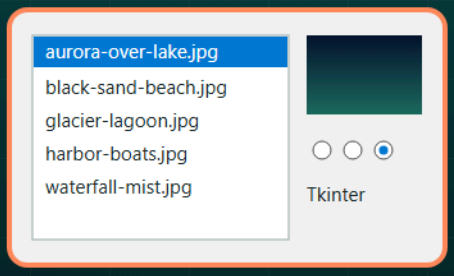



```
class StarRating(QWidget):
    ratingChanged = Signal(int)
stars.ratingChanged.connect(
    controller.set_rating)
```



A COMPLETE, PRACTICAL GUIDE

Desktop GUI Apps with Python

Build real desktop applications with Tkinter and PySide6 (Qt)

Events · Layouts · Signals & Slots · Model/View · Threads
SQLite · Styling · Accessibility · Testing · Packaging

TKINTER · PYSIDE6 · PILLOW · SQLITE · PYTEST

THENEXBOOKS

Desktop GUI Apps with Python

Build Real Desktop Applications with Tkinter and PySide6 (Qt)
From your first window to a tested, packaged photo organizer — built step by step

First Edition

THENEXBOOKS

Desktop GUI Apps with Python
Build Real Desktop Applications with Tkinter and PySide6 (Qt)

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. The code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python, Tk, Qt, PySide6, Pillow, and the other tools in this book change over time; APIs, defaults, and tool options can differ between versions and platforms. Always consult the official documentation for the versions you use. The licensing information in this book is a general overview, not legal advice; consult a qualified lawyer about your own situation.

Python is a registered trademark of the Python Software Foundation. Qt is a registered trademark of The Qt Company Ltd. and its subsidiaries. Windows is a trademark of the Microsoft group of companies; macOS is a trademark of Apple Inc.; Linux is a registered trademark of Linus Torvalds. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation, The Qt Company, or any other company mentioned.

Shutterbox and Shutterbox Lite, the applications built as the running example in this book, are fictional, