narrow. Set the bar's `reveal` from a breakpoint and hide the header one.

`Adw.ToolbarView` Not adaptive itself, but it is what lets a breakpoint move a toolbar from the top to the bottom without rebuilding anything.

The full example is `examples/gtk4/adaptive-window.py`. Resize the window past 600sp and the sidebar changes behaviour with no code running in between.

1.4.2 Keyboard shortcuts

Menus give you accelerators nearly free, and the reason to do it through actions rather than key handlers is that the action already exists and the menu item already names it:

```
app.set_accels_for_action("app.save", ["<Control>s"])
app.set_accels_for_action("app.shortcuts", ["<Control>question", "F1"])
```

The plural is not decoration — an action can have several accelerators, which is how one command answers to both a modern and a legacy key.

For a key that only means something while part of the window has focus, attach a `Gtk.ShortcutController` to that widget instead:

```
controller = Gtk.ShortcutController()
controller.set_scope(Gtk.ShortcutScope.LOCAL)
controller.add_shortcut(Gtk.Shortcut(
    trigger=Gtk.ShortcutTrigger.parse_string("<Control>d"),
    action=Gtk.CallbackAction.new(self.on_duplicate)))
self.add_controller(controller)
```

`Gtk.CallbackAction` is the escape hatch that runs a Python function, and the function **must return `True`** to say it handled the key. Return `False` or fall off the end and GTK keeps looking for someone else to handle it, which presents as a shortcut that works only sometimes.

The third part of the job is telling the user the keys exist. `Adw.ShortcutsDialog` (libadwaita 1.8) is the standard window for that, bound to `<Control>question` by convention:

```
dialog = Adw.ShortcutsDialog()
section = Adw.ShortcutsSection(title="General")
section.add(Adw.ShortcutsItem(title="Save", accelerator="<Control>s"))
dialog.add(section)
dialog.present(window)
```

Keep the accelerator strings in one table and build the menu, the bindings and the dialog from it. Three hand-maintained copies of the same list is three chances for the help window to start lying.

The full example is `examples/gtk4/shortcuts.py`.

1.5 Summary

You can now build a GTK 4 application: an application object with an activate handler, a window with a box in it, widgets connected to callbacks, a menu driven by actions, and dialogs that ask questions without blocking.

The things to carry forward:

- `gi.require_version()` before every `from gi.repository import ...`.
- `Gtk.Application` owns the main loop; `activate` builds windows, `startup` builds actions and menus.
- Widgets are visible by default; single-child containers use `set_child()`.
- `expand` and `align` live on the child, not on the `append()` call.
- Radio buttons are grouped check buttons.
- Menus are models plus actions, and the action prefix (`app.` or `win.`) matters.
- Dialogs are asynchronous, and their `*_finish()` methods raise when dismissed. `Adw.AlertDialog` uses string responses and does not raise on dismissal.
- Reach for libadwaita before you reach for a hand-built layout.
- Adaptive layout is a breakpoint — a condition and some property setters — not a resize handler. Measure in `sp`, and 600sp is the usual divide.
- Bind keys to actions with `set_accels_for_action`; use a `Gtk.ShortcutController` only for keys that are local to a widget, and return `True` from its callback.
- A window cannot position itself. Wayland decides, which is why every dialog gets a parent.

GObject is next: properties, signals and bindings, which are what the list widgets in the chapter after it are built on.

If you are porting an existing PyGTK program, Migrating from PyGTK is a translation table for the idioms this chapter replaced.

Chapter 2

GObject: Properties, Signals and Bindings

> Every listing in this chapter is a file under `examples/gtk4/gobject/`. They are run on each build, so if one of them stops working the build says so.

2.1 Introduction

GTK is written in C, and C has no objects. GObject is the object system that was built to give it some — classes, inheritance, virtual methods, properties, signals and runtime type information, in plain C.

You could reasonably ask why a Python programmer should care. Python already has all of that.

The answer is that **GObject's versions of those things are the ones GTK can see**. A Python attribute is invisible to GTK. A GObject property can be watched for changes, bound to another object's property, read by a `Gtk.Expression`, sorted and filtered on, set from a `.ui` file, and animated by libadwaita. Everything in More GTK 4 — list models, sorters, filters — is built on properties, and none of it works on a plain attribute.

This chapter is the fifteen minutes that makes the rest of Part I stop feeling arbitrary.

2.2 Subclassing

```
from gi.repository import GObject


class Person(GObject.Object):
    __gtype_name__ = "Person"

    def __init__(self, name=""):
        super().__init__()
        self.name = name
```

Two things are different from an ordinary Python class.

`super().__init__()` is not optional, and it has to run before anything touches a property. GObject construction happens on the C side; skipping it leaves a half-built object that fails later and confusingly.

`__gtype_name__` gives the type a name on the C side. Without it PyGObject invents one from the class name, which is fine until two classes in one process end up with the same invented name — at which

point registration fails outright. Set it, make it unique, and prefix it with something of your own in a library.

You subclass GTK widgets the same way, which is how `class Window(Adw.ApplicationWindow)` works in every other chapter.

2.3 Properties

```
class Person(GObject.Object):
    __gtype_name__ = "Person"

    name = GObject.Property(type=str, default="")
    age = GObject.Property(type=int, default=0, minimum=0, maximum=150)
```

That is the whole declaration. `name` and `age` now read and write like ordinary attributes:

```
person.age = 37
print(person.name)
```

and also by name, which is the part that matters:

```
person.get_property("name")
person.set_property("age", 37)
```

Anything generic — a binding, a sorter, a `.ui` file, an animation — reaches your data through that second form.

2.3.1 Naming

A property declared as `date_of_birth` is called **`date-of-birth`** by GTK. Underscores become hyphens at the boundary. Python code uses the underscore spelling, and everything taking a property *name* as a string — `bind_property`, `notify::`, `Gtk.PropertyExpression` — uses hyphens. Both spellings are accepted in most places; hyphens are what the documentation uses.

2.3.2 Computed and validating properties

For a property that is derived, or that has to check or normalise what it is given, use the decorator form:

```
@GObject.Property(type=str)
def nickname(self):
    return self._nickname

@nickname.setter
def nickname(self, value):
    self._nickname = value.strip().title()
```

A read-only property is one with no setter and the `READABLE` flag:

```
@GObject.Property(type=str, flags=GObject.ParamFlags.READABLE)
def summary(self):
    return f"{self.name}, aged {self.age}"
```

A derived property has to be **told** when its inputs change, or nothing watching it will ever hear:

```
self.connect("notify::name", lambda *_: self.notify("summary"))
self.connect("notify::age", lambda *_: self.notify("summary"))
```

2.3.3 Watching for changes

Every property emits `notify::<name>` when it changes:

```
person.connect("notify::age", lambda obj, pspec: print(obj.age))
```

The handler takes the object and a `GParamSpec` describing the property. You will ignore the second argument almost every time, and it still has to be in the signature.

This is the same mechanism as the `notify::selected`, `notify::active` and `notify::timestamp` handlers in earlier chapters. It is not a special case for widgets — it is how every GObject reports change.

2.3.4 How they fail

The two failure modes are not symmetrical, and one of them is easy to miss:

```
person.age = 300        # out of range
person.age = "thirty"   # wrong type
```

Out of range warns and leaves the old value in place. It is not clamped, and nothing is raised — you get a `Warning` on stderr and `person.age` is still whatever it was. In a program that already prints warnings, this is invisible.

The wrong type raises `TypeError`, because the conversion cannot even be attempted.

So a range on a property is a good assertion and a bad validator. If a value coming from outside your program has to be inside a range, check it yourself before assigning.

The full example is `examples/gtk4/gobject/properties.py`.

2.4 Signals

Widgets emit signals; so can your own objects. A custom signal is right whenever a component needs to announce something without knowing who is listening — which keeps a component from having to hold a reference to everything that cares about it.

```
class Download(GObject.Object):
    __gtype_name__ = "Download"

    @GObject.Signal(arg_types=(str,))
    def started(self, url):
        print(f"started {url}")

    finished = GObject.Signal()
```

`finished` is a signal with no arguments and no body. `started` takes a string, and its method body is the signal's **default handler**.

```
download.emit("started", "https://example.com/f")
download.connect("started", lambda _d, url: ...)
```

2.4.1 The default handler runs first

With no flags, a signal is `RUN_FIRST`: the method body runs **before** anything connected with `connect()`. That surprises people who expect the class's own behaviour to be the fallback. If you want it to be a fallback, say so:

```
@GObject.Signal(flags=GObject.SignalFlags.RUN_LAST)
```

2.4.2 Return values and vetoes

A signal can return a value, and an **accumulator** decides what happens when several handlers each return one. The useful one is `signal_accumulator_true_handled`: emission stops at the first handler returning `True`, and that becomes the result.

```
@GObject.Signal(return_type=bool, arg_types=(str,),
                flags=GObject.SignalFlags.RUN_LAST,
                accumulator=GObject.signal_accumulator_true_handled)
def confirm_overwrite(self, filename):
    return False        # nobody objected


if self.emit("confirm_overwrite", filename):
    return              # somebody did
```

That is the veto pattern, and it is exactly how GTK's own `delete-event`-style signals work: any listener can say "I have handled this, stop".

With no listeners connected, the default handler's `False` stands — so the sensible behaviour is the one you get for free.

2.4.3 Disconnecting

```
handler = obj.connect("finished", on_finished)
obj.disconnect(handler)
```

Worth caring about for one reason: **a connected handler holds a reference**. If a long-lived object holds a handler that closes over a window, that window cannot be finalised when it closes. The symptom is a program whose memory grows every time a dialog is opened.

The rule of thumb: connecting a widget to itself, or to something it owns, needs no thought. Connecting a *short-lived* object to a *long-lived* one means keeping the handler id and disconnecting when the short-lived one goes away.

The full example is `examples/gtk4/gobject/signals.py`.

2.5 Bindings

`bind_property()` is the most under-used thing in GObject. Any time you would write "when this changes, set that", a binding is shorter, cannot get the two out of sync, and cleans itself up when either object is finalised.

```
switch.bind_property("active", settings, "enabled",
                     GObject.BindingFlags.SYNC_CREATE
                     | GObject.BindingFlags.BIDIRECTIONAL)
```

The flags are the whole API:

`SYNC_CREATE` Copy the current value across immediately. **Almost always what you want.** Without it the target keeps whatever it had until the source next changes, which is the usual cause of "my binding does not work" on the first frame.

`BIDIRECTIONAL` Changes flow both ways.

`INVERT_BOOLEAN` For the commonest transform of all.

For anything else, supply a transform function. It returns a `(handled, value)` pair, and returning `False` rejects the value and leaves the target alone:

```
def to_percent(_binding, value):
    return True, f"{value * 100:.0f}%"

settings.bind_property("volume", label, "label",
                       GObject.BindingFlags.SYNC_CREATE, to_percent)
```

One property can drive several targets, which is how a single `enabled` setting greys out four widgets with no handler at all.

2.5.1 Not everything is a property

A method called `get_value()` does not imply a property called `value`. The one that catches everyone:

```
scale.bind_property("value", settings, "volume", ...)
# TypeError: Cannot create binding from <Gtk.Scale ...>.value
```

A `Gtk.Scale` keeps its value in its `Gtk.Adjustment`, and that is what the binding attaches to:

```
scale.get_adjustment().bind_property("value", settings, "volume", ...)
```

Similarly `Gtk.WebView.can_go_back()` in Embedding Web Content is a method with no property behind it. When a binding fails with "cannot create binding", the property does not exist under that name — check the API reference, or ask the object:

```
print([p.name for p in obj.list_properties()])
```

2.5.2 Keeping the binding

`bind_property()` returns a `GObject.Binding`. You can ignore it — the binding lives as long as both objects — but keep it if it has to be taken apart early:

```
binding = source.bind_property(...)
binding.unbind()
```

The full example is `examples/gtk4/gobject/bindings.py`.

2.6 Why the list widgets needed all this

Now the shape of More GTK 4 should make sense.

A `Gio.ListStore` holds GObjects because the machinery around it — sorters, filters, expressions — reaches values through the property system. A `Gtk.PropertyExpression.new(Package, None, "name")` is a compiled instruction to read the `name` property, evaluated in C over a million rows without Python being involved. The `bind_property` in a list factory's `bind` handler keeps a check button and a data object in step with no callback. And a row updates when the underlying object changes because the object emits `notify`.

None of it can see `self.name = name`. All of it can see `name = GObject.Property(type=str)`.

2.7 Summary

- `super().__init__()` first, and set `__gtype_name__`.
- `GObject.Property` is what GTK can see; a plain attribute is not.
- `date_of_birth` in Python is `date-of-birth` to anything taking a name.
- Every property emits `notify::<name>`; derived properties must call `notify()` themselves.
- Out-of-range assignment warns and keeps the old value; the wrong type raises.

- A signal's default handler runs *first* unless you ask for `RUN_LAST`.
- `signal_accumulator_true_handled` gives you a veto.
- Keep handler ids when a short-lived object connects to a long-lived one.
- `bind_property` with `SYNC_CREATE` replaces most "when this changes" handlers.
- "Cannot create binding" means the property does not exist under that name.

More GTK 4 is next, and now the list machinery has a reason for being the shape it is.

Chapter 3

More GTK 4

> Every listing in this chapter is a file under `examples/gtk4/`. They are run on each build, so if one of them stops working the build says so.

3.1 Introduction

Getting Started with GTK 4 covered the widgets a window is made of. This chapter covers the machinery around them: showing a list of things, letting the user pick a file, moving data around by dragging it, putting a picture on screen, and describing an interface in a file instead of in code.

The list section is the long one, and it is worth the time. GTK 4's list widgets look intimidating next to `gtk.ListStore` and turn out to be simpler once the pieces have names — and they are the same pieces that drive drop downs, grid views and the sidebar of every GNOME application.

3.2 Lists and tables

`GtkTreeView`, `GtkListStore` and `GtkTreeStore` are deprecated in GTK 4. They are replaced by a set of small parts that snap together:

A list model holds the data. `Gio.ListStore` is the usual one. It holds GObjects — not tuples, not strings — so each row is an object with properties.

A factory builds row widgets and fills them in. `Gtk.SignalListItemFactory` does this with two signals: `setup` creates an empty row, `bind` points an existing row at an item.

A selection model wraps the list model and tracks what is selected. `Gtk.SingleSelection`, `Gtk.MultiSelection`, `Gtk.NoSelection`.

A view draws it. `Gtk.ListView` for a list, `Gtk.ColumnView` for a table, `Gtk.GridView` for tiles.

They chain: store → (filter) → (sort) → selection → view.

The reason for all these parts is **recycling**. A `GtkTreeView` built a row widget for every row you had. A `GtkListView` builds enough row widgets to fill the visible area and reuses them as you scroll, which is why it can hold a million items without noticing. `setup` runs once per *widget*; `bind` runs every time a widget is pointed at a different *item*.

3.2.1 The item type

Rows are objects, so start by defining one:

```
class Task(GObject.Object):
    __gtype_name__ = "Task"
```

```
    title = GObject.Property(type=str, default="")
    done = GObject.Property(type=bool, default=False)

    def __init__(self, title, done=False):
        super().__init__(title=title, done=done)
```

`GObject.Property` rather than a plain attribute, and `__gtype_name__` so the type has a name on the C side. Both matter: property expressions, sorters, filters and `bind_property` all reach data through the GObject property system, and none of them can see a plain Python attribute.

A plain Python attribute still works for anything you only ever read in Python. Use properties for the values the view needs to know about.

3.2.2 A list

```
store = Gio.ListStore(item_type=Task)
for title in ("Buy milk", "Write chapter two", "Walk the dog"):
    store.append(Task(title))

factory = Gtk.SignalListItemFactory()
factory.connect("setup", on_setup)
factory.connect("bind", on_bind)

selection = Gtk.SingleSelection(model=store)
view = Gtk.ListView(model=selection, factory=factory)
```

The factory handlers:

```
def on_setup(_factory, list_item):
    box = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=12)
    box.append(Gtk.CheckButton())
    box.append(Gtk.Label(xalign=0))
    list_item.set_child(box)


def on_bind(_factory, list_item):
    task = list_item.get_item()
    box = list_item.get_child()
    check, label = box.get_first_child(), box.get_last_child()
    label.set_text(task.title)
    check.set_active(task.done)
```

Two rules follow from recycling, and breaking either produces the same symptom — data from the wrong row appearing in a row:

Never create widgets in `bind`. It runs on every scroll. Create in `setup`, fill in `bind`.

Undo in `unbind` whatever you did in `bind`. If `bind` connects a signal or creates a binding, the row keeps it when it is recycled onto a different item. That is what `unbind` is for:

```
def on_bind(_factory, list_item):
    ...
    list_item.binding = check.bind_property(
        "active", task, "done", GObject.BindingFlags.BIDIRECTIONAL
    )
```

```
def on_unbind(_factory, list_item):
    binding = getattr(list_item, "binding", None)
    if binding is not None:
        binding.unbind()
        list_item.binding = None
```

`bind_property` is worth knowing on its own: it keeps two GObject properties in sync with no callback at all. Here, ticking the check button writes straight through to `task.done`, and changing `task.done` in code moves the check button.

A list view scrolls, but it does not scroll itself — put it in a `Gtk.ScrolledWindow`.

The full example is `examples/gtk4/list-view.py`.

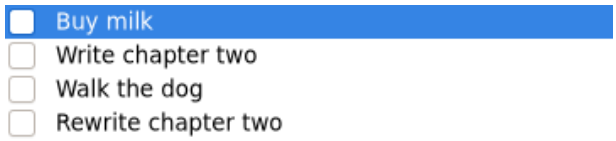

Figure 3.1: A list view over a Gio.ListStore

3.2.3 A sortable table

`Gtk.ColumnView` is a list view with columns. Each column has its own factory, so each cell in that column is built and filled the same way:

```
factory = Gtk.SignalListItemFactory()
factory.connect("setup", setup)
factory.connect("bind", bind)

column = Gtk.ColumnViewColumn(title="Name", factory=factory)
column.set_expand(True)
view.append_column(column)
```

Sorting is done by the model, not the view. A column gets a sorter; the view combines the sorters of whichever column headers were clicked; a `Gtk.SortListModel` sorts the data through that combined sorter:

```
expression = Gtk.PropertyExpression.new(Package, None, "name")
column.set_sorter(Gtk.StringSorter(expression=expression))
...
sorted_model = Gtk.SortListModel(model=store)
```

```
sorted_model.set_sorter(view.get_sorter())
```

A `Gtk.Expression` is a compiled way of reading a value out of an object. `Gtk.PropertyExpression.new(Package, None, "name")` means "the `name` property of a `Package`", and it is evaluated in C, so sorting a large list does not run Python per comparison. Use `Gtk.NumericSorter` for numbers — a `Gtk.StringSorter` on a size column will happily sort 1,000 before 900.

Filtering works the same way, one model further up the chain:

```
text_filter = Gtk.StringFilter(
    expression=Gtk.PropertyExpression.new(Package, None, "name"),
    match_mode=Gtk.StringFilterMatchMode.SUBSTRING,
)
filtered = Gtk.FilterListModel(model=store, filter=text_filter)

search = Gtk.SearchEntry()
search.connect("search-changed", lambda e: text_filter.set_search(e.get_text()))
```

The whole chain is store → filter → sort → selection → view, and each link is a list model in its own right. Nothing copies the data; each wrapper is a view onto the one below. That is why you can filter a list while a selection is live and get sensible behaviour.

For a filter that Python has to decide, use `Gtk.CustomFilter`:

```
recent = Gtk.CustomFilter.new(lambda item: item.size > 5_000)
```

The full example is `examples/gtk4/column-view.py`.

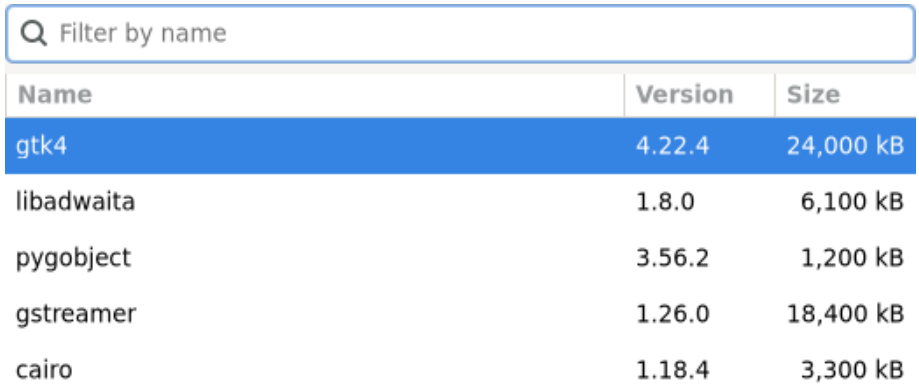

Figure 3.2: A column view, sortable by header and filtered by the search entry

3.2.4 Choosing between the views

`Gtk.ListView` one column of rows, uniform height, arbitrarily long.
`Gtk.ColumnView` a table with sortable, resizable columns.
`Gtk.GridView` tiles, for thumbnails.
`Gtk.ListBox` rows you add as widgets, with no factory and no recycling. Fine for a settings page or a sidebar of a dozen items; wrong for anything that grows. libadwaita's rows (`Adw.ActionRow`, `Adw.SwitchRow`, `Adw.EntryRow`) go in one of these.

If the list is short and fixed, `Gtk.ListBox` is much less code. If it comes from data, use a list view.

3.3 File dialogs

`GtkFileChooserDialog` is deprecated. `Gtk.FileDialog` replaces it, with the same asynchronous shape as `Gtk.AlertDialog`:

```
dialog = Gtk.FileDialog(title="Open a file")
dialog.set_filters(text_filters())
dialog.open(window, None, on_opened, label)


def on_opened(dialog, result, label):
    try:
        file = dialog.open_finish(result)
    except GLib.Error:
        return          # cancelled
    print(file.get_path())
```

There is `open()`, `save()`, `select_folder()` and their plural forms (`open_multiple()`), each with a matching `*_finish()`. All of them raise `GLib.Error` when the user cancels.

Filters are a list model of `Gtk.FileFilter`:

```
filters = Gio.ListStore(item_type=Gtk.FileFilter)

text = Gtk.FileFilter()
text.set_name("Text files")
text.add_mime_type("text/plain")
text.add_suffix("txt")
filters.append(text)

dialog.set_filters(filters)
```

3.3.1 You get a GFile, not a path

`open_finish()` returns a `Gio.File`. It is tempting to call `get_path()` and hand the string to `open()`, and for a local file that works. It is worth resisting:

```
ok, contents, _etag = file.load_contents(None)
text = contents.decode("utf-8", "replace")

file.replace_contents(b"...", None, False, Gio.FileCreateFlags.NONE, None)
```

`get_path()` returns `None` for anything that is not a local file — a document on a remote share, an attachment, a file handed over by the desktop portal. Under Flatpak, `Gtk.FileDialog` is answered by the **file portal**: the user picks a file in a dialog drawn by the desktop, and your sandbox is granted access to just that one file. `Gio.File` handles all of it; a raw path does not.

The full example is `examples/gtk4/file-dialog.py`.

3.4 Drag and drop

Drag and drop moved onto event controllers along with the rest of input handling. A widget that can be dragged gets a `Gtk.DragSource`; a widget that can receive gets a `Gtk.DropTarget`. There are no `drag_source_set()` calls and no `drag-data-received` signal.

The source says what is being dragged by returning a content provider:

```
def on_prepare(_source, _x, _y):
    value = GObject.Value(str, text)
    return Gdk.ContentProvider.new_for_value(value)


source = Gtk.DragSource(actions=Gdk.DragAction.COPY)
source.connect("prepare", on_prepare)
label.add_controller(source)
```

Returning `None` from `prepare` refuses the drag, which is how you make some rows draggable and others not.

Give the drag something visible to carry, or the pointer drags nothing at all:

```
def on_drag_begin(source, _drag):
    source.set_icon(Gtk.WidgetPaintable.new(label), 0, 0)
```

The target declares the type it accepts and returns `True` from `drop` if it took the data:

```
def on_drop(_target, value, _x, _y):
    label.set_text(f"Got: {value}")
    return True


target = Gtk.DropTarget.new(GObject.TYPE_STRING, Gdk.DragAction.COPY)
target.connect("drop", on_drop)
frame.add_controller(target)
```

`enter` and `leave` are where the highlight goes — a drop target that gives no feedback feels broken even when it works.

Because everything travels as a `GValue`, dragging your own objects between two lists in your own program needs no serialisation: `Gdk.ContentProvider.new_for_value(GObject.Value(Task, task))`. Dragging to another application needs a type that application understands: `GObject.TYPE_STRING` for text, `Gio.File` for files.

The full example is `examples/gtk4/drag-and-drop.py`.

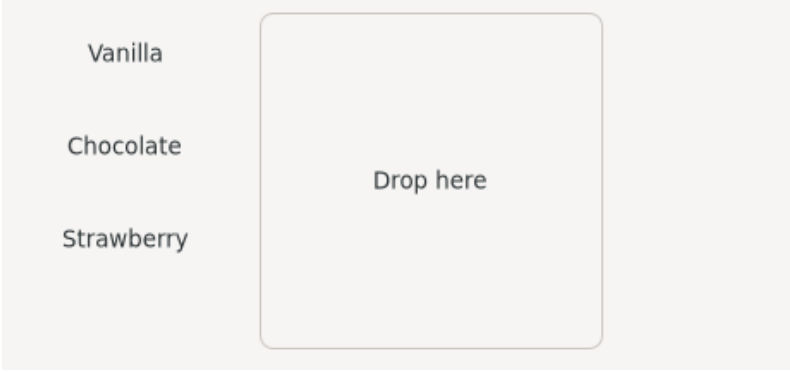

Figure 3.3: Drag sources on the left, a drop target on the right

3.5 Copy and paste

The clipboard is the same content machinery as drag and drop, which is why it belongs here rather than three chapters away. It hangs off the display, so you ask a widget for it:

```
self.get_clipboard().set(self.entry.get_text())
```

`set()` takes a Python value and works out which content formats to advertise. Underneath it is `set_content()` with a `Gdk.ContentProvider`, which is what you use when you want to offer several formats at once — plain text *and* HTML, say — built with `Gdk.ContentProvider.new_union()`.

Reading is asynchronous, and there is no synchronous version to wish for:

```
def on_paste(self, _button):
    self.get_clipboard().read_text_async(None, self.on_text_ready)


def on_text_ready(self, clipboard, result):
    try:
        text = clipboard.read_text_finish(result)
    except GLib.Error as error:
        self.status.set_text(f"nothing to paste ({error.message})")
        return
    self.status.set_text(text)
```

That is not API pedantry. On Wayland the clipboard's contents live in whichever application owns the selection; fetching them is inter-process communication with a program that may be busy, may be slow, and may exit halfway through. GTK 3's `gtk_clipboard_wait_for_text()` blocked the main loop on exactly that, and it is gone. The `try` is the normal path, not the exceptional one: an empty clipboard, or one holding a format you did not ask for, arrives as a `GLib.Error`.

`read_texture_async()` is the same shape for images, and `read_value_async()` for anything else with a registered type.

There are two clipboards on Linux. `get_clipboard()` is the ordinary one that Ctrl+C fills. `get_primary_clipboard()` is the primary selection — filled just by selecting text, pasted with middle-click. Text widgets already maintain the primary selection for you; the thing not to do is write to it on Ctrl+C, which overwrites a selection the user made somewhere else.

The full example is `examples/gtk4/clipboard.py`.

3.6 Images and pictures

There are two widgets, and picking the wrong one is the usual cause of an image that refuses to grow past 16 pixels.

`Gtk.Image` is for **icons**. It sizes itself from the icon theme and ignores how much room it has:

```
icon = Gtk.Image.new_from_icon_name("dialog-information-symbolic")
icon.set_pixel_size(48)
```

`Gtk.Picture` is for **content**. It scales to the space it is given:

```
texture = Gdk.Texture.new_from_filename("sample.jpg")
picture = Gtk.Picture.new_for_paintable(texture)
picture.set_content_fit(Gtk.ContentFit.CONTAIN)
picture.set_vexpand(True)
```

`GdkPixbuf` still exists and still loads files, but `Gdk.Texture` is the modern path: decoded once, held on the GPU, drawn without a copy per frame. Use `Gdk.Texture.new_from_filename()` or `new_from_resource()` unless you need to manipulate pixels, in which case load a pixbuf and convert with `Gdk.Texture.new_for_pixbuf()`.

`set_content_fit()` takes `CONTAIN` (fit inside, keep aspect), `COVER` (fill, crop), `FILL` (stretch) or `SCALE_DOWN` (never enlarge).

The full example is `examples/gtk4/images.py`.

Figure 3.4: A themed icon above a photograph in a Gtk.Picture

3.7 Tooltips

```
button.set_tooltip_text("Save the document")
button.set_tooltip_markup("Save the <b>document</b>")
```

That is the whole API for the common case. Tooltips belong on icon-only buttons, where they are the only thing naming the action, and they should say what the control does rather than repeating its label.

For a tooltip whose text depends on what is under the pointer, set `has-tooltip` and handle `query-tooltip`.

3.8 Building interfaces from UI files

Writing a layout in Python is fine until it is a hundred lines of `append()`. The alternative is a `.ui` file: GTK's own XML, loaded by `Gtk.Builder`.

Glade is not an option any more. It never gained GTK 4 support and is unmaintained. What replaced it:

- Write the XML by hand. It is verbose but obvious, and it is what the other tools produce.
- **Blueprint** — a compact language that compiles to `.ui`. Most new GNOME applications use it.
- **Cambalache** — a graphical designer that does support GTK 4 and libadwaita.

The libglade format and `gtk-builder-convert` are long gone; if you have a `.glade` file from the GTK 2 days, it is a rewrite rather than a conversion.

3.8.1 GtkTemplate

You can load a `.ui` file and pull widgets out of it by id, but the pleasant way is `Gtk.Template`, which binds a Python class to a `<template>` element:

```
<interface>
  <requires lib="gtk" version="4.0"/>
  <template class="ExampleWindow" parent="GtkApplicationWindow">
    <property name="title">Built from a UI file</property>
    <child type="titlebar">
      <object class="GtkHeaderBar"/>
    </child>
    <property name="child">
      <object class="GtkBox">
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkEntry" id="name_entry">
            <property name="placeholder-text" translatable="yes">Your name</property>
          </object>
        </child>
        <child>
          <object class="GtkButton" id="greet_button">
            <property name="label" translatable="yes">Greet me</property>
            <signal name="clicked" handler="on_greet_clicked" swapped="no"/>
          </object>
        </child>
        <child>
          <object class="GtkLabel" id="greeting"/>
        </child>
      </object>
    </property>
  </template>
</interface>
```

```
@Gtk.Template(filename=str(UI_FILE))
class ExampleWindow(Gtk.ApplicationWindow):
    __gtype_name__ = "ExampleWindow"

    name_entry = Gtk.Template.Child()
    greet_button = Gtk.Template.Child()
    greeting = Gtk.Template.Child()

    @Gtk.Template.Callback()
    def on_greet_clicked(self, _button):
        name = self.name_entry.get_text().strip() or "stranger"
        self.greeting.set_text(f"Hello, {name}!")
```

Three names have to agree, and when they do not the error message is not always helpful:

- `__gtype_name__` on the class must equal the `class` attribute of `<template>`.
- Each `Gtk.Template.Child()` attribute name must equal an `id` in the file.
- Each `handler` named in a `<signal>` must exist on the class and be decorated with `@Gtk.Template.Callback()`.

`translatable="yes"` marks a string for translation, and `xgettext` reads it straight out of the XML — see Internationalization.

Shipping the `.ui` file next to the `.py` file works while you develop. For anything you install, compile

the files into a **GResource** bundle and load them with `Gtk.Template(resource_path=...)`: one file to install, and the data is compiled into the binary rather than looked up on disk. That is covered with the rest of installation in the packaging chapter.

The full example is `examples/gtk4/builder/`.

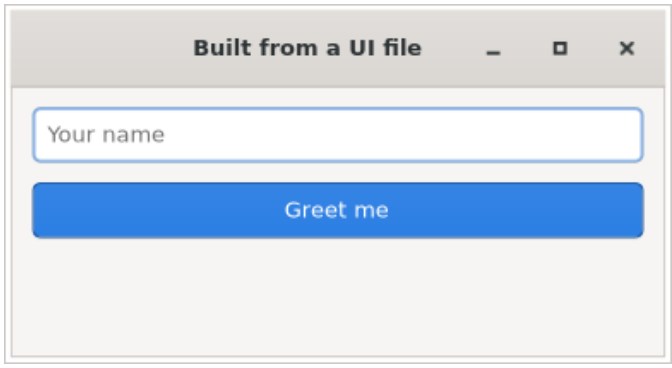

Figure 3.5: A window whose layout came from a .ui file

3.9 Notifications, not status icons

`GtkStatusIcon` is gone, and the system tray it drew into is not part of GNOME. If your program needs to say "something finished" while the user is looking at something else, send a notification:

```
notification = Gio.Notification.new("Export finished")
notification.set_body("holiday-photos.zip is ready in your Downloads folder.")
notification.set_icon(Gio.ThemedIcon.new("document-save-symbolic"))
notification.add_button("Show it", "app.reveal")
notification.set_default_action("app.reveal")

app.send_notification("export-done", notification)
```

The buttons name actions, exactly as menu items do — which means the notification can be acted on after your program has exited, and the desktop will start it again to deliver the action.

The string id is a handle: sending again with the same id **replaces** the earlier notification rather than stacking a second one, and `app.withdraw_notification("export-done")` takes it back.

One catch that costs people an afternoon: the notification only appears if there is an installed `.desktop` file whose basename matches the application id (`com.example.Notification.desktop` for `com.example.Notification`). Without it the call succeeds silently and nothing is shown. Desktop files are in Desktop Integration.

The full example is `examples/gtk4/notification.py`.

3.10 Summary

- List widgets are four parts: a model of GObjects, a factory, a selection model, and a view. `setup` builds a row, `bind` fills it, `unbind` undoes what `bind` did.
- Sorting and filtering are models wrapped around the store, driven by `Gtk.Expression`, so they run in C rather than in Python.
- File dialogs are asynchronous and hand you a `Gio.File`. Use it rather than `get_path()`, and the portal works for free.
- Drag and drop is a `Gtk.DragSource` and a `Gtk.DropTarget` added as controllers, moving `GValues`.

- The clipboard uses the same content types as drag and drop. Writing is immediate; reading is asynchronous and raises when there is nothing to read.
- `Gtk.Image` is for icons, `Gtk.Picture` is for content.
- Layouts can live in `.ui` files; `Gtk.Template` binds one to a class. Glade is not part of this any more.
- Status icons are gone; `Gio.Notification` covers what they were used for.

Threads and Asynchronous Work is next: how to do something slow without the window freezing.

Chapter 4

Threads and Asynchronous Work

> Every listing in this chapter is a file under `examples/gtk4/threads/`. They are run on each build, so if one of them stops working the build says so.

4.1 Introduction

Everything your program does happens on one thread, in one loop. A callback that takes 200 ms makes the window stutter; one that takes two seconds makes it stop responding, stop redrawing, and — if the user clicks anything — get reported to the desktop as hung.

So the question this chapter answers is: how do you do something slow without that happening?

There are four answers, and picking the right one matters more than the details of any of them:

Use an asynchronous API. For files, network and D-Bus, Gio already has one. No thread, no locking, no marshalling. This should be your first choice and it usually is not, because starting a thread looks more familiar.

Use `async/await`. PyGObject can run asyncio *on* the GLib main loop, which turns those Gio calls into ordinary `await` expressions and lets you use any `async def` library. On PyGObject 3.50 or later this is the pleasantest option in the chapter, and it has its own section.

Use a thread. For work that is genuinely CPU-bound, or a blocking library with no async version. Compute on the thread, touch no widgets, hand the result back to the main loop.

Break the work up. For something that can be done in pieces, do a piece per idle callback and let the loop breathe between them.

4.2 The one rule

GTK is not thread-safe. Widgets belong to the main thread.

Not "prefer the main thread" — only the main thread. Reading a label's text from a worker is undefined behaviour just as much as setting it. It will appear to work for months and then corrupt something.

GTK 2 had `gdk_threads_enter()` and `gdk_threads_leave()`, and the previous edition of this book used them. They are **gone**, with no replacement. There is no lock you can take to make widget access safe from another thread; the answer is not to do it.

What you get instead is one function:

```
GLib.idle_add(callback, *args)
```

`GLib.idle_add` and `GLib.timeout_add` are **safe to call from any thread**, and the callback they schedule runs on the main thread. That is the entire interface between a worker and your interface, and it is enough.

A callback returning `GLib.SOURCE_REMOVE` (`False`) runs once; returning `GLib.SOURCE_CONTINUE` (`True`) runs again. Forgetting to return anything means `None`, which is falsy, which happens to mean "run once" — so the bug only shows up when you meant to repeat.

4.3 Asynchronous I/O, without threads

For anything that waits on the world rather than on the CPU, Gio has it covered already:

```
file = Gio.File.new_for_path(path)
file.load_contents_async(cancellable, self.on_read_done)


def on_read_done(self, file, result, _data=None):
    try:
        ok, contents, _etag = file.load_contents_finish(result)
    except GLib.Error as error:
        return
    text = contents.decode("utf-8", "replace")
```

This is the same shape as the dialogs in Getting Started and the D-Bus calls in D-Bus: an `*_async` that returns immediately, a callback, and a `*_finish` that either gives you the value or raises `GLib.Error`.

Nearly everything has one — reading, writing, copying, deleting, enumerating a directory, resolving a hostname, opening a socket, making an HTTP request through libsoup. Enumeration is asynchronous twice over, once to open the directory and again per batch of entries:

```
folder.enumerate_children_async(
    "standard::name,standard::size,standard::type",
    Gio.FileQueryInfoFlags.NONE, GLib.PRIORITY_DEFAULT,
    cancellable, self.on_enumerate_done,
)
...
enumerator.next_files_async(50, GLib.PRIORITY_DEFAULT,
                            cancellable, self.on_files_done)
```

That batching is deliberate: a directory with a hundred thousand files does not arrive as one hundred-thousand-element list that blocks the loop while it is built.

Learn this callback shape first, because it is the one the GTK documentation is written in and the one you will read in other people's code. Then know that on PyGObject 3.50 and later you can drop the callback argument and `await` the same call instead, which is the section on asyncio below.

4.3.1 Cancellation

Every async Gio call takes a `Gio.Cancellable`, and this is where it earns its place over a thread:

```
self.cancellable = Gio.Cancellable()
file.load_contents_async(self.cancellable, self.on_read_done)
...
self.cancellable.cancel()
```

Cancelling makes the pending `*_finish()` raise, and you can tell that case apart from a real failure:

```
except GLib.Error as error:
    if error.matches(Gio.io_error_quark(), Gio.IOErrorEnum.CANCELLED):
        self.status.set_text("cancelled")
    else:
        self.status.set_text(f"failed: {error.message}")
```

This is **real cancellation** — the operation actually stops. A thread you have asked to stop keeps running until it next checks the flag, and if it is blocked in a syscall it may never check at all.

Use one cancellable per operation, not one for the program. Reusing a cancelled one makes every future operation fail immediately, which is a confusing bug to find. If you must, `cancellable.reset()`.

The full example is `examples/gtk4/threads/gio-async.py`.

4.4 Threads

When the work is genuinely CPU-bound, or the library you have to call has no asynchronous version, use a thread — and keep it strictly on the far side of the line:

```
def on_start(self, _button):
    self.cancel = threading.Event()
    self.thread = threading.Thread(target=self.run, daemon=True)
    self.thread.start()


def run(self):
    """Off the main thread. No widget access."""
    result = slow_work(self.cancel, self.report_progress)
    GLib.idle_add(self.on_finished, result)


def report_progress(self, done, total):
    GLib.idle_add(self.on_progress, done, total)


def on_progress(self, done, total):
    """Back on the main thread."""
    self.progress.set_fraction(done / total)
    return GLib.SOURCE_REMOVE
```

The worker's only contact with the interface is `GLib.idle_add`. Everything else is ordinary Python.

Four things worth doing every time:

`daemon=True`, so a half-finished worker cannot keep the process alive after the last window closes.

Cancel with a `threading.Event` the worker checks between units of work. This is cooperative — the worker stops when it next looks — which is the best a thread can do.

Disable the button that starts it. Two clicks should not start two workers unless you meant it.

Put a spinner somewhere. If it stops turning, something is blocking the main thread, and it tells you at a glance.

4.4.1 About the GIL

Python threads do not run Python bytecode in parallel, so a thread will not make pure-Python computation faster — it only keeps it off the main thread, which is what you wanted anyway.

Where they genuinely run in parallel is in C code that releases the GIL, which includes most of what you would actually thread: file and socket I/O, `zlib`, `hashlib`, image decoding, NumPy. For pure-Python CPU work that must go faster rather than merely elsewhere, that is `multiprocessing` — and then the results come back through a queue you poll from a `GLib.timeout_add`, or through `Gio.Subprocess`, which is asynchronous and needs no thread at all.

The full example is `examples/gtk4/threads/worker-thread.py`.

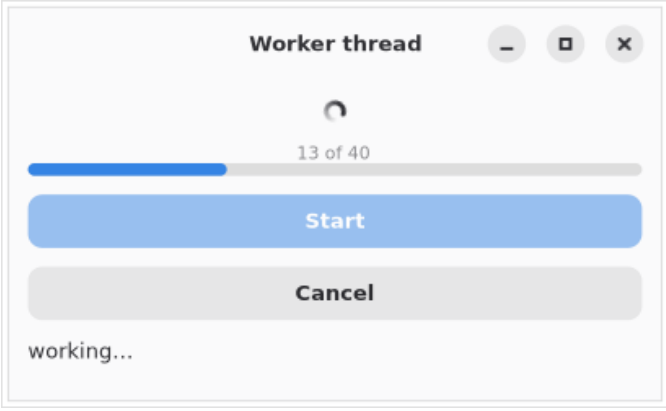

Figure 4.1: Progress reported from a worker thread through GLib.idle_add

4.5 Breaking work up

For work that divides neatly, there is a third option that needs neither a thread nor an async API — do a piece at a time and return to the loop between pieces:

```
def step(self):
    chunk, self.remaining = self.remaining[:100], self.remaining[100:]
    self.process(chunk)
    self.progress.set_fraction(1 - len(self.remaining) / self.total)
    return GLib.SOURCE_CONTINUE if self.remaining else GLib.SOURCE_REMOVE

GLib.idle_add(self.step)
```

An idle callback runs when the loop has nothing better to do, so the interface stays responsive and the work still finishes promptly. No thread, no locking, and you can touch widgets directly because you never left the main thread.

Size the chunk so each pass is a few milliseconds. Too small and the overhead dominates; too large and you are back to stuttering.

4.6 asyncio

Sooner or later you will want a library that is `async def` all the way down — `httpx`, `aiohttp`, `asyncpg`. GTK runs a GLib main loop and asyncio runs its own, and the old advice was that only one of them can own the main thread.

That is no longer true. **PyGObject 3.50 ships an asyncio event loop that is the GLib main loop.** Install it and there is one loop, GTK turns it, and `async def` code runs on it. Wrap the call to `run()`:

```
import asyncio
from gi.events import GLibEventLoop

app = Adw.Application(application_id="com.example.AsyncioAwait")
app.connect("activate", on_activate)

with GLibEventLoop(None):
    sys.exit(app.run(sys.argv))
```

Two things follow from that, and both are large.

4.6.1 Gio's async methods become awaitable

Omit the callback argument from any Gio `*_async` method and you get an awaitable back instead. The `*_async`/`*_finish` pair — two functions, with the interesting part split across them — collapses into one line:

```
async def read_file(self):
    file = Gio.File.new_for_path("example.txt")
    ok, contents, _etag = await file.load_contents_async()
    self.view.get_buffer().set_text(contents.decode("utf-8", "replace"))
```

Compare that with the callback version in the section above, which needs `on_read`, `on_read_done`, and a `load_contents_finish()` whose failure mode you have to remember to catch. Here failure is an ordinary exception at the point of the `await`:

```
try:
    ok, contents, _etag = await file.load_contents_async()
except GLib.Error as error:
    self.status.set_text(f"read failed: {error.message}")
```

4.6.2 A coroutine may touch widgets

This is the part that matters most, and it is the reverse of everything in the threads section. A coroutine on this loop runs **on the main thread**, between frames, like any other main loop source. The one rule is not being broken, because there is no other thread: setting a label inside an `async def` is exactly as safe as setting it in a `clicked` handler. No `GLib.idle_add`, no marshalling, no result-and-error tuple threaded back through a callback.

`asyncio.gather` also just works, because there is a running loop:

```
async def gather(self):
    self.status.set_text("three at once…")
    results = await asyncio.gather(slow(0.3), slow(0.2), slow(0.1))
    self.status.set_text(" / ".join(results))
```

4.6.3 Two traps

Keep a reference to your tasks. asyncio holds only a *weak* reference to a running task. Drop yours and it can be garbage collected in the middle of an `await`, which presents as work that silently never finishes — no error, no result. Hold them in a set and discard each one as it completes:

```
def spawn(self, coro):
    task = asyncio.ensure_future(coro)
    self.tasks.add(task)
    task.add_done_callback(self.tasks.discard)
    return task
```

Create tasks from inside the loop, not before it. With the context-manager form there is no current event loop until `run()` starts turning it, so this fails:

```
with GLibEventLoop(None):
    asyncio.ensure_future(startup())   # RuntimeError: There is no current event loop
    app.run(sys.argv)
```

From `activate` onwards — and from every signal handler after it — it works. Start your first coroutine in `activate`.

Cancellation is asyncio's: `task.cancel()` raises `asyncio.CancelledError` inside the coroutine at its next `await`. Catch it if you have something to clean up, and re-raise.

The full example is `examples/gtk4/threads/asyncio-await.py`.

4.6.4 The older forms

You will meet two other spellings of this.

`asyncio.set_event_loop_policy(GLibEventLoopPolicy())` is what the PyGObject documentation shows and what most existing code does. It works, and unlike the context manager it gives you a loop before `run()` starts. But event loop policies are deprecated in Python 3.14 and **removed in 3.16**, so new code should prefer `GLibEventLoop`.

Before PyGObject 3.50 there was no integration at all, and the answer was to give asyncio a thread of its own for the life of the program, submitting coroutines to it with `run_coroutine_threadsafe` — the only asyncio function that is safe to call from another thread — and marshalling results back with `GLib.idle_add`. `examples/gtk4/threads/asyncio-bridge.py` is that program, kept because you will still meet it, and because it is what you need if you are stuck on an older PyGObject. It carries a trap of its own: `run_coroutine_threadsafe` takes a coroutine, and `asyncio.gather()` off the loop thread does not return one, so `gather` has to be wrapped in an `async def` first.

The third-party bridges, **gbulb** and **asyncio-glib**, solved this before PyGObject did. They are no longer the answer.

4.7 Choosing

Waiting on a file, a socket, a subprocess or D-Bus A Gio `*_async` call. No thread. Real cancellation.

A long computation, or a blocking library with no async version A thread, with `GLib.idle_add` for every interface update.

Work that divides into pieces `GLib.idle_add` returning `SOURCE_CONTINUE`. No thread at all.

An `async def` library `GLibEventLoop`, on PyGObject 3.50 or later. An asyncio loop on its own thread if you are stuck below it.

Several I/O operations at once, whose results update the interface `GLibEventLoop` and `asyncio.gather`. The coroutine can set the widgets itself.

Making pure-Python computation actually faster `multiprocessing`, or `Gio.Subprocess`.

4.8 Summary

- Widgets are main-thread only. `gdk_threads_enter()` is gone and has no replacement.
- `GLib.idle_add` and `GLib.timeout_add` are safe from any thread and are the whole interface between a worker and the interface.
- Return `SOURCE_REMOVE` or `SOURCE_CONTINUE` deliberately; falling off the end returns `None`, which means remove.
- Prefer a Gio async call to a thread. `Gio.Cancellable` really stops the work; a `threading.Event` only asks.
- One cancellable per operation.
- Threads want `daemon=True`, a cancel flag, a disabled start button and a spinner.
- The GIL means a thread moves Python work off the main thread rather than making it faster.
- `with GLibEventLoop(None):` around `run()` makes asyncio's loop the GLib main loop. Gio's `*_async` methods then return awaitables when you omit the callback.
- A coroutine on that loop runs on the main thread, so it may touch widgets. This is the reason to prefer it to a thread.
- Hold a reference to every task; asyncio only holds a weak one.
- Create your first task from `activate`, not before `run()`.
- Event loop policies still work but are removed in Python 3.16.

Drawing with Cairo is next.

Chapter 5

Drawing with Cairo

> Every listing in this chapter is a file under `examples/gtk4/cairo/`. They are run on each build, and the figures below are their output rather than screenshots, so the pictures and the code cannot drift apart.

5.1 Introduction

Sooner or later no existing widget draws what you need — a chart, a waveform, a seating plan, a game board — and you draw it yourself. Cairo is the library for that: a 2D vector graphics API with paths, fills, strokes, gradients, clipping and transformations, rendering the same drawing to a window, a PNG, a PDF or a printer.

Cairo's place in GTK has changed, and it is worth being clear about it. GTK 4 does **not** paint with Cairo. Widgets build a tree of render nodes and GSK hands that tree to the GPU. Cairo survives inside `Gtk.DrawingArea`, which is a widget that wraps a single Cairo render node — GTK renders your Cairo drawing to a texture and composites it like anything else.

So there are two ways to draw in GTK 4, and both belong in this chapter:

Cairo, through `Gtk.DrawingArea`, when the drawing is arbitrary — curves, paths, text, anything you would describe as *illustration*. It is also the only option when the same drawing has to go to a PDF or a printer.

GtkSnapshot, in a custom widget, when the drawing is made of the primitives GSK already knows — rectangles, rounded borders, gradients, shadows, existing textures. It stays on the GPU and stays sharp at any scale.

Most application drawing is Cairo. Most *widget* drawing is snapshot.

5.2 Cairo basics

Two objects do all the work.

A **surface** is what you draw on. `cairo.ImageSurface` is a block of pixels in memory. `cairo.PDFSurface`, `cairo.SVGSurface` and `cairo.PSSurface` write vector files. GTK gives you a surface for a widget without your asking.

A **context** is what you draw with. It holds the current colour, line width, font, transformation and clip, and it holds the path you are building.

```
import cairo

surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, 400, 300)
context = cairo.Context(surface)

context.set_source_rgb(1, 1, 1)
context.paint()

context.set_line_width(15)
context.set_line_cap(cairo.LINE_CAP_ROUND)
context.set_source_rgb(0.2, 0.3, 0.7)
context.move_to(50, 50)
context.line_to(350, 100)
context.stroke()

surface.write_to_png("draw-to-png.png")
```

Colour components run from 0 to 1, not 0 to 255. `set_source_rgba()` takes an alpha as well. `paint()` floods the whole clip region with the current source, which is the usual way to set a background.

The origin is the **top left**, y increases **downwards**, and angles are in **radians** with zero pointing right. `math.radians()` converts if you would rather think in degrees.

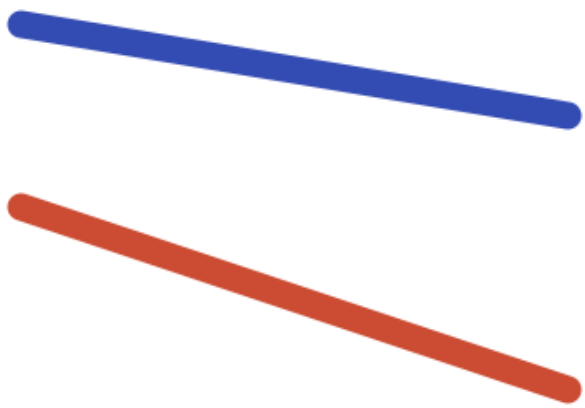

Figure 5.1: Two lines drawn to an image surface

5.2.1 Surface formats

`cairo.ImageSurface` needs a pixel format:

`cairo.FORMAT_ARGB32` 32 bits per pixel: alpha, then red, green, blue, stored native-endian, with **pre-multiplied alpha**. 50% transparent red is `0x80800000`, not `0x80ff0000`. This is the default choice.

`cairo.FORMAT_RGB24` 32 bits per pixel with the top 8 unused. No transparency, slightly faster.

`cairo.FORMAT_A8` 8 bits of alpha and nothing else — a mask.

`cairo.FORMAT_A1` 1 bit of alpha, packed 32 to a word.

5.2.2 The same drawing, different surfaces

Because the drawing code only touches the context, retargeting it costs nothing:

```
def draw(context):
    context.set_source_rgb(0.1, 0.5, 0.3)
    context.rectangle(40, 40, 220, 120)
    context.fill_preserve()      # fill, but keep the path
    context.set_source_rgb(0, 0, 0)
    context.set_line_width(4)
    context.stroke()             # ...so it can be stroked too


png = cairo.ImageSurface(cairo.FORMAT_ARGB32, 300, 200)
draw(cairo.Context(png))
png.write_to_png("surfaces.png")

pdf = cairo.PDFSurface("surfaces.pdf", 300, 200)
draw(cairo.Context(pdf))
pdf.finish()                     # flush and close the file
```

`finish()` matters for the file-backed surfaces: without it the file may be empty or truncated when the program exits.

One thing to watch: the numbers mean different things. On an image surface they are device pixels. On a PDF, SVG or PostScript surface they are **points**, at 72 to the inch — so a 300×200 PDF is about 106×70 millimetres, not a small picture. This is the same unit the printing chapter uses.

`fill_preserve()` and `stroke_preserve()` keep the path so the next operation can use it. Plain `fill()` and `stroke()` clear it, which is what you want most of the time and a source of confusion the one time you did not.

The full example is `examples/gtk4/cairo/surfaces.py`.

5.3 Drawing in a widget

GTK 2's `expose-event` and GTK 3's `draw` signal are gone. A `Gtk.DrawingArea` takes a draw function:

```
def draw(_area, context, width, height):
    context.set_source_rgb(0.98, 0.97, 0.94)
    context.paint()

    centre_x, centre_y = width / 2, height / 2
    radius = min(width, height) / 2 - 20

    context.set_source_rgb(0.2, 0.4, 0.8)
    context.arc(centre_x, centre_y, radius, 0, 2 * math.pi)
    context.fill()


area = Gtk.DrawingArea()
area.set_draw_func(draw)
area.set_hexpand(True)
area.set_vexpand(True)
```

The function is handed a context that is already clipped to the widget and has its origin at the widget's top left, plus the current width and height. Use them. Drawing to sizes you assumed rather than sizes

you were given is the single most common bug in custom drawing, and it only shows up when someone resizes the window.

Three rules:

Never call the draw function yourself. Call `widget.queue_draw()` and let GTK decide when — it will coalesce several requests into one frame.

Do no work in it. It runs on the frame clock. Loading a file, querying a database or computing a layout inside a draw function makes the whole interface stutter. Compute in advance, draw from the result.

Do not keep the context. It is valid for that one call.

The full example is `examples/gtk4/cairo/drawing-area.py`.

Figure 5.2: Cairo drawing inside a Gtk.DrawingArea

5.4 Paths

Everything Cairo draws is a path: a sequence of lines and curves, which you then stroke or fill.

```
context.move_to(x, y)                        # start a new sub-path
context.line_to(x, y)                        # straight segment
context.curve_to(x1, y1, x2, y2, x3, y3)     # cubic Bézier: 2 controls, 1 end
context.rel_line_to(dx, dy)                  # ...relative to where you are
context.arc(cx, cy, radius, start, end)      # clockwise
context.arc_negative(cx, cy, radius, start, end)   # anticlockwise
context.rectangle(x, y, width, height)
context.close_path()                         # straight back to the sub-path start
```

Three things about `arc()` catch people out:

It **continues the current path**. If there is already a current point, Cairo draws a straight line from it to the start of the arc. To start a fresh circle, call `new_sub_path()` first — this is why two `arc()` calls in a row produce two circles joined by a diagonal.

Angles are radians and **zero points right**, increasing clockwise (because y points down). A quarter turn "up" from zero is `-math.pi / 2`.

An arc under a non-uniform scale becomes an ellipse — which is how you draw one: `scale(1, 0.5)` then `arc()`.

5.4.1 Stroking

```
context.set_line_width(12)
context.set_line_cap(cairo.LINE_CAP_ROUND)    # BUTT, ROUND, SQUARE
context.set_line_join(cairo.LINE_JOIN_ROUND)  # MITER, ROUND, BEVEL
context.set_dash([8, 4], 0)                   # on 8, off 4
context.stroke()
```

The line width is in user-space units, and it is **centred on the path** — half either side. A 1-pixel line drawn on an integer coordinate therefore straddles two pixel columns and comes out 2 pixels wide and grey. Offset by a half:

```
context.move_to(x + 0.5, 0)
context.line_to(x + 0.5, height)
```

That half pixel is the difference between a crisp grid and a blurry one.

5.4.2 Filling

`fill()` colours the inside of the path, and *inside* depends on the fill rule:

```
context.set_fill_rule(cairo.FILL_RULE_EVEN_ODD)   # or FILL_RULE_WINDING
```

`WINDING` (the default) counts direction: a hole only appears if the inner sub-path runs the opposite way round. `EVEN_ODD` counts crossings, so any nested sub-path punches a hole regardless of direction. The left shape below is even-odd, the right one winding, and the two circles are drawn identically in both.

5.4.3 Colour and patterns

The *source* is what fills or strokes put down. It can be a colour or a pattern:

```
gradient = cairo.LinearGradient(0, 0, 180, 0)
gradient.add_color_stop_rgb(0.0, 0.9, 0.3, 0.2)
gradient.add_color_stop_rgb(1.0, 0.2, 0.3, 0.9)
context.set_source(gradient)
context.rectangle(0, 0, 180, 50)
context.fill()

radial = cairo.RadialGradient(60, 130, 5, 60, 130, 45)
radial.add_color_stop_rgba(0, 1, 1, 1, 1)
radial.add_color_stop_rgba(1, 0.2, 0.4, 0.8, 0)   # fade out to transparent
```

Gradient coordinates are in **user space, not in the shape** — the gradient is positioned on the canvas and the fill reveals part of it. Move the shape without moving the gradient and the colours change. Wrapping both in `translate()` is usually what you meant.

`cairo.SurfacePattern` uses another surface as the source, which is how you tile an image or draw one drawing into another.

5.4.4 Transformations and state

```
context.translate(x, y)
context.rotate(math.radians(20))
context.scale(sx, sy)
```

These change user space for everything drawn afterwards, and they compose — rotate after translate rotates about the new origin, which is nearly always what you want and the reverse of what you get if

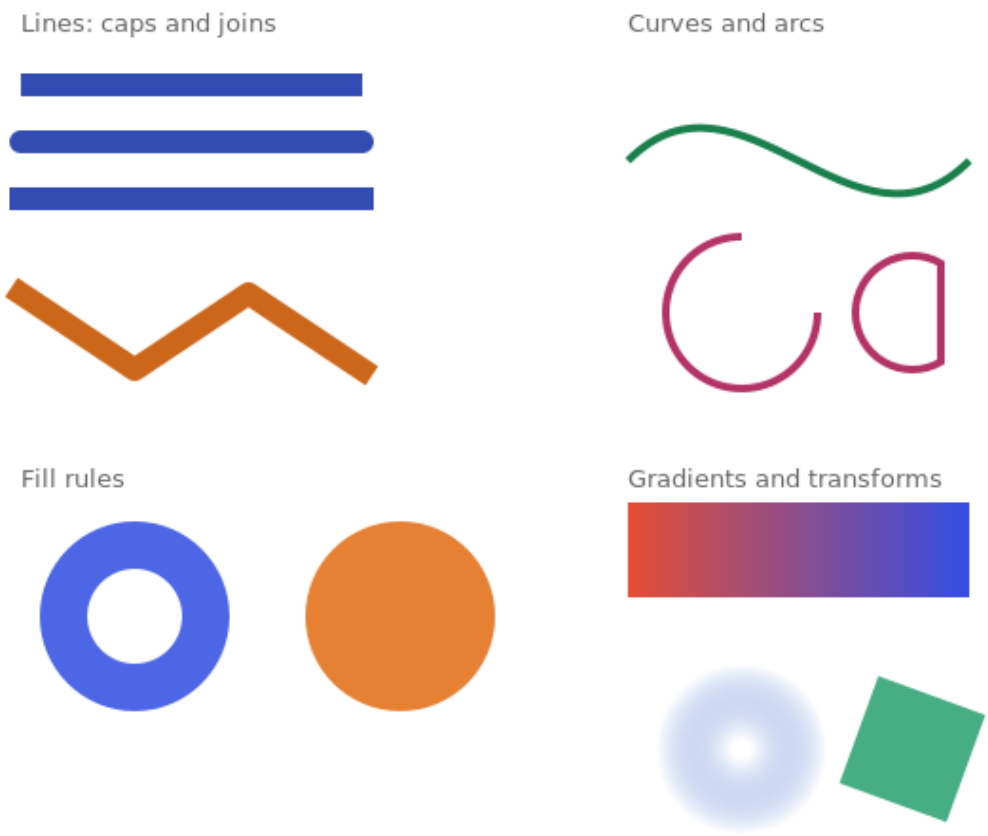

Figure 5.3: Paths, fills, gradients and transforms

you write them the other way round.

Because the transform is part of the context state, and so are the colour, line width, font and clip, `save()` and `restore()` are how you avoid leaking one part of a drawing into the next:

```
context.save()
context.translate(150, 130)
context.rotate(math.radians(20))
context.rectangle(-30, -30, 60, 60)
context.fill()
context.restore()
```

They nest. Any drawing routine that changes state should bracket itself this way, or callers end up debugging a rotation they did not ask for.

The full example is `examples/gtk4/cairo/paths-and-fills.py`.

5.5 Text

Cairo has a text API, and its own documentation calls it a *toy*:

```
context.select_font_face("sans-serif", cairo.FONT_SLANT_NORMAL,
                         cairo.FONT_WEIGHT_NORMAL)
context.set_font_size(13)
context.move_to(20, 30)
context.show_text("one line, one font, no wrapping")

extents = context.text_extents("centred")
context.move_to(width / 2 - extents.width / 2, 55)
context.show_text("centred")
```

It is genuinely fine for an axis label on a chart. It cannot wrap text, cannot mix fonts, cannot shape Arabic or Devanagari, and cannot lay out a right-to-left run. For anything a user will read, use **Pango**:

```
import gi
gi.require_version("Pango", "1.0")
gi.require_version("PangoCairo", "1.0")
from gi.repository import Pango, PangoCairo

layout = PangoCairo.create_layout(context)
layout.set_width(480 * Pango.SCALE)
layout.set_wrap(Pango.WrapMode.WORD)
layout.set_justify(True)
layout.set_font_description(Pango.FontDescription("Serif 11"))
layout.set_markup("<b>Pango</b> wraps text to a width and understands <i>markup</i>.")

context.move_to(20, 80)
PangoCairo.show_layout(context, layout)

_, height = layout.get_pixel_size()    # what it actually took
```

Two things to remember. Pango measures in **Pango units**, 1024 to the point, hence the `* Pango.SCALE`; `get_pixel_size()` gives you pixels back. And `layout.set_markup()` takes the same Pango markup as a label, so the same warning applies — escape anything you did not write with `GLib.markup_escape_text()`.

The full example is `examples/gtk4/cairo/text-with-pango.py`.

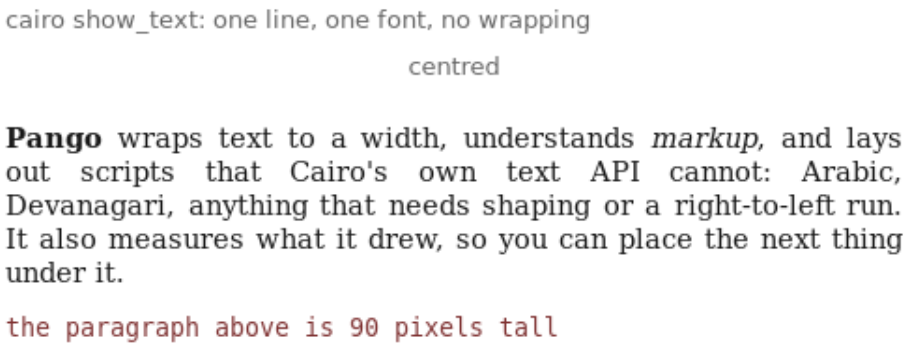

Figure 5.4: Cairo's text API next to a Pango layout

5.6 Antialiasing

Antialiasing softens the stair-stepping that comes from approximating a curve with square pixels. It is on by default, it costs almost nothing, and turning it off is nearly always a mistake:

```
context.set_antialias(cairo.ANTIALIAS_NONE)
```

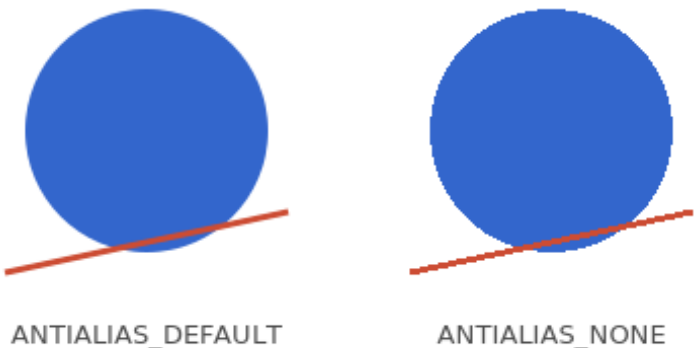

Figure 5.5: The same shapes with antialiasing on and off

Straight horizontal and vertical lines look identical either way — the difference is entirely in the curves and the diagonal. The one legitimate use is drawing something that must be pixel-exact, such as a colour picker where a blended edge would report the wrong value. If a line looks blurry, the fix is the half-pixel offset from the stroking section, not switching antialiasing off.

The full example is `examples/gtk4/cairo/antialias.py`.

5.7 Making it interactive

A drawing becomes a widget when it responds to input. That needs an event controller for the input and `queue_draw()` for the repaint:

```
class Sketch(Gtk.DrawingArea):
    def __init__(self):
        super().__init__()
        self.points = []
        self.set_draw_func(self.on_draw)

        click = Gtk.GestureClick()
        click.connect("pressed", self.on_pressed)
        self.add_controller(click)

        motion = Gtk.EventControllerMotion()
        motion.connect("motion", self.on_motion)
        self.add_controller(motion)

    def on_pressed(self, gesture, n_press, x, y):
        if n_press == 2:
            self.points.clear()
        else:
            self.points.append((x, y))
        self.queue_draw()
```

The coordinates handed to a controller are already **relative to the widget**, in the same space the draw function uses, so a click at `(x, y)` lands where you draw at `(x, y)`. There is no `gtk.EventBox` to wrap and no `add_events()` to call — controllers attach to any widget.

`n_press` counts clicks, so double-click handling needs no timer.

The full example is `examples/gtk4/cairo/interactive.py`.

5.8 Beyond Cairo: GtkSnapshot

For a widget whose appearance is made of shapes GSK already knows, skip Cairo and build render nodes directly. Override `do_snapshot()`:

```
class Meter(Gtk.Widget):
    __gtype_name__ = "Meter"

    def do_snapshot(self, snapshot):
        width, height = self.get_width(), self.get_height()

        track = Graphene.Rect().init(0, 0, width, height)
        filled = Graphene.Rect().init(0, 0, width * self.fraction, height)

        radius = Graphene.Size().init(height / 2, height / 2)
        rounded = Gsk.RoundedRect()
        rounded.init(track, radius, radius, radius, radius)

        snapshot.push_rounded_clip(rounded)
        snapshot.append_color(rgba(0.9, 0.9, 0.89), track)
        snapshot.append_linear_gradient(
            filled,
            Graphene.Point().init(0, 0),
            Graphene.Point().init(width, 0),
            [color_stop(0.0, rgba(0.25, 0.5, 0.9)),
             color_stop(1.0, rgba(0.55, 0.3, 0.85))],
```

```
        )
        snapshot.pop()
```

Every `push_*` needs a matching `pop()`, and GTK warns loudly at runtime if they do not balance.

Boxed types are a small trap here. `Gdk.RGBA` and `Gsk.ColorStop` are constructed empty and filled in, not built from keyword arguments:

```
def rgba(red, green, blue, alpha=1.0):
    colour = Gdk.RGBA()
    colour.red, colour.green, colour.blue, colour.alpha = red, green, blue, alpha
    return colour
```

`Gdk.RGBA(red=..., ...)` is accepted and silently ignores every argument, which produces a transparent black widget and no error at all.

Snapshot drawing is worth reaching for when the widget is many small pieces, when it redraws every frame, or when it must stay sharp on a scaled display. For a chart, Cairo is less code and no slower in practice.

The full example is `examples/gtk4/cairo/snapshot-widget.py`.

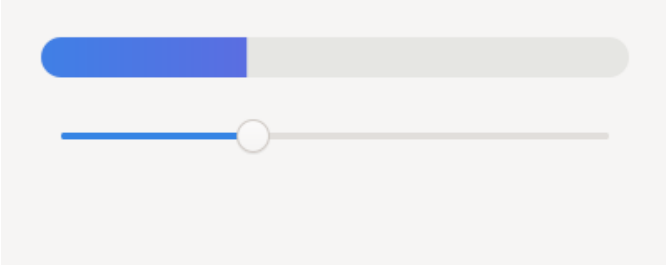

Figure 5.6: A widget drawn from render nodes rather than with Cairo

5.9 Summary

- Cairo draws onto a surface with a context; the same drawing code retargets to a PNG, a PDF, an SVG or a printer by changing only the surface.
- Colours are 0 to 1, the origin is top left, angles are radians, and vector surfaces measure in points.
- `Gtk.DrawingArea.set_draw_func()` is where `expose-event` went. Draw to the width and height you are given, do no work in it, and call `queue_draw()` to ask for a repaint.
- Build a path, then stroke or fill it. `arc()` continues the current path — use `new_sub_path()`. Offset thin lines by half a pixel.
- Cairo's text API is for labels; Pango is for text people read.
- For widget chrome made of rectangles and gradients, `do_snapshot()` keeps the drawing on the GPU.

Custom Widgets is next, and picks up where the snapshot section here leaves off.

Chapter 6

Custom Widgets

> Every listing in this chapter is a file under `examples/gtk4/custom-widgets/`. They are run on each build, so if one of them stops working the build says so.

6.1 Introduction

The previous edition of this book had a chapter called *Custom Widgets* containing one line: “This chapter is not yet written :)”. This is that chapter.

There are three levels of “custom widget”, and most of the time the answer is the first one:

Compose. Subclass an existing container, build some children in `__init__`, and give it properties and signals so the rest of the program sees one thing.

Draw. Subclass `Gtk.Widget` and override `do_snapshot()` — covered in Drawing with Cairo, where the meter widget does exactly this.

Lay out. Subclass `Gtk.Widget` and override `do_measure()` and `do_size_allocate()` as well, so it can arrange children GTK has no container for.

Only the third is genuinely hard, and only because the sizing contract has a rule that is easy to miss. That rule is the most useful thing in this chapter.

6.2 Compose first

Before overriding anything, ask whether you are making a new *kind* of widget or a reusable arrangement of existing ones. It is nearly always the second.

```
class SearchBar(Gtk.Box):
    __gtype_name__ = "SearchBar"

    query = GObject.Property(type=str, default="")
    scope = GObject.Property(type=int, default=0)

    @GObject.Signal(arg_types=(str, int))
    def search_requested(self, query, scope):
        """Default handler; there is nothing for it to do."""

    def __init__(self, **kwargs):
        super().__init__(orientation=Gtk.Orientation.HORIZONTAL, spacing=6, **kwargs)
```

```
        self._entry = Gtk.SearchEntry(hexpand=True, placeholder_text="Search…")
        self._scope = Gtk.DropDown.new_from_strings(SCOPES)
        self._button = Gtk.Button(label="Search")

        self.append(self._entry)
        self.append(self._scope)
        self.append(self._button)
```

What makes this a *widget* rather than a helper function is the public surface — the properties and the signal from GObject. From outside, nothing needs to know there is an entry and a drop down inside:

```
search.connect("search-requested", lambda _b, query, scope: ...)
search.bind_property("query", label, "label", GObject.BindingFlags.SYNC_CREATE)
```

Inside, connect the children to the properties with **bindings** rather than handlers, so the widget and its state cannot drift apart:

```
self._entry.bind_property("text", self, "query",
                          GObject.BindingFlags.BIDIRECTIONAL
                          | GObject.BindingFlags.SYNC_CREATE)
```

Two details make a composed widget feel like a real one. Prefix the children with an underscore — they are implementation, and something will eventually reach in and grab them if you do not. And override `grab_focus()` to forward to whichever child should actually get the focus:

```
def grab_focus(self):
    return self._entry.grab_focus()
```

For anything with more than a handful of children, put the layout in a `.ui` file and use `Gtk.Template`, as in More GTK 4. The class is the same shape; only where the children come from changes.

The full example is `examples/gtk4/custom-widgets/composed-widget.py`.

Figure 6.1: A composed widget: one entry, one drop down and one button, presented as a single thing

6.3 A widget that lays out children

When no existing container does what you need — and GTK has a lot of them, so check first — you subclass `Gtk.Widget` and implement three methods.

GTK 4 has **no `GtkContainer`**. There is no `add()` to override and no child list to maintain. A child is parented to you, and the widget itself keeps the list:

```
def append(self, child):
    child.set_parent(self)

def children(self):
    child = self.get_first_child()
    while child is not None:
        yield child
        child = child.get_next_sibling()
```

6.3.1 Unparent your children, or GTK complains

```
def do_dispose(self):
    child = self.get_first_child()
    while child is not None:
        next_child = child.get_next_sibling()
        child.unparent()
        child = next_child
    Gtk.Widget.do_dispose(self)
```

Leave this out and you get, at some later and unrelated moment:

```
Finalizing GtkWidget, but it still has children left
```

Take the next sibling **before** unparenting, because unparenting is what detaches it from the list you are walking.

6.3.2 Measuring

```
def do_measure(self, orientation, for_size):
    return minimum, natural, min_baseline, nat_baseline
```

GTK asks how big you would like to be, then gives you a size that may be different, then asks you to draw. The contract is that `do_measure` must not lie: a widget asking for less than it needs gets clipped, and one asking for more leaves holes.

Return `-1, -1` for the baselines unless you are aligning text across widgets.

6.3.3 The rule that catches everyone

If your height depends on your width — a wrapping layout, a text flow, anything that reflows — you must say so:

```
def do_get_request_mode(self):
    return Gtk.SizeRequestMode.HEIGHT_FOR_WIDTH
```

The default is `CONSTANT_SIZE`, which tells GTK that height does not depend on width. GTK then **never passes a real width** to `do_measure()`: `for_size` is always `-1`, your widget measures itself as one row tall however narrow it gets, and the rows below the first are silently clipped.

Nothing warns. The widget looks correct at its natural width and wrong at every other width, which is exactly the case you are least likely to test first.

It is worth seeing the difference. The wrapping example below, laid out at 300 pixels wide, genuinely occupies three rows:

```
row y=0:  ['alpha', 'bravo', 'charlie']
row y=42: ['delta', 'echo', 'foxtrot', 'golf']
row y=84: ['hotel']
```

With the default request mode, `measure(VERTICAL, 300)` reports **34** — one row. With `HEIGHT_FOR_WIDTH` it reports **118**, which is what the layout actually needs.

6.3.4 Allocating

```
def do_size_allocate(self, width, height, baseline):
    for child in self.children():
        transform = Gsk.Transform().translate(Graphene.Point().init(x, y))
        child.allocate(child_width, child_height, -1, transform)
```

A child is positioned by a **transform**, not by an x/y pair — which is what makes it possible to rotate or scale a child, and why the drawing chapter's snapshot transforms compose with layout.

Ask each child how big it wants to be rather than assuming:

```
child_width = child.measure(Gtk.Orientation.HORIZONTAL, -1)[1]
child_height = child.measure(Gtk.Orientation.VERTICAL, child_width)[1]
```

Note the second call passes the width — that is how you get a correct height from a child that is itself height-for-width.

Write the walk once and use it for both. The example's `_layout()` places children when asked to allocate and only totals up the height when asked to measure:

```
def _layout(self, visible, width, allocate=False):
    ...
    if allocate:
        child.allocate(child_width, child_height, -1, transform)
    ...
    return total
```

Measuring and allocating with two separate pieces of code is the classic way to get a container that is subtly the wrong size, because the two drift apart the first time one of them is edited.

6.3.5 Drawing

A container usually only has to draw its children, and the default implementation already does:

```
def do_snapshot(self, snapshot):
    Gtk.Widget.do_snapshot(self, snapshot)
```

Override it properly only when you want to draw *around* the children — a background, a focus ring, separators between rows.

The full example is `examples/gtk4/custom-widgets/flow-box-widget.py`.

6.4 Layout managers

There is a fourth option worth knowing. Instead of putting the layout logic in the widget, put it in a `Gtk.LayoutManager`:

```
class WrapLayout(Gtk.LayoutManager):
    __gtype_name__ = "WrapLayout"
```

Figure 6.2: Children wrapped onto three rows by a custom container

```
    def do_get_request_mode(self, widget):
        return Gtk.SizeRequestMode.HEIGHT_FOR_WIDTH

    def do_measure(self, widget, orientation, for_size): ...
    def do_allocate(self, widget, width, height, baseline): ...


widget.set_layout_manager(WrapLayout())
```

The methods are the same, with the widget passed in. The advantage is that the layout can be swapped at runtime and reused on any widget — which is how `Gtk.BoxLayout`, `Gtk.GridLayout` and `Gtk.BinLayout` are implemented, and why `Gtk.Box` is a thin wrapper around one.

Use a layout manager if the arrangement is reusable; put it in the widget if it is specific to that widget.

6.5 Styling and accessibility

Give the widget its own CSS name so it can be styled without depending on the class name:

```
class WrapBox(Gtk.Widget):
    __gtype_name__ = "WrapBox"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.set_css_name("wrapbox")
```

```
wrapbox { padding: 6px; }
```

And tell the accessibility layer what it is, because a `Gtk.Widget` subclass defaults to a role that means nothing to a screen reader:

```
class Meter(Gtk.Widget):
    __gtype_name__ = "Meter"
```

```
    def __init__(self, **kwargs):
        super().__init__(accessible_role=Gtk.AccessibleRole.PROGRESS_BAR, **kwargs)
        self.update_property([Gtk.AccessibleProperty.VALUE_NOW], [0.35])
```

A composed widget usually needs nothing here — its children already carry their own roles — which is one more argument for composing.

6.6 Summary

- Compose before you subclass `Gtk.Widget`. Properties and signals are what make a composed widget a widget; use bindings inside it and forward `grab_focus()`.
- There is no `GtkContainer`. Children are parented, and walked with `get_first_child()` / `get_next_sibling()`.
- Unparent every child in `do_dispose()`, taking the next sibling first.
- **If height depends on width, override `do_get_request_mode()`.** Without it `for_size` is always `-1`, the widget measures one row tall, and the rest is clipped with no warning.
- Children are placed with a `Gsk.Transform`, not an x/y pair.
- Measure children with `child.measure()`, passing the width when asking for a height.
- Use one code path for measuring and allocating.
- Put reusable arrangements in a `Gtk.LayoutManager`.
- Set a CSS name, and set an accessible role on anything that draws itself.

Printing is next.

Chapter 7

Printing

Every listing in this chapter is a file under `examples/gtk4/printing/`. They are run on each build, and the printed page below is the actual output of one of them.

7.1 Introduction

Printing in GTK is Drawing with Cairo with a different surface and a small amount of paperwork. `Gtk.PrintOperation` handles the paperwork — talking to CUPS, showing the print dialog, running the preview — and then hands you a Cairo context and asks you to draw a page.

That is the whole idea. If you can draw it on screen you can print it, and the drawing code often transplants unchanged.

There are two printing APIs in GTK 4, and they are for different jobs:

`Gtk.PrintOperation` The high-level one. It paginates, drives the dialog and the preview, and calls you back per page. Use it whenever your program is generating the pages.

`Gtk.PrintDialog` Added in GTK 4.14. Asynchronous, like the other new dialogs, and it prints a **file** or a stream you already have. Use it when you already hold a finished PDF and only need the user to pick a printer.

This chapter is about the first one.

7.2 The shape of a print job

A job runs in three acts, each a signal:

`begin-print` The printer and the page setup are settled, so you can finally ask how big a page is. Work out how many pages there are and call `set_n_pages()`.

`draw-page` Called once per page, in order, with a Cairo context and a page number.

`end-print` Let go of whatever `begin-print` set up.

```
operation = Gtk.PrintOperation()
operation.set_job_name("How printing works")
operation.set_unit(Gtk.Unit.POINTS)

operation.connect("begin-print", document.on_begin_print)
operation.connect("draw-page", document.on_draw_page)
operation.connect("end-print", document.on_end_print)

result = operation.run(Gtk.PrintOperationAction.PRINT_DIALOG, window)
```

`run()` takes what to do and a parent window:

- `PRINT_DIALOG` — ask the user, then print.
- `PRINT` — print with the current settings, no dialog.
- `PREVIEW` — open the preview.
- `EXPORT` — write a file, no dialog and no printer. Needs `set_export_filename()` first.

and returns `APPLY` if the job went ahead, `CANCEL` if the user backed out, `IN_PROGRESS` for an asynchronous job, or `ERROR`. Check for `ERROR` — a print job can fail for reasons that have nothing to do with your code.

7.2.1 One line you need before any of it works

```
gi.require_foreign("cairo")
```

`context.get_cairo_context()` converts a C `cairo_t` into a `cairo.Context`, and that conversion lives in a separate PyGObject module — `python3-gi-cairo` on Debian and Ubuntu. Without it the call fails with

```
TypeError: Couldn't find foreign struct converter for 'cairo.Context'
```

inside your draw handler, where GTK swallows the exception, prints it, and carries on producing blank pages. `gi.require_foreign("cairo")` moves the failure to the top of the file where it is obvious. It is worth adding to anything that draws.

7.3 Exporting a PDF

The quickest way to develop printing code is not to print. Export instead:

```
operation = Gtk.PrintOperation()
operation.set_n_pages(3)
operation.set_unit(Gtk.Unit.POINTS)
operation.set_export_filename("print-to-pdf.pdf")
operation.connect("draw-page", on_draw_page)

result = operation.run(Gtk.PrintOperationAction.EXPORT, None)
```

No dialog, no printer, no window — the parent can be `None`. Nothing about the drawing changes when a real printer is involved later, so this is also how the examples in this chapter are tested and how the figure below was made.

It doubles as a feature. “Export as PDF” is worth having in any program that can print, and it costs one enum.

7.4 Drawing a page

```
def on_draw_page(_operation, context, page_number):
    cr = context.get_cairo_context()
    width = context.get_width()
    height = context.get_height()
```

The context has already been set up for you, and the three details that follow from that are the ones people get wrong:

The origin is the printable area, not the paper. Margins are already subtracted, so `(0, 0)` is the top left of the area you may draw in, and `get_width()` and `get_height()` are that area’s size.

Drawing at negative coordinates to "reach the edge" is not how you get a full-bleed page — that is `set_use_full_page(True)`.

The units are what you asked for. `set_unit(Gtk.Unit.POINTS)` gives points, 72 to the inch, which is what Cairo and Pango both think in. `Gtk.Unit.MM` and `Gtk.Unit.INCH` are there if your layout is specified in physical units. The default, `Gtk.Unit.NONE`, gives device units — pixels at the printer's resolution — which will be 600 dpi on a laser printer and make a 12-point font microscopic.

The resolution is not the screen's. `context.get_dpi_x()` tells you the real figure. For text, do not do this arithmetic at all: use `context.create_pango_layout()`, which returns a layout already set up for the printer's resolution.

Text on a page is Pango, exactly as in the drawing chapter:

```
title = context.create_pango_layout()
title.set_font_description(Pango.FontDescription("Sans Bold 14"))
title.set_text(f"Quarterly report – page {page_number + 1}", -1)
cr.move_to(0, 0)
PangoCairo.show_layout(cr, title)

_, title_height = title.get_pixel_size()
cr.move_to(0, height - footer_height)      # position the footer from the bottom
```

Figure 7.1: A page produced by the export example

The full example is `examples/gtk4/printing/print-to-pdf.py`.

7.5 Pagination

You cannot count pages before you know the paper size, and you do not know the paper size until the user has chosen one. That is what `begin-print` is for:

```
def on_begin_print(self, operation, context):
    layout = context.create_pango_layout()
    layout.set_font_description(FONT)
    layout.set_text("Ag", -1)
    _, self.line_height = layout.get_pixel_size()

    usable = context.get_height() - self.line_height * 3    # header and footer
    self.lines_per_page = max(1, int(usable // self.line_height))

    pages = -(-len(TEXT) // self.lines_per_page)            # ceiling division
    operation.set_n_pages(pages)
```

Measure a representative string to get a line height rather than assuming one from the point size — the two are not the same, and the difference accumulates down a page.

Whatever `begin-print` measures has to be kept for `draw-page` to use, which is why the example puts both on a small `Document` object rather than in globals. Each job gets its own.

Fixed line heights are a simplification. Real pagination has to measure each paragraph, keep a heading with the text under it, and avoid leaving one line of a paragraph stranded at the bottom of a page. Pango can tell you all of that — `layout.get_iter()` walks a layout line by line — but the shape of the job does not change.

If pagination is genuinely expensive, `set_n_pages(-1)` in `begin-print` and use the `paginate` signal instead: GTK calls it repeatedly, you do a little work each time and return `False` until you are finished, and the interface stays responsive.

7.6 Page setup and settings

Two objects carry the user's choices, and they are different things:

`Gtk.PageSetup` The paper: size, orientation, margins. Changed with the page setup dialog.
`Gtk.PrintSettings` The job: which printer, how many copies, duplex, quality, page range.

```
self.page_setup = Gtk.print_run_page_setup_dialog(self, self.page_setup, self.settings)

operation.set_default_page_setup(self.page_setup)
operation.set_print_settings(self.settings)
operation.set_embed_page_setup(True)      # let the print dialog change it too
```

`Gtk.print_run_page_setup_dialog()` blocks and hands back a **new** page setup — it does not modify the one you passed in, so assign the result.

Keep both between jobs, or every print starts from scratch:

```
if result == Gtk.PrintOperationResult.APPLY:
    self.settings = operation.get_print_settings()
```

To remember them between *runs* of the program, both objects serialise to a key file:

```
settings.to_file(path)
settings = Gtk.PrintSettings.new_from_file(path)
```

`Gtk.PaperSize` knows the standard sizes by name — `Gtk.PAPER_NAME_A4`, `Gtk.PAPER_NAME_LETTER` — and `Gtk.PaperSize.get_default()` picks the right one for the user's locale. Hard-coding either A4 or Letter is a good way to annoy half your users.

The full example is `examples/gtk4/printing/print-a-document.py`.

7.7 Under a sandbox

Inside Flatpak your process cannot see CUPS. It does not have to: `Gtk.PrintOperation` is routed through the **print portal**, which shows the dialog outside the sandbox and takes the finished document back. The code is identical and there is nothing to add.

The one visible difference is that a job may complete asynchronously, so `run()` can return `IN_PROGRESS`. Connect to `done` if you need to know when it actually finished:

```
operation.connect("done", lambda op, result: print("finished:", result))
```

7.8 Summary

- `Gtk.PrintOperation` runs the job: `begin-print` counts the pages, `draw-page` draws each one, `end-print` cleans up.
- `gi.require_foreign("cairo")` at the top, or `get_cairo_context()` fails inside your handler and prints blank pages.
- Export to PDF while developing: it needs no printer, no dialog and no window, and the drawing code is identical.
- The context's origin is the printable area, and `set_unit(Gtk.Unit.POINTS)` is almost always the unit you want.
- Use `context.create_pango_layout()` for text — it is already at the printer's resolution.
- `Gtk.PageSetup` is the paper, `Gtk.PrintSettings` is the job. Keep both between jobs, and save them to a file to keep them between runs.

Desktop Integration is next: settings, desktop files, portals and the rest of making a program part of the desktop rather than a window floating on it.

Chapter 8

Desktop Integration

> Every listing in this chapter is a file under `examples/gtk4/desktop/`. They are run on each build, so if one of them stops working the build says so.

8.1 Introduction

A program that opens a window is not yet an application. It has no name in the launcher, no icon, nowhere to keep preferences, no way to be told to open a file, and no way to ask for a password without inventing its own storage. This chapter is about the rest of it.

Most of what the previous edition covered here has been replaced. GConf is gone and GSettings took over. gnome-keyring's API is gone and libsecret took over. And a newer idea sits underneath all of it: **portals**. On a modern desktop your program may be running in a sandbox with no access to the file system, the printer, the camera or the network beyond what it has been granted. Portals are how it asks. The useful part is that GTK's own dialogs already go through them, so code written the normal way works sandboxed and unsandboxed without changing.

8.2 The application id

Nearly everything in this chapter keys off one string:

```
app = Gtk.Application(application_id="com.example.Settings")
```

The id decides the name of your desktop file, the name of your icon, the D-Bus name you are activated on, the GSettings path convention, and which notifications are yours. Get it right early — changing it later means changing five other things.

The rules: reverse-DNS, using a domain you actually control, matching the file names around it. `com.example.Settings` wants `com.example.Settings.desktop`, `com.example.Settings.svg` and `com.example.Settings.gschema.xml`. If you have no domain, GitHub-based ids like `io.github.username.AppName` are conventional and accepted by Flathub.

8.3 Settings

GConf is gone. GSettings replaced it, and the difference that matters is that **GSettings keys are declared in advance**. A schema states every key, its type, its default and what it is for. There is no writing a key that does not exist, and no reading one and wondering what type comes back.

8.3.1 The schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schemalist>
  <schema id="com.example.Settings" path="/com/example/Settings/">

    <key name="greeting" type="s">
      <default>'Hello'</default>
      <summary>The word to greet with</summary>
      <description>Shown in the entry, and remembered between runs.</description>
    </key>

    <key name="enabled" type="b">
      <default>true</default>
      <summary>Whether the greeting is shown at all</summary>
    </key>

    <key name="repeat" type="i">
      <default>1</default>
      <range min="1" max="10"/>
      <summary>How many times to repeat the greeting</summary>
    </key>

    <key name="window-size" type="(ii)">
      <default>(420, 280)</default>
      <summary>The last window size</summary>
    </key>

  </schema>
</schemalist>
```

The `type` is a **GVariant type string**: `s` string, `b` boolean, `i` int32, `d` double, `as` array of strings, `(ii)` a pair of int32s. String defaults need their own quotes inside the XML, which is the mistake everyone makes once.

`<range>` is enforced — writing 20 to `repeat` fails rather than storing it. The summary and description are read by settings editors and by translators.

The schema is compiled, not parsed at runtime:

```
sudo install -m 644 com.example.Settings.gschema.xml /usr/share/glib-2.0/schemas/
sudo glib-compile-schemas /usr/share/glib-2.0/schemas/
```

Forgetting `glib-compile-schemas` gives you a hard abort — GLib treats a missing schema as a programming error and calls `abort()`, so the message is "Settings schema 'com.example.Settings' is not installed" followed by a crash rather than an exception you can catch.

While developing, you do not have to install anything. Compile into a temporary directory and load from there:

```
source = Gio.SettingsSchemaSource.new_from_directory(
    str(compiled), Gio.SettingsSchemaSource.get_default(), False
)
schema = source.lookup(SCHEMA_ID, False)
settings = Gio.Settings.new_full(schema, None, None)
```

The example does exactly that, falling back to it when the schema is not installed, which is why it runs

from a checkout. `GSETTINGS_SCHEMA_DIR` in the environment does the same thing for a program you do not want to modify.

8.3.2 Reading and writing

```
settings = Gio.Settings.new("com.example.Settings")

greeting = settings.get_string("greeting")
repeat = settings.get_int("repeat")
settings.set_string("greeting", "Good morning")
settings.reset("greeting")            # back to the schema default
```

For types without a typed accessor, go through GVariant:

```
width, height = settings.get_value("window-size").unpack()
settings.set_value("window-size", GLib.Variant("(ii)", (800, 600)))
```

Writes are queued and applied asynchronously. That is normally invisible, but a program that writes settings and then immediately exits should call `Gio.Settings.sync()` first.

8.3.3 Binding, which is the good part

Most settings drive a widget, and for that there is no need for callbacks at all:

```
settings.bind("greeting", entry, "text", Gio.SettingsBindFlags.DEFAULT)
settings.bind("enabled", switch, "active", Gio.SettingsBindFlags.DEFAULT)
settings.bind("repeat", spin, "value", Gio.SettingsBindFlags.DEFAULT)
```

That is the whole preferences dialog: typing in the entry stores the value, and changing the value elsewhere updates the entry. `DEFAULT` is two-way; `GET` is read-only, which is how one key can also control another widget's sensitivity:

```
settings.bind("enabled", entry, "sensitive", Gio.SettingsBindFlags.GET)
```

For anything that is not a widget property, watch the key:

```
settings.connect("changed::greeting", lambda s, key: refresh())
settings.connect("changed", lambda s, key: refresh())      # any key
```

Because the value changes when *anything* writes it, a program that reacts to `changed` rather than to its own widgets stays correct when two of its windows are open, when the user edits the key in dconf-editor, and when a system policy overrides it.

The full example is `examples/gtk4/desktop/settings.py`.

8.4 Where files go

Settings cover preferences. For everything else there are the XDG base directories, and GLib knows all of them:

```
GLib.get_user_config_dir()     # ~/.config       - configuration you write
GLib.get_user_data_dir()       # ~/.local/share  - data the user would miss
GLib.get_user_cache_dir()      # ~/.cache        - safe to delete at any time
GLib.get_user_state_dir()      # ~/.local/state  - logs, recent files, window state
GLib.get_user_runtime_dir()    # /run/user/1000  - sockets, lock files; gone at logout

GLib.get_user_special_dir(GLib.UserDirectory.DIRECTORY_DOWNLOAD)
```

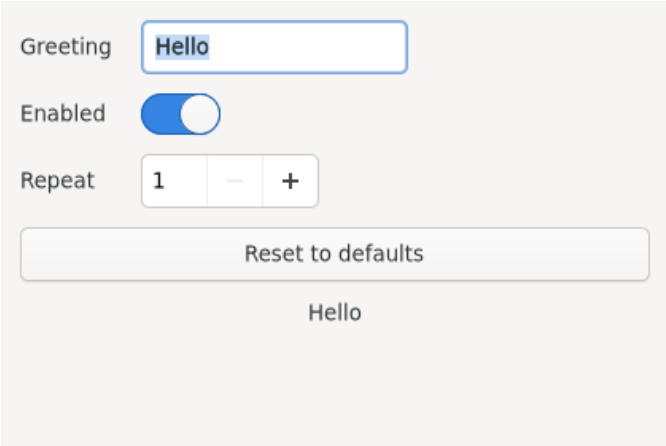

Figure 8.1: Widgets bound to GSettings keys, remembered between runs

Put your own directory *inside* one of those, named after your application id. Never build a path by joining `~` with `.myapp`: the XDG environment variables move these directories, some distributions do move them, and inside a Flatpak they are redirected into the sandbox. `GLib.get_user_special_dir()` also returns the *translated* name, so a French user's downloads are found in `~/Téléchargements` without your knowing about it.

The distinction between cache and data is worth honouring. Anything in the cache directory may be deleted while your program is not running, and anything in the data directory gets backed up. Putting a database in the cache loses it; putting thumbnails in data means backing them up forever.

8.5 Desktop files

A `.desktop` file is what puts your program in the launcher:

```
[Desktop Entry]
Type=Application
Version=1.5
Name=Settings Example
Comment=Shows how GSettings stores preferences
Exec=settings-example %U
Icon=com.example.Settings
Terminal=false
Categories=Utility;GTK;
Keywords=settings;preferences;example;
StartupNotify=true
DBusActivatable=true
```

`Name`, `Comment` and `Keywords` are shown to the user and should be translated — add `Name[de]=…` lines, or let your build system merge them from your `.po` files.

`Exec` must name something on `PATH` or give an absolute path, and it is **not** a shell command line: no pipes, no redirection, no quoting tricks. The `%U` is a field code that expands to the URIs the program was asked to open — `%F` for local file paths, `%u`/`%f` for a single one. A file manager passing you a document is `%U` at work, so leave it in even if you ignore it for now.

`Icon` should be an icon **name**, not a path — the base name of a file you installed into the icon theme:

```
~/.local/share/icons/hicolor/scalable/apps/com.example.Settings.svg
```

`Categories` decides where the entry appears in menus that still have categories, and the valid values are a fixed list in the freedesktop menu specification. Inventing one silently does nothing.

`DBusActivatable=true` says your application id is a D-Bus name and the desktop may start you by activating it rather than by running `Exec`. `Gtk.Application` already does everything needed for this; it is what makes a second launch raise your existing window instead of starting a second copy.

Install into `~/.local/share/applications` for one user or `/usr/share/applications` for everyone, then update the caches:

```
update-desktop-database ~/.local/share/applications
gtk-update-icon-cache -f -t ~/.local/share/icons/hicolor
```

Skipping those is why a new entry sometimes does not appear until the next login. And validate the file — the parser is stricter than it looks:

```
desktop-file-validate ~/.local/share/applications/com.example.Settings.desktop
```

The full example is `examples/gtk4/desktop/install-desktop-file.sh`.

8.5.1 Notifications need one

The notification API in More GTK 4 only works if an installed desktop file's basename matches the application id. Without it, `send_notification()` succeeds and nothing appears. This is the single most common "my notifications do not work" cause, and there is no error to go looking for.

8.6 Passwords

Passwords do not belong in GSettings. GSettings values are readable by anything in the user's session, are synchronised by some setups and end up in backups in the clear.

libsecret talks to the Secret Service — gnome-keyring on GNOME, KWallet on KDE:

```
gi.require_version("Secret", "1")
from gi.repository import Secret

SCHEMA = Secret.Schema.new(
    "com.example.Passwords",
    Secret.SchemaFlags.NONE,
    {"service": Secret.SchemaAttributeType.STRING,
     "username": Secret.SchemaAttributeType.STRING},
)

Secret.password_store(
    SCHEMA, {"service": "com.example.Passwords", "username": "alice"},
    Secret.COLLECTION_DEFAULT,
    "Example password for alice",      # shown in the keyring UI
    password, None, on_stored,
)
```

The schema here is a lookup key, not a security boundary: the attributes are what you search by when you want the password back. They are stored **unencrypted** — only the secret itself is protected — so do not put anything sensitive in an attribute.

Every call has a synchronous and an asynchronous form, and you want the asynchronous one. The keyring may be locked, in which case the desktop prompts the user for their password, and the synchronous call blocks your interface until they answer.

Handle "not found" separately from "failed". `password_lookup_finish()` returns `None` when there is simply nothing stored, and raises `GLib.Error` when something went wrong — a locked keyring the user refused to unlock, or no Secret Service at all, which is normal on a bare X session.

The full example is `examples/gtk4/desktop/passwords.py`.

8.7 Opening things

Shelling out to `xdg-open` is no longer necessary, and does not work in a sandbox. GTK 4.10 added launchers that go through the portal when there is one:

```
launcher = Gtk.UriLauncher(uri="https://gtk.org/")
launcher.launch(window, None, on_done)

launcher = Gtk.FileLauncher(file=Gio.File.new_for_path(path))
launcher.launch(window, None, on_done)          # open it
launcher.open_containing_folder(window, None, on_done)   # or show it in the file manager
```

Asynchronous, like every other dialog, with a `launch_finish()` that raises if it did not work. Passing the parent window lets the portal show its confirmation dialog on top of yours.

To find out what *would* open a file, or to offer a choice, `Gio.AppInfo` still answers:

```
info = Gio.AppInfo.get_default_for_type("text/plain", False)
print(info.get_display_name())
```

The full example is `examples/gtk4/desktop/launch-and-locate.py`.

8.8 Portals and the sandbox

Under Flatpak your process has no access to the file system outside its own directories, and none to the printer, camera, screen or location. Portals bridge that: a request goes over D-Bus to a service outside the sandbox, which asks the user, and the answer comes back as a specific grant.

The good news for this book is how little of it you have to write. These already go through a portal when there is one:

- `Gtk.FileDialog` — the user picks a file, and your sandbox is granted that file.
- `Gtk.PrintOperation` — the print dialog and the job.
- `Gtk.UriLauncher` and `Gtk.FileLauncher`.
- `Gio.Notification`.
- libsecret.

Which means a program written the way this chapter describes is already sandbox-ready. The rest — screenshots, screen casting, location, running in the background, autostart, inhibiting suspend — has no GTK wrapper, and you either speak to the D-Bus interface directly (see D-Bus) or use the `libportal` library, which wraps them all.

Two habits make the difference between a program that works sandboxed and one that does not:

Ask for files through a dialog rather than guessing paths. A grant follows what the user chose. `os.listdir(os.path.expanduser("~"))` does not, and returns almost nothing inside a sandbox.

Keep hold of the `Gio.File` you were given, not a path string you derived from it. Under the portal the path may exist only for as long as the grant does.

8.9 Summary

- The application id names your desktop file, icon, D-Bus name and notifications. Pick it once, in reverse DNS, and make everything match.
- GSettings keys are declared in a schema, which is compiled and installed. Missing schema means an abort, not an exception.
- `settings.bind()` connects a key to a widget property in both directions, and is most of what a preferences dialog needs.
- Use `GLib.get_user_*_dir()` rather than building paths from `~`. Cache is deletable; data is backed up.
- A `.desktop` file needs a matching icon name, valid categories and `update-desktop-database` afterwards — and notifications do not work without one.
- Passwords go in libsecret, asynchronously, never in GSettings.
- `Gtk.UriLauncher` and `Gtk.FileLauncher` replace `xdg-open` and work in a sandbox.
- GTK's own dialogs already go through portals; ask for files rather than guessing paths, and keep the `Gio.File`.

Audio and Video with GStreamer is next.

Chapter 9

Audio and Video with GStreamer

> Every listing in this chapter is a file under `examples/gtk4/multimedia/`. They are run on each build, against a three-second test clip the examples generate for themselves, so nothing here depends on a media file you have to find.

9.1 Introduction

GStreamer is the multimedia framework the Linux desktop is built on. It plays files, streams from the network, captures from a camera, transcodes, mixes, records, and can be assembled into things that are much stranger than any of that. This chapter covers the small part of it that an application usually needs.

Start with the good news: **for playing a file in a window, you may not need GStreamer at all**. GTK 4 has playback built in. `Gtk.MediaFile` is a media stream backed by GStreamer, and `Gtk.Video` is a widget that shows one. That is a page of code for a working video player, and it is where this chapter starts.

The rest of the chapter is the layer underneath, for when the built-in player is not enough: pipelines, the bus, inspecting files, and converting them.

Everything here is GStreamer **1.0**. The 0.10 series that the previous edition covered has been gone for over a decade; the module names, the element names and the state machine all changed, and nothing written for it will run.

```
# Debian, Ubuntu
sudo apt install python3-gi gir1.2-gstreamer-1.0 gir1.2-gst-plugins-base-1.0 \
    gstreamer1.0-plugins-good gstreamer1.0-plugins-bad gstreamer1.0-libav
```

The plugin packages matter more than the library. `good` is what you want, `bad` is where a lot of useful-but-less-polished elements live, and `libav` carries the codecs that make ordinary video files play.

`Gtk.MediaFile` needs one more package, and its absence is quiet:

```
sudo apt install libgtk-4-media-gstreamer
```

GTK's media support is a loadable backend, not part of the library. Without it `Gtk.Video` shows a "no media" symbol, `get_duration()` stays at zero and nothing is logged — the file looks unplayable when the problem is that GTK has nothing to play it with. If a file plays with `gst-launch-1.0` and not in your window, this is why.

9.2 A video player

```
media = Gtk.MediaFile.new_for_file(Gio.File.new_for_path(path))
media.set_loop(True)

video = Gtk.Video(media_stream=media)
media.play()
```

That plays a file. `Gtk.Video` draws the frames, handles fullscreen and shows its own controls if you do not supply any.

`Gtk.MediaStream` reports through **properties**, not signals, so watching it is `notify::`:

```
media.connect("notify::timestamp", self.on_timestamp)
media.connect("notify::error", self.on_error)
```

Positions and durations are in **microseconds**. `get_duration()` returns 0 until the file has been opened far enough to know, so guard the division:

```
duration = media.get_duration()
if duration <= 0:
    return
fraction = media.get_timestamp() / duration
```

Seeking is one call, and worth checking first — a stream from the network may not be seekable:

```
if media.is_seekable():
    media.seek(int(fraction * duration))
```

The one piece of fiddliness in a player is the position slider, because it is both an output and an input. Setting its value from the timestamp fires `value-changed`, which seeks, which changes the timestamp. Block your own handler while you write to it:

```
self.position.handler_block_by_func(self.on_seek)
self.position.set_value(fraction)
self.position.handler_unblock_by_func(self.on_seek)
```

Errors arrive on the `error` property rather than as an exception, since decoding happens on another thread:

```
def on_error(self, media, _pspec):
    error = media.get_error()
    if error is not None:
        self.status.set_text(f"error: {error.message}")
```

The full example is `examples/gtk4/multimedia/video-player.py`.

9.3 Pipelines

Under `Gtk.MediaFile` is a GStreamer pipeline, and three ideas cover most of what one is:

Elements do one thing each — read a file, demux a container, decode a stream, resample audio, put frames on screen. `Gst.ElementFactory.make("audiotestsrc", "source")` creates one.

Pads connect elements and negotiate the format that flows between them. Linking two elements fails if they cannot agree, which is why `link()` returns a boolean worth checking.

The bus carries messages back out of the pipeline — errors, warnings, state changes, end of stream — onto your main loop. Decoding happens on other threads; the bus is what makes that safe.

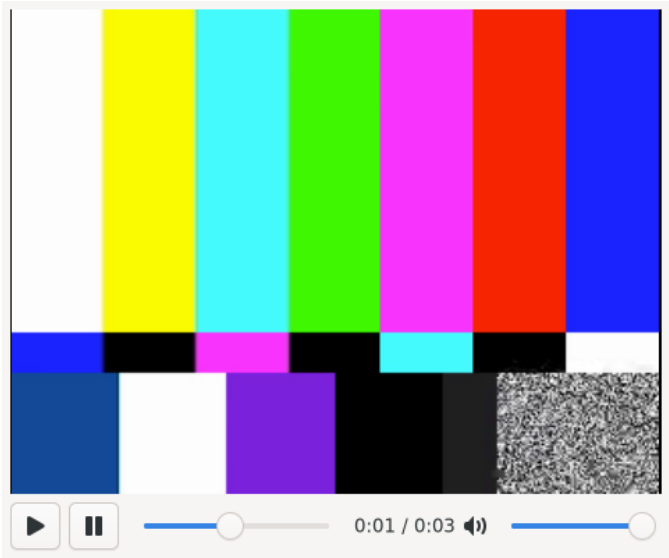

Figure 9.1: Gtk.Video playing the generated test clip

```
Gst.init(None)

pipeline = Gst.Pipeline.new("tone")
source = Gst.ElementFactory.make("audiotestsrc", "source")
convert = Gst.ElementFactory.make("audioconvert", "convert")
sink = Gst.ElementFactory.make("fakesink", "sink")

for element in (source, convert, sink):
    pipeline.add(element)

if not source.link(convert) or not convert.link(sink):
    raise SystemExit("the elements would not link")

pipeline.set_state(Gst.State.PLAYING)
```

`Gst.init(None)` has to run before anything else in GStreamer.

`Gst.ElementFactory.make()` returns `None` when the element is not installed — a missing plugin package, not a typo, most of the time. Check the result; the alternative is an `AttributeError` on `None` fifty lines later.

9.3.1 Building a pipeline from a string

Wiring elements up by hand is worth doing once. After that, `parse_launch()` takes the same syntax as the `gst-launch-1.0` command line:

```
pipeline = Gst.parse_launch(
    "videotestsrc num-buffers=90 ! video/x-raw,width=320,height=240 "
    "! vp8enc ! webmmux ! filesink location=sample.webm"
)
```

This is how you should prototype. Get it working on the command line with `gst-launch-1.0`, where you can iterate in seconds, then paste the string into `parse_launch()`. `Gst.parse_launch` accepts named elements (`webmmux name=mux`) so you can fetch one later with `pipeline.get_by_name("mux")` and set

properties or connect signals on it.

The `!` is a link, and a bare caps string between two elements (`! video/x-raw,width=320 !`) is a **capsfilter**: a constraint on what may be negotiated there. That is how you pin a resolution or a sample rate.

9.3.2 States

A pipeline moves through `NULL` → `READY` → `PAUSED` → `PLAYING`, and back down again. Two things about that are worth internalising:

`set_state()` can return `ASYNC`, meaning "started, watch the bus". It has not failed; it has not finished either. Use `get_state()` with a timeout if you really must wait.

Always come back to `NULL`. That is what releases the audio device, the file handles and the hardware decoder. A pipeline left in `PLAYING` at exit leaks all three, and on some systems the next program to want the sound card cannot have it.

```
finally:
    pipeline.set_state(Gst.State.NULL)
```

9.3.3 The bus

```
bus = pipeline.get_bus()
bus.add_signal_watch()
bus.connect("message", on_message, loop)
```

`add_signal_watch()` is what attaches the bus to the main loop. Without it the `message` signal never fires and you sit there wondering why nothing happens.

```
def on_message(_bus, message, loop):
    if message.type == Gst.MessageType.EOS:
        loop.quit()

    elif message.type == Gst.MessageType.ERROR:
        error, debug = message.parse_error()
        print(f"error: {error.message}")
        print(f"debug: {debug}")
        loop.quit()

    elif message.type == Gst.MessageType.STATE_CHANGED:
        if message.src is pipeline:      # every element reports its own
            old, new, _pending = message.parse_state_changed()
```

The `debug` string from `parse_error()` is the useful half. It names the element that failed and the file and line inside GStreamer, which is usually enough to identify which part of a pipeline is unhappy. Print it.

`STATE_CHANGED` arrives from every element in the pipeline, not just the pipeline itself. Filter on `message.src` unless you want a hundred lines of log.

For pipelines that run without a main loop — a script, a test — `timed_pop_filtered()` blocks instead:

```
message = bus.timed_pop_filtered(
    30 * Gst.SECOND, Gst.MessageType.EOS | Gst.MessageType.ERROR
)
```

The full example is `examples/gtk4/multimedia/pipeline-basics.py`.

9.4 What is in a file

`GstDiscoverer` opens a file, works out what is in it, and reports back without playing anything:

```
gi.require_version("GstPbutils", "1.0")
from gi.repository import GstPbutils

discoverer = GstPbutils.Discoverer.new(5 * Gst.SECOND)
info = discoverer.discover_uri(Gst.filename_to_uri(path))

print(info.get_duration() / Gst.SECOND, "seconds")
print(info.get_seekable())

for stream in info.get_stream_list():
    caps = stream.get_caps()
    print(GstPbutils.pb_utils_get_codec_description(caps))
```

Note `Gst.filename_to_uri()`: GStreamer works in URIs, and hand-building `"file://" + path` breaks on the first filename with a space or a non-ASCII character in it.

The stream objects are typed, so `isinstance` tells you what you have:

```
if isinstance(stream, GstPbutils.DiscovererVideoInfo):
    print(stream.get_width(), stream.get_height())
elif isinstance(stream, GstPbutils.DiscovererAudioInfo):
    print(stream.get_channels(), stream.get_sample_rate())
```

Tags — title, artist, album — hang off the individual streams. `DiscovererInfo.get_tags()`, which merged them all together, is deprecated.

The timeout is a hard limit and you want one. A discoverer given something it cannot make sense of will otherwise sit indefinitely.

The full example is `examples/gtk4/multimedia/discover.py`.

9.5 Missing codecs

When a file needs a plugin that is not installed, the result is not a crash. The discoverer returns `MISSING_PLUGINS` and hands you installer detail strings:

```
if info.get_result() == GstPbutils.DiscovererResult.MISSING_PLUGINS:
    for detail in info.get_missing_elements_installer_details():
        print(detail)
```

Those strings are not for humans. They go to `GstPbutils.install_plugins_async()`, which asks the distribution's package installer — PackageKit on most systems — to fetch and install the right package, then tells you whether to reload the registry:

```
GstPbutils.install_plugins_async(details, None, on_installed)

def on_installed(result, _data=None):
    if result == GstPbutils.InstallPluginsReturn.SUCCESS:
        Gst.update_registry()
```

A playing pipeline reports the same thing as a `MissingPluginMessage` on the bus, which you recognise with `GstPbutils.is_missing_plugin_message(message)`.

Whether this actually installs anything depends on the system: it needs a session helper, and inside a Flatpak there is nothing to install into — the runtime is fixed. Handle the failure case by telling the user

what is missing, which the `get_description()` on the message will phrase for you.

9.6 Converting a file

The pattern for any background media job: build a pipeline, watch the bus, poll for position, come back to `NULL`.

```
PIPELINE = """
  uridecodebin name=source uri={uri}
  theoraenc name=videoenc ! oggmux name=mux ! filesink location={out}
  vorbisenc name=audioenc ! mux.
  videoconvert name=videoconv ! videoenc.
  audioconvert name=audioconv ! audioenc.
"""
```

`uridecodebin` works out how to decode whatever it is given, so this converts anything the installed plugins can read. The price is that **its output pads do not exist until it has looked at the stream**, so the encoders are connected in a handler rather than in the pipeline string:

```
def on_pad_added(_element, pad, pipeline):
    caps = pad.get_current_caps() or pad.query_caps(None)
    kind = caps.to_string().split(",")[0]

    if kind.startswith("video/"):
        target = pipeline.get_by_name("videoconv")
    elif kind.startswith("audio/"):
        target = pipeline.get_by_name("audioconv")
    else:
        return

    pad.link(target.get_static_pad("sink"))
```

That `or pad.query_caps(None)` is worth keeping. A pad that has just appeared may have **no current caps** — nothing has flowed through it yet, so nothing has been negotiated — and an obvious-looking `if caps is None: return` then skips the stream and the pipeline dies with `streaming stopped, reason not-linked`. `query_caps()` asks what the pad is *willing* to carry, which is enough to tell audio from video.

9.6.1 uridecodebin, not uridecodebin3

The two look interchangeable. They are not, and the difference costs an afternoon if you meet it the hard way.

`uridecodebin3` is built for **playback**, where a stream selection mechanism decides which of several audio or subtitle tracks is actually decoded. Put it in front of a muxer and the pipeline builds perfectly, both pads appear, both link with `Gst.PadLinkReturn.OK` — and then EOS is never propagated downstream. The muxer never finalises the file, the progress query sticks at around 90%, and the job hangs until something kills it. There is no error and nothing on the bus.

The same pipeline with plain `uridecodebin` finishes in under a second:

```
uridecodebin3    -> STALLED    in  29.9s
uridecodebin     -> EOS        in   0.4s
```

So: `uridecodebin3` and `playbin3` for playing things, `uridecodebin` and `decodebin` for pipelines that write a file. When a job links up correctly and then simply never ends, this is the first thing to check.

Progress is a query, not a message:

```
ok, position = pipeline.query_position(Gst.Format.TIME)
ok2, duration = pipeline.query_duration(Gst.Format.TIME)
if ok and ok2 and duration > 0:
    print(f"{100 * position / duration:.1f}%")
```

Poll it from a `GLib.timeout_add()` a few times a second. Both return times in **nanoseconds** — `Gst.SECOND` is the conversion factor, and it is not the same unit `Gtk.MediaStream` uses, which is microseconds.

The full example is `examples/gtk4/multimedia/transcode.py`.

9.7 Video in your own widget

`Gtk.Video` covers playing a file. When the video comes from a pipeline you built — a camera, a stream, something with a filter in it — you need the frames in a widget of your own. The element for that is **`gtk4paintablesink`**, from the Rust plugin set:

```
sudo apt install gstreamer1.0-gtk4        # or gst-plugin-gtk4 from gst-plugins-rs
```

```
sink = Gst.ElementFactory.make("gtk4paintablesink", "sink")
paintable = sink.get_property("paintable")

picture = Gtk.Picture.new_for_paintable(paintable)
pipeline.set_property("video-sink", sink)
```

It hands GTK a `Gdk.Paintable`, which any `Gtk.Picture` can show — so the video composites with the rest of your interface, scales properly on a HiDPI screen, and can have widgets drawn over it.

This is the part that changed most since the previous edition. Embedding video used to mean catching a `prepare-window-handle` message and passing an X11 window id to the sink. That does not work on Wayland, it does not work with a scaled display, and it is not how it is done any more. If you find code calling `set_xwindow_id()`, it is from that era.

9.7.1 Let the compositor do the work

GTK 4.14. `Gtk.GraphicsOffload` needs it.

Playing video the ordinary way means every frame is uploaded, composited into the window with everything else, and drawn. `Gtk.GraphicsOffload` asks for the frames to go straight to a hardware plane in the compositor instead, skipping GTK's renderer entirely. Wrap the video widget in one:

```
offload = Gtk.GraphicsOffload(child=video)
offload.set_enabled(Gtk.GraphicsOffloadEnabled.ENABLED)
```

That is the whole change. On a laptop it is a visible saving in power; on a slow GPU it can be the difference between smooth playback and dropped frames.

It is a *request*, and the conditions are easy to break without noticing: the frames have to be in a format the compositor can scan out, nothing may be drawn on top of the widget, and it has to be rectangular and unrotated. Put a control overlay across the video and offloading quietly stops. Nothing fails when it does — GTK falls back to compositing normally — which is why it is safe to ask for and also why you should check with `GDK_DEBUG=offload` rather than assume.

`Gtk.Video`, `Gtk.Picture` and a `gtk4paintablesink` paintable all work with it.

9.8 Summary

- For playing a file, `Gtk.MediaFile` and `Gtk.Video` are the whole answer. Timestamps are microseconds; state arrives through `notify::`.
- `Gst.init(None)` first. Elements are made by factory and return `None` when the plugin is missing.
- Prototype with `gst-launch-1.0`, then paste the string into `Gst.parse_launch()`.
- `bus.add_signal_watch()` or no messages arrive. Print the debug half of `parse_error()`.
- Always return the pipeline to `NULL`.
- decodebin grows its pads at runtime; use `get_current_caps() or query_caps(None)` in `pad-added`.
- `uridecodebin3` is for playback. A pipeline that writes a file wants `uridecodebin`, or it links correctly and then never reaches EOS.
- Positions in GStreamer are nanoseconds (`Gst.SECOND`); in `Gtk.MediaStream` they are microseconds.
- Video in a custom widget is `gtk4paintablesink` and a `Gtk.Picture`, not an X11 window id.
- Wrap the video widget in a `Gtk.GraphicsOffload` to hand frames to the compositor. It fails silently back to normal rendering, so verify with `GDK_DEBUG=offload`.

D-Bus is next: talking to the rest of the session, and to programs that are not yours.

Chapter 10

D-Bus

> Every listing in this chapter is a file under `examples/gtk4/dbus/`. They are run on each build, under a private session bus, so the client and the service in the last two sections are known to talk to each other.

10.1 Introduction

D-Bus is how programs on a Linux desktop talk to each other. Your music player tells the shell what is playing; the shell asks your application to open a file; your program asks the desktop to show a notification, unlock a keyring or pick a file on its behalf. Almost everything in Desktop Integration is D-Bus with a wrapper over it.

There are two buses:

The session bus one per logged-in session, for the programs a user is running. This is where nearly everything in this chapter happens.

The system bus one per machine, for services that outlive a login — NetworkManager, systemd, UPower. Reading from it is usually allowed; changing things needs authorisation through polkit.

Use **GDBus**, which is part of GLib and therefore already imported. The old `dbus-python` module still exists and still appears in search results; it has its own main loop integration, its own type system and its own pitfalls, and there is no reason to start with it now.

```
from gi.repository import Gio, GLib

connection = Gio.bus_get_sync(Gio.BusType.SESSION, None)
```

10.2 The vocabulary

Five words, and then everything else is detail:

A **bus name** identifies a connection. Well-known names look like `org.gnome.Shell`; unique names look like `:1.42` and are handed out by the bus.

An **object path** looks like a file path: `/org/gnome/Shell`. One program can export many objects.

An **interface** is a named group of members: `org.freedesktop.DBus.Properties`. One object can implement several.

Methods are calls with a reply. **Signals** are broadcasts with no reply. **Properties** are values, read and written through a standard interface.

Signatures describe types: `s` string, `i` int32, `u` uint32, `b` boolean, `d` double, `o` object path, `v` variant, `as` array of strings, `a{sv}` a dictionary from string to variant — the shape almost every "options" argument has. `(is)` is a struct.

10.3 Looking before you write

Do not start by writing code. Start by looking at what is on the bus, with `gdbus`, `busctl` or the D-Spy application:

```
gdbus introspect --session --dest org.freedesktop.Notifications \
  --object-path /org/freedesktop/Notifications

busctl --user list
```

The same three moves in Python are listing the names, introspecting an object, and calling a method:

```
reply = connection.call_sync(
    "org.freedesktop.DBus", "/org/freedesktop/DBus", "org.freedesktop.DBus",
    "ListNames", None,
    GLib.VariantType("(as)"),          # the reply signature you expect
    Gio.DBusCallFlags.NONE, -1, None,
)
(names,) = reply.unpack()
```

That `(names,)` is not a typo. **A D-Bus reply is always a tuple**, even when it carries one value, so unpacking one result means unpacking a one-element tuple. Forgetting it gives you a tuple where you expected a list, and the error appears somewhere else entirely.

`Gio.DBusNodeInfo.new_for_xml()` turns the introspection XML into objects, which is how the graphical tools build their trees:

```
info = Gio.DBusNodeInfo.new_for_xml(xml)
for interface in info.interfaces:
    for method in interface.methods:
        print(method.name, [a.signature for a in method.in_args])
```

The full example is `examples/gtk4/dbus/explore-the-bus.py`.

10.4 Calling a service

`call_sync()` is fine for one call. For an object you use repeatedly, a **proxy** is much better: it fetches the introspection data, caches the properties, and turns method calls into ordinary Python calls.

```
proxy = Gio.DBusProxy.new_for_bus_sync(
    Gio.BusType.SESSION,
    Gio.DBusProxyFlags.NONE,
    None,                      # introspection data; None means fetch it
    "com.example.Counter",     # bus name
    "/com/example/Counter",    # object path
    "com.example.Counter",     # interface
    None,
)

print(proxy.Increment("(i)", 3))     # method name, signature, arguments
print(proxy.Describe())
```

That `"(i)"` is the signature of the **arguments**, and it is required because Python cannot tell an int32 from an int64 from a uint32. Get it wrong and the call fails with a type error rather than silently sending the wrong thing.

Two things about proxies surprise people:

Creating a proxy succeeds even when nobody is there. It is a placeholder that starts working when the service appears. To find out whether anything is home:

```
if proxy.get_name_owner() is None:
    print("not running")
```

Properties are cached, and the cache is only as fresh as the last `PropertiesChanged` signal.

```
proxy.get_cached_property("Value").unpack()
```

costs nothing because the value arrived with the proxy — but if the service changes it without announcing it, you will read the old value forever. This cuts both ways: when you *write* a service, emit `PropertiesChanged`, and when you use one that does not, call `Get` explicitly instead of trusting the cache.

10.4.1 Calling asynchronously

`call_sync()` and the proxy's attribute-style calls block until the reply comes back, and the default timeout is 25 seconds. In a program with a window that is 25 seconds of frozen interface. Anything triggered by a user action should be asynchronous:

```
def on_reset(proxy, result, _data=None):
    try:
        proxy.call_finish(result)
    except GLib.Error as error:
        print(f"failed: {error.message}")

proxy.call("Reset", None, Gio.DBusCallFlags.NONE, -1, None, on_reset)
```

Same shape as every other asynchronous call in this book: a callback, a `*_finish()` inside a `try`, and `GLib.Error` for the failure. A D-Bus call can fail because the service is not running, because it returned an error, because it took too long, or because you were not allowed — all of them arrive here.

The full example is `examples/gtk4/dbus/call-our-service.py`.

10.5 Listening

A proxy re-emits its object's signals as `g-signal`:

```
proxy.connect("g-signal", lambda p, sender, signal, params: print(signal, params.unpack()))
proxy.connect("g-properties-changed", on_properties_changed)
```

When you want to hear about something that is not tied to one object — or from a program that is not running yet — subscribe on the connection:

```
subscription = connection.signal_subscribe(
    None,                           # sender
    "org.freedesktop.DBus",         # interface
    "NameOwnerChanged",             # signal
    "/org/freedesktop/DBus",        # path
    None,                           # first argument must equal this
    Gio.DBusSignalFlags.NONE,
```

```
    on_name_owner_changed,
)
```

Any filter may be `None`, meaning "do not care", but the more of them you fill in the less traffic the bus sends you. `NameOwnerChanged` in particular fires constantly on a busy session.

For the common case — "tell me when this service comes and goes" — there is a helper that also handles the case where it is *already* running, which a plain subscription does not:

```
watch = Gio.bus_watch_name(
    Gio.BusType.SESSION, "com.example.Counter",
    Gio.BusNameWatcherFlags.NONE, on_appeared, on_vanished,
)
```

Keep the ids and undo both when you are finished: `connection.signal_unsubscribe(subscription)` and `Gio.bus_unwatch_name(watch)`.

The full example is `examples/gtk4/dbus/watch-signals.py`.

10.6 Being a service

Exporting your own object is three things: own a name, describe an interface, and answer calls.

Describe the interface in the same XML that `Introspect` returns:

```
<node>
  <interface name="com.example.Counter">
    <method name="Increment">
      <arg type="i" name="by" direction="in"/>
      <arg type="i" name="value" direction="out"/>
    </method>
    <method name="Reset"/>
    <signal name="Changed">
      <arg type="i" name="value"/>
    </signal>
    <property name="Value" type="i" access="read"/>
    <property name="Label" type="s" access="readwrite"/>
  </interface>
</node>
```

Register an object with one callback per job:

```
node = Gio.DBusNodeInfo.new_for_xml(INTERFACE_XML)
connection.register_object(
    PATH, node.interfaces[0],
    counter.on_method_call, counter.on_get_property, counter.on_set_property,
)
```

The method handler dispatches on the name:

```
def on_method_call(self, _connection, _sender, _path, _interface,
                   method, parameters, invocation):
    if method == "Increment":
        (by,) = parameters.unpack()
        self.value += by
        invocation.return_value(GLib.Variant("(i)", (self.value,)))
    elif method == "Reset":
        self.value = 0
```

```
        invocation.return_value(None)
    else:
        invocation.return_error_literal(
            Gio.dbus_error_quark(), Gio.DBusError.UNKNOWN_METHOD,
            f"no such method: {method}",
        )
```

Always answer the invocation, on every path through the handler, including the ones you did not expect. A caller that gets no reply does not get an error — it waits for its timeout and then gets a confusing one. `return_value(None)` is the reply for a method that returns nothing, and it is not optional.

The reply is a GVariant **tuple** again, so a method returning one integer returns `GLib.Variant("(i)", (value,))`.

Signals are emitted on the connection:

```
connection.emit_signal(
    None, PATH, NAME, "Changed", GLib.Variant("(i)", (self.value,))
)
```

And when a property changes, say so — nothing does it for you:

```
connection.emit_signal(
    None, PATH, "org.freedesktop.DBus.Properties", "PropertiesChanged",
    GLib.Variant("(sa{sv}as)", (NAME, {"Value": GLib.Variant("i", value)}, [])),
)
```

Leave that out and every client's cached copy stays at whatever it was when they connected. The three parts of the payload are the interface, the properties whose new values you are sending, and a list of properties that changed but whose values you are not sending.

10.6.1 Owning the name

You can call `Gio.bus_own_name()` directly, but if your program is a `Gio.Application` — or a `Gtk.Application` — it already owns a name: the application id **is** the bus name.

```
app = Gio.Application(application_id="com.example.Counter",
                      flags=Gio.ApplicationFlags.IS_SERVICE)
app.connect("startup", on_startup)          # register the object here
```

`app.get_dbus_connection()` gives you the connection to register on. `IS_SERVICE` means the process exists to serve, and `set_inactivity_timeout()` lets it exit when nothing has called it for a while — combined with a `.service` file, that is how a service gets started on demand rather than at login.

There is a bonus you get without asking. Every `Gio.Action` you add to a `Gtk.Application` is already exported over D-Bus, on the standard `org.gtk.Actions` interface. Which means this works against any GTK application, with no code on your side at all:

```
gdbus call --session --dest com.example.App --object-path /com/example/App \
  --method org.gtk.Actions.Activate quit "[]" "{}"
```

That is also how the shell shows your application's menu, and how a desktop file with `DBusActivatable=true` gets you started.

The full example is `examples/gtk4/dbus/export-a-service.py`.

10.7 Testing without a desktop

D-Bus code is easy to test, because a bus is cheap:

```
dbus-run-session -- python3 my-service.py
dbus-run-session -- sh -c "python3 export-a-service.py & sleep 1; python3 call-our-service.py"
```

`dbus-run-session` starts a private session bus, runs the command with `DBUS_SESSION_BUS_ADDRESS` pointing at it, and tears it down afterwards. Nothing else on the machine can see it, so tests cannot collide, and it is how the examples in this chapter are verified on each build.

10.8 Portals are D-Bus

The portals from the desktop integration chapter are ordinary D-Bus services on `org.freedesktop.portal.Desktop` at `/org/freedesktop/portal/desktop`, and anything without a GTK wrapper is reachable the way anything else is.

They have one unusual convention. A portal method does not return the answer; it returns an **object path for a request**, and the answer arrives later as a `Response` signal on that object. That is because a portal call may involve asking the user, which can take as long as the user takes. So the sequence is: subscribe to the response path, make the call, and wait for the signal.

This is exactly the sort of thing worth using a library for. libportal wraps every portal in a normal asynchronous API and has GObject introspection, so it is `Xdp` in PyGObject:

```
gi.require_version("Xdp", "1.0")
from gi.repository import Xdp

portal = Xdp.Portal.new()
portal.request_background(parent, "Syncing in the background", None,
                          Xdp.BackgroundFlags.AUTOSTART, None, on_done)
```

Screenshots, screen casting, location, inhibiting suspend, autostart, opening a URI, and the trash all live there.

10.9 Summary

- Use GDBus from GLib, not `dbus-python`.
- Session bus for the user's programs, system bus for the machine's.
- Every reply is a tuple: `(value,) = reply.unpack()`.
- Method arguments need an explicit signature, because Python's ints do not have one.
- A proxy exists even when the service does not; check `get_name_owner()`.
- Cached properties are only as fresh as the last `PropertiesChanged` — emit it from your services, and do not trust it from services that do not.
- Use the asynchronous calls in anything with a window. The default timeout is 25 seconds.
- In a method handler, answer the invocation on every path, including the unexpected ones.
- A `Gtk.Application`'s id is already a bus name, and its actions are already exported.
- `dbus-run-session` gives you a private bus for testing.

Animation and Transitions is next.

Chapter 11

Animation and Transitions

> Every listing in this chapter is a file under `examples/gtk4/animation/`. They are run on each build, so if one of them stops working the build says so.

11.1 Introduction

The previous edition of this book had a chapter on Clutter — a separate scene graph, with its own actors, its own stage and its own animation framework, bolted onto a GTK window. That is not how any of this works now.

Clutter is gone. It was folded into GNOME's compositor, deprecated as a public library, and the parts an application actually needed came back inside GTK 4. GTK now has a GPU-backed scene graph of its own (GSK), a frame clock, transforms on every widget, and — through libadwaita — a proper animation API. There is nothing left to bolt on.

So this chapter is about four ways to make something move, in the order you should reach for them:

1. **Let a container do it.** Stacks, revealers and navigation views animate their own changes.
2. **CSS.** Transitions and keyframes, for anything that is really a style change.
3. **`Adw.Animation`.** Timed or spring-driven, when you are animating a value.
4. **The frame clock.** When you are drawing every frame yourself.

Most applications never get past the first two.

11.2 Transitions you get for free

Nearly all the motion in a well-behaved GTK application is a container animating its own state change.

```
stack = Gtk.Stack()
stack.set_transition_duration(400)
stack.set_transition_type(Gtk.StackTransitionType.CROSSFADE)
stack.add_titled(first_page, "one", "One")
stack.add_titled(second_page, "two", "Two")
```

Changing the visible child now animates. The types include `CROSSFADE`, `SLIDE_LEFT_RIGHT`, `SLIDE_UP_DOWN`, `OVER_UP` and a rotation, and `Gtk.StackSwitcher` or `Adw.ViewSwitcher` will drive the stack for you.

`Gtk.Revealer` does the same for showing and hiding one thing:

```
revealer = Gtk.Revealer()
revealer.set_transition_type(Gtk.RevealerTransitionType.SLIDE_DOWN)
revealer.set_transition_duration(300)
revealer.set_child(extra_controls)

toggle.bind_property("active", revealer, "reveal-child",
                     GObject.BindingFlags.SYNC_CREATE)
```

That `bind_property` is worth noticing: the revealer's state *is* a property, so there is no handler to write at all.

libadwaita adds more of the same idea — `Adw.NavigationView` for push-and-pop navigation, `Adw.OverlaySplitView` for a sidebar that slides away on a narrow window, `Adw.Carousel` for swipeable pages. All animated, none of it your code.

The full example is `examples/gtk4/animation/transitions.py`.

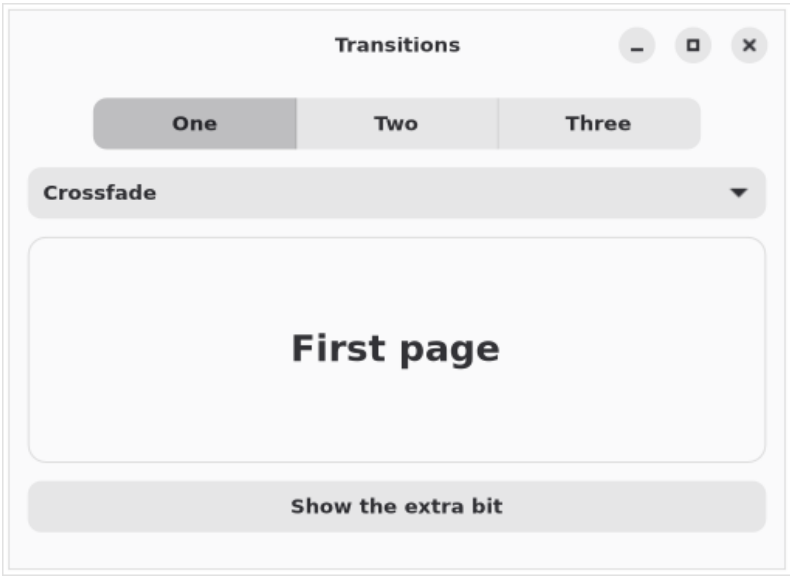

Figure 11.1: A stack with a switcher, and a revealer below it

11.3 CSS

GTK styles its widgets with a subset of CSS, and that subset includes transitions and keyframe animations. For anything that is really a *style* change, this is the least code:

```
.swatch {
    background: #3584e4;
    border-radius: 12px;
    transition: background 400ms ease-in-out,
                border-radius 400ms ease-in-out;
}

.swatch:hover      { background: #813d9c; }
.swatch.round      { background: #2ec27e; border-radius: 60px; }

@keyframes pulse {
```

```
    from { opacity: 1;    }
    50%  { opacity: 0.35; }
    to   { opacity: 1;    }
}

.pulsing { animation: pulse 1.2s ease-in-out infinite; }
```

Load it once, onto the display rather than onto a widget:

```
provider = Gtk.CssProvider()
provider.load_from_data(CSS)
Gtk.StyleContext.add_provider_for_display(
    Gdk.Display.get_default(), provider,
    Gtk.STYLE_PROVIDER_PRIORITY_APPLICATION,
)
```

After that, animating is `widget.add_css_class("round")` and `widget.remove_css_class("round")`. The hover state costs nothing at all.

Two things to know. GTK's CSS is a *subset* — there is no layout in it, no `display`, no `float`, no positioning; it styles widgets that GTK has already laid out. And the property names are GTK's, so it is worth reading the GTK CSS documentation rather than assuming the web's.

The GTK Inspector (`GTK_DEBUG=interactive python3 app.py`) has a CSS editor that applies changes live, which turns styling from a compile-and-look loop into something interactive.

The full example is `examples/gtk4/animation/css-animation.py`.

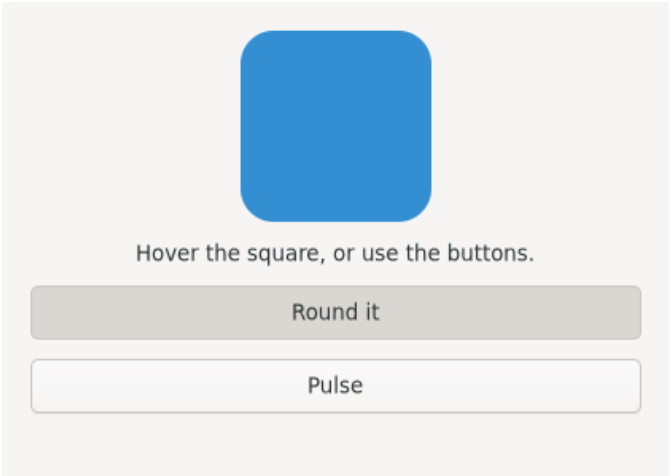

Figure 11.2: The swatch mid-transition, after the round class was added

11.4 AdwAnimation

When you are animating a *value* rather than a style, libadwaita has the API. An animation is three things: a widget, to borrow a frame clock from; a range of values; and a **target** that receives each value.

```
animation = Adw.TimedAnimation.new(
    widget, 0, 380, 900,                          # from, to, milliseconds
    Adw.CallbackAnimationTarget.new(self.on_value),
)
```

```
animation.set_easing(Adw.Easing.EASE_IN_OUT_CUBIC)
animation.play()
```

There are two kinds of target, and the second one is the reason to like this API:

```
Adw.CallbackAnimationTarget.new(fn)                    # fn(value) every frame
Adw.PropertyAnimationTarget.new(widget, "opacity")     # no callback at all
```

A property target writes straight to a GObject property, so fading a widget is an animation object and nothing else.

`Adw.TimedAnimation` has a fixed duration and an easing curve — around thirty of them, from `LINEAR` through `EASE_IN_OUT_CUBIC` to `EASE_OUT_BOUNCE`. It can also repeat and alternate, which covers "flash twice" in one object:

```
animation.set_alternate(True)
animation.set_repeat_count(2)
```

`Adw.SpringAnimation` has no duration. It simulates a spring and runs until it settles:

```
Adw.SpringAnimation.new(
    widget, 0, 380,
    Adw.SpringParams.new(0.6, 1, 180),    # damping ratio, mass, stiffness
    target,
)
```

A damping ratio below 1 overshoots and wobbles; exactly 1 is critically damped and arrives as fast as it can without overshooting; above 1 crawls in. Springs are the right choice for anything the user is *dragging*, because a spring can be handed a starting velocity — `set_initial_velocity()` — and continue naturally from the gesture that started it. That is why a flicked list decelerates the way it does.

The `done` signal fires when an animation finishes:

```
animation.connect("done", lambda _a: self.status.set_text("finished"))
```

Calling `play()` on a running animation restarts it. `pause()`, `resume()`, `reset()` and `skip()` do what they say, and `skip()` jumps straight to the end value while still emitting `done` — which is the correct way to cancel, because whatever the animation was setting ends up where it was going.

The full example is `examples/gtk4/animation/adwaita-animations.py`.

11.5 The frame clock

For a drawing that changes every frame — a visualiser, a game, a clock with a sweeping second hand — you want the frame clock directly:

```
def on_tick(self, widget, frame_clock):
    now = frame_clock.get_frame_time()          # microseconds, monotonic
    if self.start_time is None:
        self.start_time = now

    self.phase = ((now - self.start_time) % PERIOD_US) / PERIOD_US
    widget.queue_draw()
    return GLib.SOURCE_CONTINUE

widget.add_tick_callback(on_tick)
```

Do not animate with `GLib.timeout_add(16, ...)`. A timeout is not synchronised with the display: frames land slightly early or late, and the result judders in a way that is hard to see in a screenshot and

obvious in motion. The frame clock's `get_frame_time()` is the time the frame will be *displayed*, which is what makes motion smooth.

Compute position **from elapsed time**, never by adding a step per frame. A frame can be dropped, and a per-frame increment turns a dropped frame into a permanent drift; deriving the position from the clock makes a dropped frame invisible.

The callback runs until it returns `GLib.SOURCE_REMOVE`, and stops on its own when the widget is unmapped. Keep the id from `add_tick_callback()` if you want to stop it yourself with `remove_tick_callback()`.

The full example is `examples/gtk4/animation/frame-clock.py`.

11.6 Moving a whole widget

Every widget has a transform, applied by its parent when it is snapshotted. To move, rotate or scale one without touching its layout, override `do_snapshot()` and transform the snapshot before chaining up:

```
def do_snapshot(self, snapshot):
    snapshot.save()
    snapshot.translate(Graphene.Point().init(self.offset, 0))
    snapshot.rotate(self.angle)
    Gtk.Widget.do_snapshot(self, snapshot)
    snapshot.restore()
```

Combine that with an `Adw.CallbackAnimationTarget` that sets `self.angle` and calls `queue_draw()`, and you have any transform animation you like — running on the GPU, with no re-layout, at any scale factor.

This is what Clutter's actors were for, and it is now three lines in a widget you already have.

11.7 Respecting the user

Some people get motion sickness from animated interfaces, and every desktop has a setting for it. Honour it.

GTK's own transitions already do. If you write your own, check:

```
settings = Gtk.Settings.get_default()
if settings.get_property("gtk-enable-animations"):
    animation.play()
else:
    animation.skip()          # jump to the end value, no motion
```

`skip()` rather than not playing, so whatever the animation was setting still ends up where it should be.

In CSS, the same preference is the media query browsers use:

```
@media (prefers-reduced-motion: reduce) {
    .swatch  { transition: none; }
    .pulsing { animation: none; }
}
```

Two more habits worth keeping. Animation durations belong in the 150–400 ms range; anything slower stops feeling responsive and starts feeling broken. And never animate something the user is waiting on — a spinner during a two-second load is fine, a 600 ms slide before a menu opens is not.

11.8 Colour scheme, accent and contrast

Reduced motion is one of four preferences in this area, and the other three are about appearance rather than movement: whether the interface is dark, which accent colour the user picked, and whether they need higher contrast. All three arrive through `Adw.StyleManager`, and libadwaita applies all three to its own widgets without you doing anything.

```
style = Adw.StyleManager.get_default()

style.get_dark()                            # is the interface dark right now
style.get_accent_color()                    # Adw.AccentColor, since 1.6
style.get_high_contrast()
style.get_system_supports_color_schemes() # False on a desktop that has no setting
```

Each is a property, so each has a `notify::` signal, and connecting to those three signals is the entire job of "responding to the theme". There is nothing to poll:

```
for property_name in ("dark", "accent-color", "high-contrast"):
    style.connect(f"notify::{property_name}", self.on_style_changed)
```

The default is to follow the desktop, and the default is correct. `Adw.ColorScheme.DEFAULT` means "whatever the user's session says". If you offer a preference at all, offer three states — system, light, dark — with system selected, and store the choice in GSettings. An application that opens dark on a light desktop because its author prefers dark is an application that has substituted its taste for the user's.

```
style.set_color_scheme(Adw.ColorScheme.DEFAULT)      # follow the desktop
style.set_color_scheme(Adw.ColorScheme.FORCE_DARK)   # the user asked for dark
```

The `PREFER_` variants are requests the desktop may decline; the `FORCE_` ones are not. Use `FORCE_` for an explicit user choice and `DEFAULT` for everything else.

11.8.1 What this means for anything you drew yourself

The stylesheet reaches widgets. It does not reach a `Gtk.DrawingArea`, a Cairo chart or a custom `snapshot()`, and that is where theme support usually breaks: a window that goes dark around a graph that stays white.

Two rules cover it. Repaint when the theme changes — `queue_draw()` from the `notify::` handler above. And take your colours from the theme rather than hardcoding them:

```
rgba = style.get_accent_color().to_standalone_rgba()
cr.set_source_rgba(rgba.red, rgba.green, rgba.blue, rgba.alpha)
```

`to_standalone_rgba()` rather than the raw accent, because it returns the colour adjusted for the current light or dark background — the brand blue is not legible on both. For the ordinary foreground and background, look up the named colours with `widget.get_color()` and the CSS variables libadwaita defines (`--accent-bg-color`, `--window-fg-color` and the rest) instead of picking greys by eye.

High contrast is the one people skip, and it is the one with a legal shape in some markets. When `get_high_contrast()` is true, drop decorative low-contrast fills and give things borders.

The full example is `examples/gtk4/animation/style-manager.py`.

11.9 Summary

- Clutter is gone. GTK 4 has the scene graph, the frame clock and the transforms; libadwaita has the animation API.

- Reach for a container transition first, CSS second, `Adw.Animation` third, the frame clock last.
- `Adw.PropertyAnimationTarget` animates a GObject property with no callback at all.
- Springs take an initial velocity, which is what makes gesture-driven motion feel continuous.
- Use `add_tick_callback()`, not a 16 ms timeout, and derive position from elapsed time rather than accumulating per frame.
- Transform a whole widget by transforming its snapshot before chaining up.
- Check `gtk-enable-animations` and `prefers-reduced-motion`, and cancel with `skip()`.
- `Adw.StyleManager` reports dark, accent and high contrast as properties. Connect to their `notify::` signals; do not poll.
- Follow the desktop's colour scheme by default; a preference has three states, not two.
- Anything you draw yourself has to repaint on a theme change and take its colours from the theme. `to_standalone_rgba()` gives an accent that works on both backgrounds.

Embedding Web Content is next.

Chapter 12

Embedding Web Content

> Every listing in this chapter is a file under `examples/gtk4/web/`. They are run on each build, so if one of them stops working the build says so.

12.1 Introduction

The previous edition of this book explained how to embed Mozilla with `gtkmozembed`, and how to put an Internet Explorer control in a GTK window on Windows. Both are gone, and have been for a long time. What is left is **WebKitGTK**, and it is in much better shape than either of them ever was: a current browser engine, multi-process, sandboxed, maintained alongside Safari's, and packaged by every distribution.

You would embed it for three reasons: to show documentation or a preview inside your application, to render content that is genuinely HTML — an email, a Markdown preview, a report — or to build an application whose interface is HTML and whose logic is Python.

```
# Debian, Ubuntu
sudo apt install gir1.2-webkit-6.0

# Fedora
sudo dnf install webkitgtk6.0
```

12.1.1 Get the version right

```
gi.require_version("WebKit", "6.0")
from gi.repository import WebKit
```

This trips up everyone once, because there are three namespaces in circulation:

Namespace	Toolkit	Status
`WebKit 6.0`	GTK 4	what you want
`WebKit2 4.1`	GTK 3	the previous generation
`WebKit2 4.0`	GTK 3, libsoup 2	end of life

Most tutorials and most Stack Overflow answers are `WebKit2`, and the class names inside are nearly identical, so code copied from one to the other looks right and fails at `require_version`. The API version and the library version are also not the same number — `WebKit 6.0` reports itself as WebKit 2.52.

12.2 A browser

```
view = WebKit.WebView()
view.load_uri("https://gnome.org/")
```

That is a working browser widget. Everything else is chrome around it.

State arrives as **properties**, so it is `notify::` again:

```
view.connect("notify::uri", self.on_uri_changed)
view.connect("notify::estimated-load-progress", self.on_progress)
view.connect("notify::title", self.on_title_changed)
```

History is not a property, though, and this is a trap worth naming. `can_go_back()` and `can_go_forward()` are **methods**, not properties, so `bind_property("can-go-back", …)` fails at runtime with "has no property called can-go-back". Refresh those from `load-changed` instead:

```
def on_load_changed(self, view, event):
    self.back.set_sensitive(view.can_go_back())
    self.forward.set_sensitive(view.can_go_forward())
    if event == WebKit.LoadEvent.FINISHED:
        self.progress.set_visible(False)
```

`load-changed` reports `STARTED`, `REDIRECTED`, `COMMITTED` and `FINISHED`. `FINISHED` fires whether the load worked or not — a failure is `load-failed` first, then `FINISHED`.

`load_uri()` wants a real URI. A bare `gnome.org` is not one, so a browser has to fix up what the user types:

```
if "://" not in text:
    text = "https://" + text
```

The full example is `examples/gtk4/web/browser.py`.

12.3 Deciding what the page may do

`decide-policy` fires before anything is navigated to, opened in a new window, or downloaded, and it is where an embedded view stops being a browser:

```
def on_decide_policy(self, _view, decision, decision_type):
    if decision_type == WebKit.PolicyDecisionType.NAVIGATION_ACTION:
        uri = decision.get_navigation_action().get_request().get_uri()
        if not uri.startswith("https://docs.example.com/"):
            decision.ignore()               # refuse it
            Gtk.UriLauncher(uri=uri).launch(self, None, None)   # or hand it over
            return True
    decision.use()
    return True
```

Three answers: `use()` allows it, `ignore()` refuses it, and `download()` turns it into a download. You must call one of them and return `True`, or the decision is left hanging and the page stops.

For a view that shows *your* content, this is where you keep it that way: allow your own origin, and send everything else to the user's real browser. A help viewer that will happily navigate to any link in the document is a help viewer that can be pointed anywhere.

12.4 Running JavaScript

```
view.evaluate_javascript(
    "document.title",
    -1,                  # length; -1 means nul-terminated
    None, None, None,    # world, source uri, cancellable
    self.on_evaluated,
)


def on_evaluated(self, view, result, _data=None):
    try:
        value = view.evaluate_javascript_finish(result)
    except GLib.Error as error:
        return
    print(value.to_string() if value.is_string() else value.to_json(0))
```

Asynchronous, like everything else, and the result is a `JSCValue` rather than a Python object. `is_string()`, `is_number()`, `is_array()` and friends ask what it is; `to_string()` and `to_json(0)` get it out. `to_json()` is the pragmatic choice for anything structured — take the JSON and hand it to `json.loads()`.

Never build a script by formatting a string with user data into it. It is `eval` with the same consequences it has anywhere else. Pass values in by defining a function in the page and calling it with `JSON.stringify`-safe arguments, or by setting them through the message channel below.

12.5 Letting the page call back

The other direction is a script message handler. Register a name, and it appears in the page as `window.webkit.messageHandlers.<name>`:

```
manager = WebKit.UserContentManager()
manager.register_script_message_handler("fromPage", None)
manager.connect("script-message-received::fromPage", self.on_message)

view = WebKit.WebView(user_content_manager=manager)
```

```
window.webkit.messageHandlers.fromPage.postMessage(
    JSON.stringify({clicks: count})
);
```

```
def on_message(self, _manager, value):
    payload = value.to_string()          # a JSCValue again
```

The content manager has to be attached **when the view is created** — it is a construct-only property, so creating the view first and adding the manager afterwards silently does nothing.

`UserContentManager` also injects scripts and stylesheets into every page a view loads:

```
manager.add_script(WebKit.UserScript.new(
    "console.log('injected before the page runs');",
    WebKit.UserContentInjectedFrames.TOP_FRAME,
    WebKit.UserScriptInjectionTime.START,
    None, None,
))
```

`START` runs before the page's own scripts, which is where you set up an API for the page to use. `END` runs after the document is parsed, which is where you modify what is there.

The full example is `examples/gtk4/web/javascript-bridge.py`, which does both directions.

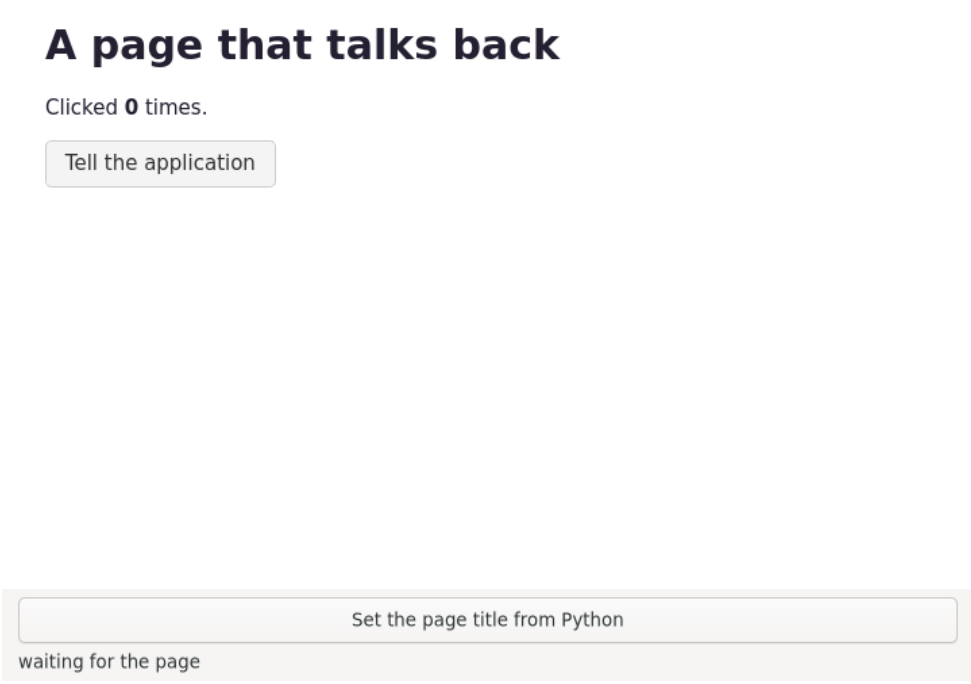

Figure 12.1: A page posting a message to Python, and Python reading a value back

12.5.1 Treat the page as untrusted

A message handler is a hole through the sandbox that you made on purpose. Once a page can call `postMessage`, anything that ends up in that page can call it — an advert, an injected script, a compromised CDN, a link the user followed.

So: only expose handlers to content you control, validate every payload as if it came off the network, keep handlers narrow and specific (`save_document`, not `run_command`), and never accept a file path from the page and act on it without checking. Combine it with `decide-policy` so the view cannot navigate somewhere you did not intend in the first place.

12.6 Loading your own HTML

```
view.load_html(PAGE, "file:///")
```

The second argument is the **base URI**, and leaving it `None` causes puzzling failures later: the page is treated as having no origin, so relative URLs do not resolve and anything origin-checked — `fetch`, local storage, modules — is refused.

For a view that shows content you ship, `Gtk.Template`-style resource loading is the usual approach: put the HTML, CSS and images in a GResource and register a custom URI scheme so the page can load them:

```
session = view.get_network_session()
session.get_website_data_manager()      # cookies, cache, storage all live here
```

`WebKit.NetworkSession` is where cookies, the cache and website data live in WebKit 6.0. An **ephemeral** session — `WebKit.NetworkSession.new_ephemeral()` — keeps nothing on disk, which is what you want for a preview pane or anything private.

12.7 Settings worth changing

```
settings = view.get_settings()
settings.set_enable_developer_extras(True)          # right-click, Inspect
settings.set_enable_write_console_messages_to_stdout(True)
settings.set_enable_javascript(False)               # for a preview pane
settings.set_user_agent_with_application_details("MyApp", "1.0")
```

The developer tools are the same Web Inspector Safari uses, and having them in your own application while debugging a bridge is worth the one line.

12.8 The sandbox

WebKitGTK runs the web content in a separate, sandboxed process, using bubblewrap and user namespaces. That is a feature and occasionally an obstacle: in a container without permission to create namespaces, the web process cannot start and the view dies with

```
bwrap: Creating new namespace failed: Operation not permitted
```

The fix in a container is to allow user namespaces. There *is* an environment variable that turns the sandbox off, and its name — `WEBKIT_DISABLE_SANDBOX_THIS_IS_DANGEROUS` — is the documentation. It is acceptable in a throwaway test container, which is how the examples in this chapter are checked on machines that cannot do namespaces. It is not acceptable anywhere a real page will be loaded.

12.9 Summary

- `gi.require_version("WebKit", "6.0")` for GTK 4. `WebKit2` is the GTK 3 one, and is what most of the search results are about.
- `WebKit.WebView()` plus `load_uri()` is a browser; the rest is chrome.
- Load state is properties and `load-changed`, but `can_go_back()` is a method — there is nothing to bind to.
- `decide-policy` is where you keep an embedded view from wandering. Answer with `use()`, `ignore()` or `download()` and return `True`.
- `evaluate_javascript()` is asynchronous and returns a `JSCValue`; `to_json(0)` is the practical way out.
- A script message handler is how the page calls you. Attach the `UserContentManager` when you create the view, and treat everything that comes through it as untrusted input.
- `load_html()` needs a base URI or the page has no origin.

Internationalization is next.

Chapter 13

Internationalization

> Every listing in this chapter is a file under `examples/gtk4/i18n/`. The example ships a real German translation, and the build extracts, compiles and runs it, so the pipeline described here is a pipeline that works.

13.1 Introduction

Internationalization is making a program *able* to be translated; localization is translating it. The mechanism is **gettext**, it has not changed in twenty years, and most of what the previous edition said about it is still true. What has changed is the tooling around it: `libglade` is gone, `gtk-builder-convert` is gone, `intltool` has been superseded by `xgettext` reading `.ui` files directly, and Meson generates the whole pipeline from a few lines.

Three things have to line up, and getting two of them right produces an untranslated program with no error message at all:

1. Strings are marked in the source and extracted into a `.po` file.
2. The compiled `.mo` file is installed where gettext looks for it.
3. The program binds its text domain — **twice**, once for Python and once for the C library, which is the part that is specific to GTK.

13.2 Marking strings

```
import gettext

_ = gettext.gettext
ngettext = gettext.ngettext

self.set_title(_("Greeter"))
```

`_()` is a convention, not a language feature: it is short, and `xgettext` looks for it by name. Some of the shapes that come up:

```
_("Your name")                                  # a string to translate now
N_("Deferred")                                  # mark it, translate it later
ngettext("one file", "{n} files", n).format(n=n) # a plural
pgettext("verb", "Greet")                       # disambiguate by context
```

`N_()` is for strings that have to be marked where they are *defined* but translated where they are *used* —

a table of menu labels at module level, for instance. It does nothing at runtime; it just gives `xgettext` something to find.

13.2.1 Plurals

```
ngettext("You have said hello once.",
         "You have said hello {count} times.",
         self.count).format(count=self.count)
```

Do not write `if n == 1`. English has two plural forms, Japanese has one, Polish has three, Arabic has six, and the rule for choosing between them lives in the `Plural-Forms` header of each `.po` file. `ngettext` applies whichever rule the translator's language needs, and your code never learns what it was.

13.2.2 Context

The same English word is often two different words elsewhere. "Open" as a verb on a button is not "Open" as a state in a status line; German needs *Öffnen* and *Geöffnet*. `pgettext` lets the translator tell them apart:

```
def pgettext(context, message):
    lookup = f"{context}\x04{message}"
    translated = gettext.gettext(lookup)
    return message if translated == lookup else translated
```

That `\x04` is how gettext encodes a context, and the fallback is what makes an untranslated string come back without the context glued to the front of it.

13.2.3 Formatting

Never build a sentence by concatenation.

```
_("Hello, {name}!").format(name=name)      # good
_("Hello, ") + name + "!"                  # untranslatable
```

Word order differs between languages, and a translator has to be able to move the parts around. Named placeholders are better than positional ones for the same reason: `{name}` tells the translator what will be there, and can be reordered without counting.

13.2.4 Comments for translators

A translator sees the string and nothing else. If it is ambiguous, say so:

```
# Translators: this is the window title.
self.set_title(_("Greeter"))
```

`xgettext --add-comments=Translators:` copies those into the `.po` file. It costs a line and saves a round trip.

13.3 The GTK-specific part

This is the section people skip and then spend an afternoon on.

```
import gettext
import locale

locale.setlocale(locale.LC_ALL, "")
```

```
gettext.bindtextdomain(DOMAIN, LOCALE_DIR)
gettext.textdomain(DOMAIN)

locale.bindtextdomain(DOMAIN, LOCALE_DIR)      # the C library's copy
locale.textdomain(DOMAIN)
```

Three separate traps live in those six lines.

`setlocale(locale.LC_ALL, "")` is required. Without it the C library stays in the `C` locale, and nothing is translated no matter how correct everything else is. The empty string means "take it from the environment".

The domain has to be bound twice. `gettext.bindtextdomain()` sets it for Python's `_()` calls. `locale.bindtextdomain()` sets it for the C library — and that is the one `GtkBuilder` uses when it translates `translatable="yes"` strings out of a `.ui` file. Bind only the Python one and your code is translated while your interface files are not, which is a confusing half-working state.

On Windows, `locale.bindtextdomain` does not exist. Python's `locale` module does not expose it there, and you have to call into `libintl-8.dll` with `ctypes`:

```
import ctypes
libintl = ctypes.cdll.LoadLibrary("libintl-8.dll")
libintl.bindtextdomain(DOMAIN.encode(), LOCALE_DIR.encode())
libintl.bind_textdomain_codeset(DOMAIN.encode(), b"UTF-8")
```

13.4 Strings in .ui files

```
<interface domain="greeter">
  <template class="GreeterAboutWindow" parent="GtkWindow">
    <property name="title" translatable="yes">About</property>
    <child>
      <object class="GtkLabel">
        <property name="label" translatable="yes"
                  comments="Shown under the title">A small example.</property>
      </object>
    </child>
    <child>
      <object class="GtkButton">
        <property name="label" translatable="yes" context="verb">Close</property>
      </object>
    </child>
  </template>
</interface>
```

`translatable="yes"` marks it, `context=` is the equivalent of `pgettext`, and `comments=` is the equivalent of a `Translators:` comment. The `domain` attribute on `<interface>` says which text domain the file belongs to.

Modern `xgettext` reads `.ui` files directly — you list them alongside the `.py` files and it works out the rest. That is what `intltool` used to be for, and it is why a modern project has no `intltool-extract`, no `.h` files generated from XML, and no `.in` files.

13.5 The pipeline

Four commands, in order. The example wraps them in `update-translations.sh`; a real project has Meson generate them.

Extract the strings into a template:

```
xgettext \
  --from-code=UTF-8 \
  --add-comments=Translators: \
  --keyword=_ \
  --keyword=N_ \
  --keyword=ngettext:1,2 \
  --keyword=pgettext:1c,2 \
  --output=po/greeter.pot \
  greeter.py window.ui
```

The `--keyword` arguments are how `xgettext` learns your calls. `ngettext:1,2` says arguments one and two are the singular and the plural; `pgettext:1c,2` says argument one is a context and argument two is the string. Omit them and plurals and contexts are silently missed.

Start a language, once:

```
msginit --locale=de --input=po/greeter.pot --output=po/de.po
```

Merge later changes into existing translations:

```
msgmerge --update --backup=none po/de.po po/greeter.pot
```

`msgmerge` keeps the translations that still apply and marks the ones whose source string changed as **fuzzy** — translated, but needing review. Fuzzy entries are *not used* at runtime, which is deliberate: a stale translation is worse than an untranslated string.

Compile to the binary format gettext reads:

```
msgfmt --check --output-file=locale/de/LC_MESSAGES/greeter.mo po/de.po
```

That path is not negotiable. gettext looks in `<locale dir>/<language>/LC_MESSAGES/<domain>.mo` and nowhere else. Nearly every "my translations do not load" is a wrong path, a wrong domain, or a `.mo` that was never recompiled after the `.po` changed.

`--check` catches the mistakes that would otherwise show up as a crash in one language: a translation with a `%s` where the original had `%d`, or a missing placeholder.

13.6 Testing it

```
LANGUAGE=de python3 greeter.py
LANGUAGE=de LC_ALL=de_DE.UTF-8 python3 greeter.py
```

`LANGUAGE` looks like the convenient one: it overrides only the message language, takes a colon-separated list of fallbacks, and does not need the locale to be generated on your system.

It is not enough on its own, and the way it fails is confusing. The C library ignores `LANGUAGE` when the locale is `C`, and `setlocale(LC_ALL, "")` leaves you in `C` if no `LC_*` variable is set. Python's `gettext` module reads `LANGUAGE` itself, so it does not care — which means `LANGUAGE=de` alone gives you a program whose Python strings are German and whose `.ui` strings are English:

```
$ LANGUAGE=de example-app
TITLE:  Example App           # from window.ui - not translated
```

```
LABEL:  2-mal gezählt.        # from ngettext() in Python - translated

$ LC_ALL=de_DE.UTF-8 LANGUAGE=de example-app
TITLE:  Beispielanwendung
LABEL:  2-mal gezählt.
```

So test with a real locale. `locale -a` lists what you have and `sudo locale-gen de_DE.UTF-8` adds one. If you see exactly the split above — code translated, interface files not — the cause is either this or a missing `locale.bindtextdomain`, and they look identical from the outside.

The example in this chapter builds its widgets in Python rather than from a `.ui` file, so `LANGUAGE` on its own is enough for it — which is why both of these are the same program with one environment variable changed:

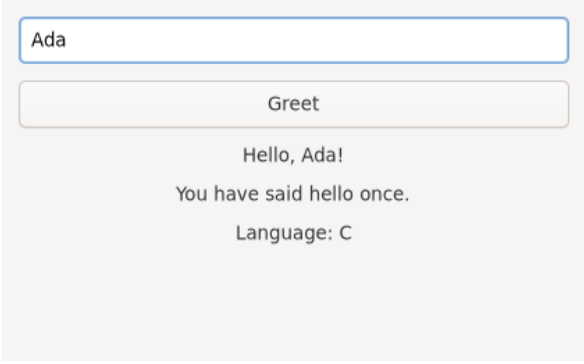

Figure 13.1: The greeter in its original English

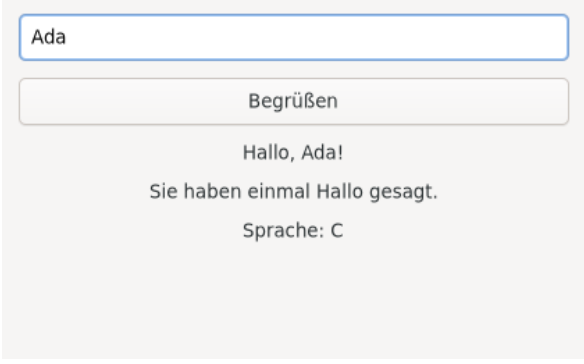

Figure 13.2: The same window under LANGUAGE=de

Note that the plural is not a suffix on a number: *"You have said hello once"* becomes *"Sie haben einmal Hallo gesagt"*, a different sentence, which is what `ngettext` is for.

Two habits worth building. Check a **pseudolocale** before you have translators: `msgfmt` a `.po` where every string is wrapped in brackets, and anything unbracketed in the running application is a string you forgot to mark. And check a language whose words are much longer than English — German is the traditional choice — to find the buttons your layout cannot grow to fit.

13.7 More than words

Translation is the biggest part of this, not all of it.

Dates, numbers and currency follow the locale, not the language. Use `GLib.DateTime.format()` with the `%x` and `%X` placeholders, or Python's `locale.format_string()` and `babel`, rather than assembling them yourself.

Right-to-left languages need the whole interface mirrored, and GTK does it for you — provided you never wrote `LEFT` or `RIGHT`. Use `Gtk.Align.START` and `Gtk.Align.END`, `set_margin_start()` and `set_margin_end()`, and the mirroring is free. Test it without knowing any Arabic:

```
GTK_DEBUG=interactive python3 app.py      # the Inspector can flip direction
```

Sorting is not `sorted()`. `GLib.utf8_collate_key()` gives you a key that sorts the way the user's locale expects, which is the difference between an ä next to an a and an ä at the end of the alphabet.

Icons and colours carry meanings that do not travel. So does text in images — which is a good reason to draw text with Pango rather than shipping a picture of it.

13.8 Summary

- `_()`, `N_()`, `ngettext()` and `pgettext()` mark strings; format with named placeholders and never concatenate.
- Bind the text domain **twice**: `gettext.bindtextdomain` for Python and `locale.bindtextdomain` for the C library, or `.ui` files stay untranslated.
- `locale.setlocale(locale.LC_ALL, "")` or none of it works.
- `xgettext` reads `.ui` files directly; `intltool` is not needed any more.
- `--keyword` arguments teach `xgettext` about your plurals and contexts.
- Fuzzy entries are deliberately ignored at runtime.
- `.mo` files go in `<locale dir>/<language>/LC_MESSAGES/<domain>.mo`, exactly.
- Test with `LANGUAGE=de`, a pseudolocale, and a right-to-left language.
- Use `START`/`END` rather than `LEFT`/`RIGHT` and mirroring is free.

Testing is next, and then Packaging and Distribution, which is where the translations, the desktop file, the icon and the schema all get installed together.

Chapter 14

Testing

> Every listing in this chapter is a file under `examples/gtk4/testing/`. The tests in it run on each build, along with the tests of this book's own examples.

14.1 Introduction

Desktop applications have a reputation for being untestable, and it is mostly deserved — but the reason is not that widgets are hard to drive. It is that the interesting logic has usually been written *inside* the widgets, where nothing can reach it without a display, a main loop and a synthetic mouse.

So this chapter is less about testing frameworks than about one structural decision, made early, which is what everything else depends on:

Keep your rules out of your widgets.

Once that is true, most of your tests are ordinary Python tests with no GTK in them at all, and the small number that genuinely need a window are testing wiring rather than behaviour. Get this wrong and no amount of test tooling rescues you; get it right and you barely need any.

The example is a to-do list in three files:

`tasklist.py` A `TaskList` and a `Task`. GObject, so the interface can bind to it — but it imports `Gio` and `GObject`, never `Gtk`. This is where every rule lives: what a valid title is, what counts as a duplicate, how many tasks are left.

`window.py` The widgets. Every method is a translation between a widget and the model, and none of them decides anything.

`app.py` The `Adw.Application`. Small on purpose, because a test wants to build a `Window` without an application wrapped around it.

14.2 Setting up

The awkward part of testing PyGObject is not pytest. It is that PyGObject comes from your distribution and pytest comes from PyPI, and a virtual environment does not contain the former:

```
python3 -m venv --system-site-packages venv
./venv/bin/pip install pytest
```

`--system-site-packages` is the whole trick. Without it the environment cannot see `gi`, `import gi` fails, and the usual reaction is to `pip install PyGObject`, which tries to compile a binding against your system GTK and fails differently. The same flag is what you want with `uv venv --system-site-packages` if that is your tool.

Then run them the way you would run any tests:

```
cd examples/gtk4/testing
../../../venv/bin/python -m pytest
```

14.3 Testing the logic

With the rules in `tasklist.py`, this is just pytest:

```
def test_a_blank_title_is_refused(tasks):
    with pytest.raises(ValueError):
        tasks.add("   ")
    assert tasks.titles() == []


def test_duplicates_are_refused(tasks):
    tasks.add("Write the tests")
    with pytest.raises(ValueError, match="already on the list"):
        tasks.add("Write the tests")
    assert len(tasks.titles()) == 1
```

No display, no main loop, no fixtures beyond a fresh `TaskList`. These run in milliseconds and they are where most of your tests should be.

14.3.1 Testing that a property notifies

There is one thing worth testing here that is specific to GObject, and it catches a bug that is otherwise very hard to find.

`TaskList.remaining` is a computed property — it counts the unfinished tasks. GObject does not know that, so nothing emits `notify::remaining` on its own; every method that could change the answer has to say so. Miss one and the property still returns the right number whenever anyone asks, but the label bound to it stops updating. That presents as a display bug, and you will look for it in the wrong file.

Record the notifications and assert on them:

```
def notifications_of(source, property_name):
    seen = []
    source.connect(f"notify::{property_name}",
                   lambda obj, _pspec: seen.append(obj.get_property(property_name)))
    return seen


def test_remaining_notifies_when_a_task_is_finished(tasks):
    task = tasks.add("one")
    seen = notifications_of(tasks, "remaining")

    task.done = True
    assert seen == [0]
```

That test fails unless `TaskList` is listening to its own children:

```
task.connect("notify::done", lambda *_: self.notify("remaining"))
```

Delete that one line from `tasklist.py` and exactly two tests fail — this one, and the window test that checks the status label. Every other test still passes, which is a fair picture of how much of the suite a missing notification touches, and why you want the test rather than the luck.

The same technique works for your own signals: connect, do the thing, assert on what was recorded. Assert on the *count* as well as the values — a property that notifies twice per change is a property that will make something flicker.

14.4 Testing widgets

Some things only exist in the interface: that pressing Enter reaches the model, that a binding is connected, that a refused entry keeps its text. Those need widgets.

Two pieces of setup make it possible.

Initialise GTK once. `Adw.init()` (or `Gtk.init()`) has to happen before any widget is constructed, and it is not safe to call per test. A session-scoped autouse fixture is the right shape:

```
@pytest.fixture(scope="session", autouse=True)
def gtk_initialised():
    Adw.init()
```

Have a way to turn the main loop. Tests never call `app.run()`, so nothing iterates the main loop and anything deferred to an idle callback never happens. Turn it by hand, with a limit so that a test which would otherwise hang fails instead:

```
def run(iterations=200):
    context = GLib.MainContext.default()
    turned = 0
    while context.pending() and turned < iterations:
        context.iteration(False)
        turned += 1
    return turned
```

`context.iteration(False)` is the important half: `False` means *do not block*. With `True` a test with nothing pending waits forever.

14.4.1 Pressing things

You do not need synthetic input events. Emit the signal the input would have caused:

```
def test_typing_and_activating_adds_a_task():
    tasks = TaskList()
    window = Window(tasks=tasks)

    window.entry.set_text("Write the tests")
    window.entry.emit("activate")        # what pressing Enter does

    assert tasks.titles() == ["Write the tests"]
    assert window.entry.get_text() == ""
```

`emit("activate")` runs the same handler a keypress would, synchronously, with no display interaction at all. `button.emit("clicked")` is the same idea, and `action.activate(None)` is how you exercise something bound to a menu item without opening the menu.

Note what that test asserts: that the model got the task, and that the entry cleared. It does not re-test that blank titles are refused — that lives next door in `test_tasklist.py`, where it costs nothing to run.

14.4.2 Lists build their rows lazily

`Gtk.ListView` does not create a row per item. It creates the rows it needs during a layout pass and recycles them, which means a freshly constructed list view contains no rows at all and a test that walks

its children finds nothing.

Force a layout:

```
window.present()
window.list_view.allocate(400, 400, -1, None)
pump()
```

Then the children exist and can be walked with `get_first_child()` and `get_next_sibling()`. This is the point where widget tests start costing more than they return, so use them for the binding rather than for the contents:

```
task.done = True
assert check.get_active() is True      # the model moved the widget

check.set_active(False)
assert task.done is False              # and the widget moved the model
```

14.5 Running without a display

Widget tests need a display, and CI does not have one. There are two answers and the second is better.

Xvfb, a fake X server, is the traditional one, and it is what this book's own build uses because a couple of its examples want a real compositor's worth of behaviour:

```
xvfb-run -a python3 -m pytest
```

The broadway backend needs no X at all — GTK renders to HTML over a socket, and for a test that never looks at pixels that is enough. It needs its daemon running first, which is the part that catches people out; set only the environment variables and every widget test fails with `RuntimeError: Gtk couldn't be initialized`:

```
gtk4-broadwayd :5 &
GDK_BACKEND=broadway BROADWAY_DISPLAY=:5 python3 -m pytest
```

It is lighter than Xvfb and it is one package rather than an X server.

Since GNOME 50 there is no X11 session at all, so `xvfb-run` is XWayland compatibility rather than a representative environment. It is fine for tests; do not conclude anything from it about how your application behaves on a real desktop. If you need that, run a headless compositor — `weston --backend=headless-backend.so` or `mutter --headless` — which is what GNOME's own integration tests do.

Whichever you pick, a session bus is worth having too, or anything that owns a name or talks to a portal fails in ways that have nothing to do with your test:

```
dbus-run-session -- xvfb-run -a python3 -m pytest
```

14.6 Smoke tests

There is a cheaper test than any of the above, and for a book — or any project with a lot of example code — it catches more per line than everything else combined: **start the program and shut it down again.**

That is what `tools/smoke-test.py` in this repository does to every listing in this book. It replaces the blocking main loop with one that quits after 300 ms, runs each example in its own interpreter, and reports anything that raised on the way up.

It finds import errors, constructor arguments that were renamed, methods that no longer exist, signals that no longer exist, and anything that throws while the first window is built. That is most of what breaks when a toolkit moves underneath you, and none of it needs a single assertion.

One detail in it is worth stealing, because without it the whole thing quietly reports success:

```
crashed = "Traceback (most recent call last)" in result.stderr
if result.returncode == 0 and not crashed:
    ...
```

An exception inside a GTK signal handler does not propagate. GTK prints the traceback and carries on, and the process still exits 0. If you check only the exit status, a program whose every button handler raises looks perfectly healthy. Treat a traceback on stderr as a failure.

14.7 Driving the whole application

Between "call a handler" and "test nothing" there is a middle option: run the real application and drive it from a timeout.

```
def on_started():
    window = app.get_active_window()
    window.entry.set_text("something")
    window.entry.emit("activate")
    assert window.tasks.titles() == ["something"]
    app.quit()
    return GLib.SOURCE_REMOVE


GLib.timeout_add(100, on_started)
app.run([])
```

Everything is real: the application, the activate handler, the window. Keep a small number of these for the paths that matter, and remember that an assertion which fails inside that callback is an assertion inside a GTK callback — swallowed, per the previous section. Record the failure and re-raise it after `run()` returns.

For genuinely external testing — driving the application through the accessibility bus the way a screen reader does — the tool is **dogtail**, which talks AT-SPI. It tests the real thing including the accessibility layer, which is a useful side effect, and it is slow and brittle enough that it should be a handful of tests rather than a suite.

14.8 Types and static checks

A type checker is the test you do not have to write, and PyGObject's reputation for being uncheckable is out of date.

The introspected modules are generated at import time, so a checker cannot see into them on its own. Install the stubs:

```
pip install pygobject-stubs --no-cache-dir --config-settings=config=Gtk4,Gdk4,Soup3
```

The `config` setting picks which GTK version's stubs to generate; without it you may get GTK 3 signatures and a great deal of confident nonsense. Then mypy or pyright will catch a renamed method, a missing argument and a `set_child()` on a widget that does not have one, before you run anything.

Annotate your own code — this book's examples are annotated throughout:

```
def on_bind(self, _factory: Gtk.SignalListItemFactory,
            item: Gtk.ListItem) -> None:
```

Two rough edges are worth knowing before you turn it on, because both produce a wall of errors on correct code the first time.

Optional returns need narrowing. A great many GTK getters are typed as returning `None` as well as the thing you asked for, and they are right to be:

```
check = item.get_child()      # Widget | None
task = item.get_item()        # GObject.Object | None
check.set_label(task.title)   # error: Item "None" has no attribute "set_label"
```

Narrow it rather than silencing it. An `assert isinstance()` documents what you already know, and turns the rest of the method from unchecked to checked:

```
assert isinstance(check, Gtk.CheckButton)
assert isinstance(task, Task)
```

`**kwargs: object` does not survive contact with a GObject constructor. The pass-through `**kwargs` that every widget subclass in this book uses has to be typed `Any`; `object` makes the checker try every property of the parent class against it and reject all of them.

Then two things stay unchecked whatever you do. Signal handler signatures are not verified, because `connect()` takes a string and a callable — a handler with the wrong parameters is still a runtime error. And `GObject.Property` is opaque to type checkers, so `task.done` is not known to be a `bool`. Neither is a reason to skip the rest.

Add **ruff** for the things a type checker does not cover — unused imports, unused variables, import order — and run all of it in CI beside the tests.

14.9 What to test

A rough order of return, best first:

1. **The rules, with no GTK involved.** Cheap, fast, and where the bugs that matter live.
2. **Property notification and signals.** Specific to GObject, easy to get wrong, invisible until something stops updating.
3. **Smoke tests.** Enormous coverage per line of test code.
4. **Wiring, in widget tests.** That handlers reach the model and bindings are connected.
5. **A few whole-application paths**, driven from a timeout.
6. **AT-SPI tests**, if the application is big enough to warrant them.

And what not to bother with: that GTK works. A test asserting that `Gtk.Label.set_text()` changed the label's text is testing someone else's library. Test the code you wrote.

14.10 Summary

- The decision that makes an application testable is keeping the rules out of the widgets. Everything else is technique.
- `python3 -m venv --system-site-packages` — PyGObject comes from the distribution and a plain venv cannot see it.
- Test that computed properties *notify*, not only that they compute. A missed `self.notify()` is a display bug you will look for in the wrong file.
- `Adw.init()` once, session-scoped, before any widget exists.
- Turn the main loop by hand with `context.iteration(False)` and an iteration limit, so a stuck test fails rather than hangs.

- Emit signals instead of synthesising input: `entry.emit("activate")`.
- `Gtk.ListView` builds rows during layout, so allocate and pump before looking for them.
- Headless means Xvfb or `GDK_BACKEND=broadway`; since GNOME 50 neither is a real desktop, so do not draw conclusions from them beyond "it ran".
- An exception in a GTK callback is printed and swallowed, and the process still exits 0. Treat a traceback on stderr as a failure.
- `pygobject-stubs` with `config=Gtk4` makes mypy and pyright useful; signal handler signatures and `GObject.Property` types stay unchecked.

Packaging and Distribution is next, and it is the last chapter of Part I.

Chapter 15

Packaging and Distribution

> The example in this chapter is a complete, installable application under `examples/gtk4/packaging/`. It is built, validated, installed and run on each build of this book, so the layout described here is one that works.

15.1 Introduction

A Python GTK application is not one file. By the time it is finished it is code, `.ui` files, an icon, a desktop file, a settings schema, translations and a piece of AppStream metadata — and every one of those has a place it must be installed to before it works. The desktop file has to be in `applications`, the schema has to be compiled, the `.mo` files have to be in `<lang>/LC_MESSAGES`, and the icon has to be named after the application id.

That is what this chapter is about: what has to be installed where, the build system that does it, and the two ways applications actually reach users.

The previous edition had a chapter here on IronPython and Gtk#. It is gone: it was never a supported way to write GTK applications, and the projects behind it have moved on.

15.2 What has to be installed

For the application id `com.example.ExampleApp`, under a prefix of `/usr` or `~/.local`:

What	Where
The launcher	`bin/example-app`
The Python modules	`share/example-app/exampleapp/`
The compiled resources	`share/example-app/example-app.gresource`
The desktop file	`share/applications/com.example.ExampleApp.desktop`
The icon	`share/icons/hicolor/scalable/apps/com.example.ExampleA`
The settings schema	`share/glib-2.0/schemas/com.example.ExampleApp.gschema.`
The AppStream metadata	`share/metainfo/com.example.ExampleApp.metainfo.xml`
The translations	`share/locale/<lang>/LC_MESSAGES/example-app.mo`

Plus three caches that have to be regenerated afterwards, or the desktop does not notice any of it: `glib-compile-schemas`, `gtk-update-icon-cache` and `update-desktop-database`.

Notice how much of that is the application id repeated. That is the point of picking it carefully in Desktop Integration.

15.3 Meson

GNOME builds with **Meson**, and so should a Python application that wants to fit in — not because Python needs compiling, but because Meson already knows all of the above.

```
project('example-app', version: '1.0.0', meson_version: '>= 1.0.0')

app_id = 'com.example.ExampleApp'

python = import('python').find_installation('python3')
gnome = import('gnome')
i18n = import('i18n')

prefix = get_option('prefix')
bindir = prefix / get_option('bindir')
datadir = prefix / get_option('datadir')
localedir = prefix / get_option('localedir')
pkgdatadir = datadir / meson.project_name()

subdir('src')
subdir('data')
subdir('po')

gnome.post_install(
  glib_compile_schemas: true,
  gtk_update_icon_cache: true,
  update_desktop_database: true,
)
```

`gnome.post_install()` is the three cache updates, and it is one line rather than a post-install script everyone forgets to write.

Make those paths **absolute** — `prefix / get_option('datadir')`, not `get_option('datadir')`. The relative form works fine as an install directory and then quietly breaks the launcher, which ends up looking for `share/example-app/example-app.gresource` relative to whatever directory the user happened to be in. Meson still honours `DESTDIR` when installing to an absolute path, so nothing is lost.

15.3.1 The generated launcher

The one file that has to know where things went is generated at build time:

```
configure_file(
  input: 'example-app.in',
  output: 'example-app',
  configuration: {
    'PYTHON': python.full_path(),
    'VERSION': meson.project_version(),
    'localedir': localedir,
    'pkgdatadir': pkgdatadir,
  },
  install: true,
  install_dir: bindir,
  install_mode: 'rwxr-xr-x',
)
```

```
#!@PYTHON@
pkgdatadir = '@pkgdatadir@'
localedir = '@localedir@'

sys.path.insert(1, pkgdatadir)

resource = Gio.Resource.load(os.path.join(pkgdatadir, 'example-app.gresource'))
resource._register()

locale.setlocale(locale.LC_ALL, '')
gettext.bindtextdomain('example-app', localedir)
gettext.textdomain('example-app')
locale.bindtextdomain('example-app', localedir)     # for the .ui files
locale.textdomain('example-app')

from exampleapp import application
sys.exit(application.main(VERSION))
```

Everything the application needs to find is settled here, once, from values the build system knows. Nothing downstream guesses a path from `__file__`.

The double `bindtextdomain` is the one from Internationalization, and this is where it belongs.

15.3.2 GResource

`.ui` files, CSS, icons and any other data are compiled into a single resource bundle:

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/com/example/ExampleApp">
    <file preprocess="xml-stripblanks">window.ui</file>
  </gresource>
</gresources>
```

```
gnome.compile_resources(
  'example-app', 'example-app.gresource.xml',
  gresource_bundle: true, install: true, install_dir: pkgdatadir,
)
```

Once the bundle is registered, everything in it is addressed by URI:

```
@Gtk.Template(resource_path="/com/example/ExampleApp/window.ui")
class Window(Adw.ApplicationWindow):
    __gtype_name__ = "ExampleAppWindow"
```

One file to install instead of a directory of them, no path lookups at runtime, and `Gio.File.new_for_uri("resource://` works for anything else you put in there.

15.3.3 Validate at build time

The mistakes in this chapter's file formats are all caught by a validator, and a validator you have to remember to run is one you will not run:

```
test('validate-desktop', desktop_file_validate, args: [desktop_file])
test('validate-metainfo', appstreamcli, args: ['validate', '--no-net', metainfo_file])
test('validate-schema', glib_compile_schemas, args: ['--strict', '--dry-run', dir])
```

`meson test` now checks all three. Finding out that your AppStream metadata is invalid from a rejected Flathub submission is a slower way to learn the same thing.

15.4 AppStream metadata

The `.metainfo.xml` is what a software centre shows: name, summary, description, screenshots, release notes, content rating. Flathub will not accept an application without it, and GNOME Software will not show one.

```
<component type="desktop-application">
  <id>com.example.ExampleApp</id>
  <name>Example App</name>
  <summary>An application that installs itself properly</summary>

  <metadata_license>CC0-1.0</metadata_license>
  <project_license>MIT</project_license>

  <description>
    <p>A minimal GTK 4 application, packaged the way GNOME applications are.</p>
  </description>

  <launchable type="desktop-id">com.example.ExampleApp.desktop</launchable>
  <content_rating type="oars-1.1"/>

  <releases>
    <release version="1.0.0" date="2026-01-15">
      <description><p>First release.</p></description>
    </release>
  </releases>
</component>
```

The `<id>` must match the application id and the `<launchable>` must match the desktop file, or the store shows an application that cannot be launched. `<content_rating>` is required even when the answer is "nothing to declare" — an empty `oars-1.1` element means exactly that. The `<summary>` is one line, no full stop, and must not repeat the name.

The summary and description are translated the same way everything else is: `i18n.merge_file()` merges the `.po` translations into the installed file.

15.5 Flatpak

For a Python GTK application, Flatpak is the way to reach users directly. It ships the GNOME runtime with it, so you are not at the mercy of which GTK version a distribution has, and it sandboxes the result.

```
{
    "id": "com.example.ExampleApp",
    "runtime": "org.gnome.Platform",
    "runtime-version": "50",
    "sdk": "org.gnome.Sdk",
    "command": "example-app",

    "finish-args": [
        "--share=ipc",
        "--socket=wayland",
        "--socket=fallback-x11",
```

```
        "--device=dri"
    ],

    "modules": [
        {
            "name": "example-app",
            "buildsystem": "meson",
            "sources": [{ "type": "dir", "path": "." }]
        }
    ]
}
```

```
flatpak install flathub org.gnome.Platform//50 org.gnome.Sdk//50
flatpak-builder --user --install --force-clean build com.example.ExampleApp.json
flatpak run com.example.ExampleApp
```

The GNOME runtime already contains Python, PyGObject, GTK 4, libadwaita, GStreamer and WebKitGTK, which is most of this book — so a plain GTK application often needs no extra modules at all. Dependencies from PyPI go in as their own modules; `flatpak-pip-generator` writes those for you.

`finish-args` is the sandbox's permissions, and the interesting thing is how few you need. There is no `--filesystem=home` in the list above, and there should not be: the file dialog goes through the portal, so the user granting access to a file *is* the permission. Every `finish-args` line you add is a warning triangle on your Flathub page, and most of them are avoidable by using the APIs in the desktop integration chapter rather than going around them.

15.6 The other routes

Distribution packaging. A `.deb` or an RPM built from the same Meson project. Best integration, most work, and you are unlikely to do it yourself — if your application is worth packaging, a distribution maintainer will usually do it, provided your build is a normal one. That is another argument for Meson.

A wheel on PyPI. Fine for a library, awkward for an application. `pip install` does not install desktop files, icons or schemas, and it cannot install PyGObject's dependency on GTK — that has to come from the system. Reasonable for a developer tool; not for something with an icon in the launcher.

Windows and macOS. Possible, not pleasant. On Windows the practical route is MSYS2, which packages GTK 4 and PyGObject, with `cx_Freeze` or PyInstaller for the bundle and Inno Setup for the installer. On macOS it is Homebrew or `jhbuild`, plus `py2app`. In both cases you are shipping the whole GTK stack yourself, native file dialogs are GTK's rather than the platform's, and the result does not look at home. If cross-platform look and feel is a requirement, that is a real argument for Qt — which is what Part II is about.

15.7 Try it

```
cd examples/gtk4/packaging
meson setup builddir --prefix=$HOME/.local
meson compile -C builddir
meson test -C builddir
meson install -C builddir
example-app
```

Installing to `~/.local` needs no root and puts everything where the desktop already looks. To see the translations, use a real locale rather than only `LANGUAGE`:

```
LC_ALL=de_DE.UTF-8 LANGUAGE=de example-app
```

15.8 Summary

- An application is code plus a desktop file, icon, schema, resources, translations and AppStream metadata, each with a required location.
- Meson knows all of them. `gnome.post_install()` regenerates the three caches.
- Make the paths absolute in `meson.build`, or the generated launcher looks for its resources relative to the current directory.
- Compile `.ui` files and other data into a GResource and address them by URI.
- Validate the desktop file, the metadata and the schema as part of `meson test`.
- AppStream metadata is not optional for a software centre, and its ids must match.
- Flatpak against the GNOME runtime is the direct route to users; keep `finish-args` short by using portals instead of filesystem access.

That is the end of Part I. Part II covers the same ground again with Qt 6 and PySide6.

Appendix A

Book Text Licenses

The text of this book — every chapter, and the figures drawn for it — is licensed under the **Creative Commons Attribution-ShareAlike 4.0 International** licence (CC BY-SA 4.0). You may copy it, redistribute it, translate it and build on it, including commercially, provided you give credit and license what you build under the same terms.

The sample code is under a different, more permissive licence; see Source Code License.

> **Note on the licence version.** Editions up to 0.13 were released under CC BY-SA 3.0; version 1.0 onward is 4.0. The 4.0 licences are drafted to work in every jurisdiction without a ported variant, they cover database rights, and they give you thirty days to fix a violation rather than terminating permanently on the spot. Copies of the earlier editions distributed under 3.0 keep that licence, and 3.0 already permitted redistributing adaptations under a later version with the same licence elements, so material can be carried forward either way.

The canonical text is at https://creativecommons.org/licenses/by-sa/4.0/legalcode.en, and is reproduced in full below.

A.1 Attribution-ShareAlike 4.0 International

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

A.1.1 Section 1 – Definitions

- **a.** *Adapted Material* means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
- **b.** *Adapter's License* means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.
- **c.** *BY-SA Compatible License* means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.

- **d.** *Copyright and Similar Rights* means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.
- **e.** *Effective Technological Measures* means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.
- **f.** *Exceptions and Limitations* means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.
- **g.** *License Elements* means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.
- **h.** *Licensed Material* means the artistic or literary work, database, or other material to which the Licensor applied this Public License.
- **i.** *Licensed Rights* means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.
- **j.** *Licensor* means the individual(s) or entity(ies) granting rights under this Public License.
- **k.** *Share* means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.
- **l.** *Sui Generis Database Rights* means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.
- **m.** *You* means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

A.1.2 Section 2 – Scope

- **a. License grant.**
  - **1.** Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
    - **A.** reproduce and Share the Licensed Material, in whole or in part; and
    - **B.** produce, reproduce, and Share Adapted Material.
  - **2. Exceptions and Limitations.** For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.
  - **3. Term.** The term of this Public License is specified in Section 6(a).
  - **4. Media and formats; technical modifications allowed.** The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.
  - **5.** *Downstream recipients.*
    - **A.** *Offer from the Licensor – Licensed Material.* Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
    - **B.** *Additional offer from the Licensor – Adapted Material.* Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.

        * **C.** *No downstream restrictions.* You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.
    - **6.** *No endorsement.* Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).
- **b. Other rights.**
    - **1.** Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
    - **2.** Patent and trademark rights are not licensed under this Public License.
    - **3.** To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

A.1.3 Section 3 – License Conditions

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

- **a. Attribution.**
    - **1.** If You Share the Licensed Material (including in modified form), You must:
        * **A.** retain the following if it is supplied by the Licensor with the Licensed Material:
            · **i.** identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
            · **ii.** a copyright notice;
            · **iii.** a notice that refers to this Public License;
            · **iv.** a notice that refers to the disclaimer of warranties;
            · **v.** a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
        * **B.** indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
        * **C.** indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
    - **2.** You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
    - **3.** If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
- **b. ShareAlike.** In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.
    - **1.** The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.
    - **2.** You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.
    - **3.** You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

A.1.4 Section 4 – Sui Generis Database Rights

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

- **a.** for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
- **b.** if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and
- **c.** You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

A.1.5 Section 5 – Disclaimer of Warranties and Limitation of Liability

- **a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.**
- **b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.**
- **c.** The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

A.1.6 Section 6 – Term and Termination

- **a.** This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
- **b.** Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:
  - **1.** automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
  - **2.** upon express reinstatement by the Licensor.

  For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.
- **c.** For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
- **d.** Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

A.1.7 Section 7 – Other Terms and Conditions

- **a.** The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
- **b.** Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

A.1.8 Section 8 – Interpretation

- **a.** For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
- **b.** To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.
- **c.** No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
- **d.** Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

A.2 About Creative Commons

The paragraph below is Creative Commons' own notice. It does not form part of the public license.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

Appendix B

Source Code License

Every example in this book — everything under `examples/` in the repository, and every listing reproduced in the text — is released under the MIT license. Copy it, change it, and use it in your own programs, commercial or not. You do not have to credit this book, though it is appreciated.

The book's **text** is under a different, more restrictive licence; see Book Text Licenses.

> **Note on the previous edition.** Editions up to 0.13 released their sample code under the LGPL v3. Those examples targeted PyGTK and GTK 2 and have been removed; they remain available under the LGPL in this repository's git history. All the code in this edition is new and is MIT.

B.1 The MIT License

Copyright (c) 2008–2026 Peter Gill

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B.2 What this does not cover

The libraries the examples use have their own licences, and they are not MIT:

GTK 4, GLib, GObject, Pango, GdkPixbuf LGPL v2.1 or later. Linking to them from your own program — which is what PyGObject does — does not require your program to be free software, but modifying the libraries themselves does.

libadwaita LGPL v2.1 or later.

Cairo LGPL v2.1 or the Mozilla Public License 1.1, your choice.

GStreamer LGPL v2.1 or later for the framework. **Individual plugins vary**, and some encoders and decoders carry patent considerations in some jurisdictions. Check the licence of the specific plugins you ship, particularly if you are distributing a bundle rather than depending on system packages.

WebKitGTK LGPL v2.1 and BSD.

PySide6, used in Part II LGPL v3 or a commercial Qt licence. The LGPL option has real conditions attached when you bundle Qt into an application rather than linking against a system copy — read them before you ship one.

If you are shipping a Flatpak, the runtime carries most of this for you and `flatpak-builder` records the licences of what it built.

Appendix C

Migrating from PyGTK

Earlier editions of this book taught PyGTK: GTK 2 through the `pygtk` bindings. That stack is gone. PyGTK has had no release since 2011, GTK 2 stopped getting fixes, and the bindings do not exist for Python 3 in any supported form.

This appendix is a translation table. It will not port a program for you, but it will tell you what each idiom you remember is called now, and which of them have no replacement because the pattern itself was dropped.

Everything below assumes the modern imports:

```
import gi

gi.require_version("Gtk", "4.0")
gi.require_version("Adw", "1")
from gi.repository import Adw, Gio, GLib, Gtk
```

C.1 The shape of a program

PyGTK You built a `gtk.Window`, packed it, called `show_all()`, then `gtk.main()`, and connected `destroy` to `gtk.main_quit`.

GTK 4 You build a `Gtk.Application`, create a `Gtk.ApplicationWindow` in its `activate` handler, call `present()`, and let `app.run()` return when the last window closes.

The application object is not optional ceremony. It gives you the single-instance check, actions, keyboard accelerators, the menu bar, D-Bus activation and session integration — all of which PyGTK programs had to arrange by hand or do without.

```
# then
win = gtk.Window()
win.connect("destroy", gtk.main_quit)
win.show_all()
gtk.main()

# now
def on_activate(app):
    Gtk.ApplicationWindow(application=app).present()

app = Gtk.Application(application_id="com.example.App")
app.connect("activate", on_activate)
```

```
sys.exit(app.run(sys.argv))
```

C.2 Imports and versions

`import pygtk; pygtk.require("2.0"); import gtk` becomes `import gi; gi.require_version("Gtk", "4.0"); from gi.repository import Gtk`.

The `require_version` call has to come before the import, not after, and it is needed for every library with more than one version installed. Skipping it does not fail cleanly — it picks a version, warns, and crashes later.

Names lost their lowercase module: `gtk.Window` is `Gtk.Window`, `gtk.HBox` is `Gtk.Box`, `gobject` is `GObject`, and the parts of GLib that PyGTK exposed as `gobject.timeout_add` are now `GLib.timeout_add`.

C.3 Containers and visibility

`show_all()` is gone Widgets are visible when created. Hide what you do not want with `set_visible(False)`.

`container.add(child)` is gone A widget that holds one child has `set_child()`. `Gtk.Window`, `Gtk.Button`, `Gtk.ScrolledWindow`, `Gtk.Frame` all work this way.

`box.pack_start(child, expand, fill, padding)` is gone Use `box.append(child)` or `box.prepend(child)`, and move the three arguments onto the child: `child.set_hexpand(True)`, `child.set_halign(...)`, `child.set_margin_start(...)`.

`gtk.HBox` / `gtk.VBox` are gone One class with an orientation: `Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=6)`.

`gtk.Table` is gone `Gtk.Grid`, with `attach(child, column, row, width, height)`.

`widget.destroy()` on a child is gone Remove it from its parent instead — `box.remove(child)` or `parent.set_child(None)`. Windows still have `close()`.

C.4 Widgets that were renamed or replaced

PyGTK	GTK 4
`gtk.RadioButton`	`Gtk.CheckButton` joined with `set_group()`
`gtk.ComboBox`, `gtk.ComboBoxText`	`Gtk.DropDown`
`gtk.TreeView` + `gtk.ListStore`	`Gtk.ColumnView` / `Gtk.ListView` + `Gio.ListStore`
`gtk.MessageDialog`	`Gtk.AlertDialog`
`gtk.FileChooserDialog`	`Gtk.FileDialog`
`gtk.ColorSelectionDialog`	`Gtk.ColorDialog`
`gtk.FontSelectionDialog`	`Gtk.FontDialog`
`gtk.Statusbar`	`Adw.Toast`, or a label in the header bar
`gtk.Table`	`Gtk.Grid`
`gtk.Alignment`	`halign` / `valign` / margins on the child
`gtk.EventBox`	event controllers, added to any widget
`gtk.Arrow`, `gtk.HSeparator`	`Gtk.Image` with an icon name, `Gtk.Separator`
`gtk.STOCK_*`	icon names — see Icon Names
`gtk.UIManager`, `gtk.ActionGroup`	`Gio.Menu` + `Gio.SimpleAction`
`gtk.Builder` (glade files)	`Gtk.Builder` (`.ui` files), or Blueprint

C.5 Dialogs stopped blocking

PyGTK dialogs ran a nested main loop:

```
response = dialog.run()
dialog.destroy()
if response == gtk.RESPONSE_OK:
    ...
```

GTK 4 dialogs are asynchronous. You pass a callback and return immediately:

```
dialog = Gtk.AlertDialog()
dialog.set_buttons(["Cancel", "Delete"])
dialog.choose(window, None, on_choice)

def on_choice(dialog, result, _data=None):
    try:
        index = dialog.choose_finish(result)
    except GLib.Error:
        return          # dismissed, not answered
```

This is the single most invasive change when porting. Code shaped like "ask a question in the middle of a function and carry on with the answer" has to be split in two at the question. There is no supported way to run a nested loop and wait.

The upside is that the nested-loop bugs go with it: no more reentrant signal handlers, no more dialogs that outlive the window that opened them, no more `destroy()` you forgot.

C.6 Signals

`connect()` works the same, and extra arguments are still passed through to the callback. Two differences:

Some `toggled`-style signals became property notifications. `Gtk.Switch` has no `toggled`; watch `notify::active` instead. The handler takes an extra `GParamSpec` argument you will ignore.

Input handling moved to **event controllers**. There is no `widget.connect("button-press-event", ...)` and no `gtk.EventBox` to wrap a widget that cannot receive events. Instead you attach a controller to any widget:

```
click = Gtk.GestureClick()
click.connect("pressed", on_pressed)    # (gesture, n_press, x, y)
label.add_controller(click)

keys = Gtk.EventControllerKey()
keys.connect("key-pressed", on_key)     # (controller, keyval, keycode, state)
window.add_controller(keys)

motion = Gtk.EventControllerMotion()
motion.connect("motion", on_motion)     # (controller, x, y)
widget.add_controller(motion)
```

C.7 Drawing

`expose-event` and `widget.window` are gone, along with the GDK drawing API.

```
area = Gtk.DrawingArea()
area.set_draw_func(on_draw)             # (area, cairo_context, width, height)
```

Cairo is still Cairo, so the body of an old `expose-event` handler usually transplants unchanged once you take the context from the argument list instead of calling `widget.window.cairo_create()`. See Drawing

with Cairo.

C.8 Threads

`gtk.gdk.threads_init()`, `threads_enter()` and `threads_leave()` are gone and have no replacement. GTK 4 is not thread-safe and never pretends to be: touch widgets only from the main thread.

To get a result from a worker thread back into the interface, hand it to the main loop:

```
GLib.idle_add(update_the_label, result)
```

`GLib.idle_add` and `GLib.timeout_add` are safe to call from any thread, and the callback runs on the main thread. A callback returning `GLib.SOURCE_CONTINUE` (`True`) is called again; returning `GLib.SOURCE_REMOVE` (`False`) stops it.

C.9 Things with no replacement

Some of what the previous edition covered has no modern equivalent, because the technology behind it was retired rather than replaced:

- **libglade** — use `Gtk.Builder` with `.ui` files; `gtk-builder-convert` is long gone.
- **Glade the designer** — no longer supports GTK 4. Write `.ui` files by hand, use Blueprint, or use Cambalache.
- **GConf** — use `GSettings`. See Desktop Integration.
- **Clutter** — folded into GTK; use GTK 4's own animation API. See Animation and Transitions.
- **gtkmozembed and the Internet Explorer control** — use WebKitGTK. See Embedding Web Content.
- **IronPython with Gtk#** — not a supported way to write GTK applications.
- **Empathy and Geoclue chapters** — Empathy is unmaintained; Geoclue is reached through portals now.

Appendix D

Icon Names

GTK 2 had **stock items**: `gtk.STOCK_SAVE` was an icon, a translated label and a keyboard accelerator in one constant. The whole system was deprecated in GTK 3.10 and removed in GTK 4. What replaced it is simpler and less helpful: an icon is a **name**, looked up in the icon theme, and the label and the accelerator are your problem.

```
Gtk.Button(icon_name="document-save-symbolic")
Gtk.Image.new_from_icon_name("dialog-warning-symbolic")
notification.set_icon(Gio.ThemedIcon.new("document-save-symbolic"))
```

The names come from the freedesktop icon naming specification, which every icon theme implements, so an icon named this way changes with the user's theme instead of being baked into your application.

D.0.1 The -symbolic suffix

Nearly every name below has a `-symbolic` variant, and in a GTK 4 application that is almost always the one you want. Symbolic icons are single-colour and are recoloured by GTK to match the text around them, so they stay legible in a dark theme, in a selected row, and on a coloured button. The full-colour variant (the same name without the suffix) is for large presentations of a thing — an application icon, a file type in a grid.

D.0.2 Checking a name exists

An icon name that the theme does not have renders as the "missing image" icon rather than raising, so a typo is silent. To check:

```
theme = Gtk.IconTheme.get_for_display(Gdk.Display.get_default())
print(theme.has_icon("document-save-symbolic"))
```

The `gtk4-icon-browser` tool, part of the GTK development packages, lists everything the theme has with a searchable index. It is the fastest way to find a name, and worth having open while you write.

Every name in this appendix was checked against the Adwaita theme shipped with GTK 4.22.

D.1 Files and documents

GTK 2 stock item	Icon name
`STOCK_NEW`	`document-new-symbolic`
`STOCK_OPEN`	`document-open-symbolic`

GTK 2 stock item	Icon name
STOCK_SAVE	document-save-symbolic
STOCK_SAVE_AS	document-save-as-symbolic
STOCK_REVERT_TO_SAVED	document-revert-symbolic
STOCK_EDIT	document-edit-symbolic
STOCK_PRINT	document-print-symbolic
STOCK_PRINT_PREVIEW	document-print-preview-symbolic
STOCK_PROPERTIES	document-properties-symbolic
STOCK_FILE	text-x-generic-symbolic
STOCK_DIRECTORY	folder-symbolic
—	folder-new-symbolic
STOCK_HOME	user-home-symbolic
STOCK_HARDDISK	drive-harddisk-symbolic
STOCK_CDROM	media-optical-symbolic
STOCK_NETWORK	network-server-symbolic
STOCK_PRINT_REPORT	printer-symbolic

D.2 Editing

GTK 2 stock item	Icon name
STOCK_CUT	edit-cut-symbolic
STOCK_COPY	edit-copy-symbolic
STOCK_PASTE	edit-paste-symbolic
STOCK_DELETE	edit-delete-symbolic
STOCK_UNDO	edit-undo-symbolic
STOCK_REDO	edit-redo-symbolic
STOCK_CLEAR	edit-clear-symbolic
STOCK_SELECT_ALL	edit-select-all-symbolic
STOCK_FIND	edit-find-symbolic
STOCK_FIND_AND_REPLACE	edit-find-replace-symbolic
STOCK_ADD	list-add-symbolic
STOCK_REMOVE	list-remove-symbolic
STOCK_SPELL_CHECK	tools-check-spelling-symbolic

D.3 Text formatting

GTK 2 stock item	Icon name
STOCK_BOLD	format-text-bold-symbolic
STOCK_ITALIC	format-text-italic-symbolic
STOCK_UNDERLINE	format-text-underline-symbolic
STOCK_JUSTIFY_LEFT	format-justify-left-symbolic
STOCK_JUSTIFY_CENTER	format-justify-center-symbolic
STOCK_JUSTIFY_RIGHT	format-justify-right-symbolic
STOCK_JUSTIFY_FILL	format-justify-fill-symbolic
STOCK_INDENT	format-indent-more-symbolic
STOCK_UNINDENT	format-indent-less-symbolic
STOCK_SELECT_COLOR	color-select-symbolic
STOCK_SELECT_FONT	font-select-symbolic

D.4 Navigation

GTK 2 stock item	Icon name
STOCK_GO_BACK	go-previous-symbolic
STOCK_GO_FORWARD	go-next-symbolic
STOCK_GO_UP	go-up-symbolic
STOCK_GO_DOWN	go-down-symbolic
STOCK_GOTO_FIRST	go-first-symbolic
STOCK_GOTO_LAST	go-last-symbolic
STOCK_JUMP_TO	go-jump-symbolic
STOCK_HOME	go-home-symbolic
STOCK_REFRESH	view-refresh-symbolic
STOCK_STOP	process-stop-symbolic

The go-previous and go-next icons **mirror themselves** in a right-to-left locale, which is why you should use them rather than go-left/go-right. Several names also have an explicit -rtl variant for cases where GTK cannot work out the correct direction on its own.

D.5 View and zoom

GTK 2 stock item	Icon name
STOCK_ZOOM_IN	zoom-in-symbolic
STOCK_ZOOM_OUT	zoom-out-symbolic
STOCK_ZOOM_100	zoom-original-symbolic
STOCK_ZOOM_FIT	zoom-fit-best-symbolic
STOCK_FULLSCREEN	view-fullscreen-symbolic
STOCK_LEAVE_FULLSCREEN	view-restore-symbolic
—	view-list-symbolic, view-grid-symbolic
—	sidebar-show-symbolic

D.6 Media

GTK 2 stock item	Icon name
STOCK_MEDIA_PLAY	media-playback-start-symbolic
STOCK_MEDIA_PAUSE	media-playback-pause-symbolic
STOCK_MEDIA_STOP	media-playback-stop-symbolic
STOCK_MEDIA_RECORD	media-record-symbolic
STOCK_MEDIA_PREVIOUS	media-skip-backward-symbolic
STOCK_MEDIA_NEXT	media-skip-forward-symbolic
STOCK_MEDIA_REWIND	media-seek-backward-symbolic
STOCK_MEDIA_FORWARD	media-seek-forward-symbolic
—	media-eject-symbolic

media-playback-start-symbolic mirrors in a right-to-left locale; the skip and seek icons do too.

D.7 Dialogs and status

GTK 2 stock item	Icon name
STOCK_DIALOG_INFO	dialog-information-symbolic
STOCK_DIALOG_WARNING	dialog-warning-symbolic
STOCK_DIALOG_ERROR	dialog-error-symbolic
STOCK_DIALOG_QUESTION	dialog-question-symbolic
STOCK_DIALOG_AUTHENTICATION	dialog-password-symbolic
STOCK_ABOUT	help-about-symbolic
STOCK_HELP	help-browser-symbolic
STOCK_PREFERENCES	preferences-system-symbolic
STOCK_QUIT	application-exit-symbolic
STOCK_EXECUTE	system-run-symbolic
STOCK_CLOSE	window-close-symbolic
STOCK_INFO	emblem-important-symbolic

D.8 Names with no stock ancestor

Worth knowing, because they are everywhere in modern applications:

open-menu-symbolic The hamburger button on a header bar.
view-more-symbolic A secondary menu, usually on a row rather than the header bar.
object-select-symbolic A tick. This is the "OK" icon — there is no emblem-ok.
content-loading-symbolic Something is in progress.

tab-new-symbolic, send-to-symbolic, starred-symbolic, non-starred-symbolic, find-location-symbolic, selection-mode-symbolic : The rest of the common vocabulary.

D.9 Stock items with no replacement

Some GTK 2 stock items have no icon equivalent, and that is deliberate rather than an oversight:

STOCK_OK, STOCK_CANCEL, STOCK_YES, STOCK_NO, STOCK_APPLY, STOCK_CLOSE (as a dialog button) : Use a **labelled button**. The guidance now is that a dialog's buttons should say what they do — "Delete", "Replace", "Discard" — rather than "OK", and an icon on a dialog button is noise. `Gtk.AlertDialog.set_buttons(["Cancel", "Delete"])` takes plain strings for exactly this reason.

STOCK_CONNECT, STOCK_DISCONNECT, STOCK_CONVERT, STOCK_INDEX, STOCK_ORIENTATION_*, STOCK_PAGE_SETUP, STOCK_DND : Too application-specific to be in a shared theme. Ship your own icon in your application's GResource and name it after your application id, or find a closer generic name in gtk4-icon-browser.

The label and accelerator halves of a stock item are gone too. Where GTK 2 gave you a translated "_Save" and Ctrl+S for free, you now write them yourself:

```
file_menu.append(_("_Save"), "app.save")
app.set_accels_for_action("app.save", ["<Control>s"])
```

See Menus and actions for the whole pattern, and Internationalization for translating the labels.

Further Reading

The previous edition's bibliography was a list of tutorials from 2006 to 2009. Almost every one of them is now either offline or describes a library that no longer exists, so it has been replaced with the places worth reading today.

The single most useful habit: when something in this book disagrees with the API reference, the API reference is right and this book is out of date.

Reference documentation

The API references https://docs.gtk.org/ — GTK 4, GDK, GSK, Gio, GLib, GObject and Pango, all generated from the same sources as the C documentation and kept current.

PyGObject https://pygobject.gnome.org/ — the binding itself: how introspection maps C onto Python, what `gi.require_version` does, and the awkward corners (`GObject.Property`, closures, threading).

libadwaita https://gnome.pages.gitlab.gnome.org/libadwaita/doc/ — every `Adw` widget used in Part I, with pictures.

The freedesktop specifications https://specifications.freedesktop.org/ — the desktop entry format, the icon naming specification, the base directory specification and the menu specification. Dry, short, and the authority when something will not appear in a launcher.

Guidance rather than API

The GNOME Human Interface Guidelines https://developer.gnome.org/hig/ — when to use a switch and when a check button, how a dialog should be worded, what a header bar is for. Worth reading once end to end; it explains a lot of why GTK 4 removed things.

The GNOME Developer Documentation https://developer.gnome.org/documentation/ — tutorials and "how do I" material above the level of the API reference, including the parts of application development this book covers in Part I.

Things to read the source of

GTK 4 Demo and GTK 4 Widget Factory Installed with the GTK development packages as `gtk4-demo` and `gtk4-widget-factory`. `gtk4-demo` shows its own source for every example, which makes it the fastest answer to "how is this widget actually used". `gtk4-demo` is C, but the API is the same API.

Adwaita Demo `adwaita-1-demo`, the same idea for libadwaita.

GNOME applications in Python Reading a finished application teaches the parts a book leaves out — how a real project lays out its Meson build, its resources and its state. Most GNOME applications are hosted at https://gitlab.gnome.org/GNOME/.

Tools worth having installed

`gtk4-icon-browser` Searchable index of every icon in the theme. See Icon Names.

The GTK Inspector `GTK_DEBUG=interactive python3 app.py`. A live widget tree, a CSS editor that applies as you type, and a way to see which widget is actually receiving your events.

`gdbus`, `busctl`, D-Spy For looking at the session bus before writing anything against it. See D-Bus.

`gst-launch-1.0`, `gst-inspect-1.0` For prototyping a pipeline in seconds rather than minutes. See Audio and Video with GStreamer.

`flatpak-builder` For building the manifest in Packaging and Distribution.

Where the previous edition went

This book began as *PyGTK Notebook*, covering PyGTK and GTK 2 between 2008 and 2012. That material is preserved in the git history of this repository, and the last PDF built from the original LyX source is at http://files.majorsilence.com/pygtkbook/pygtk-notebook-latest-0.13.pdf.

If you are porting something written against it, start with Migrating from PyGTK.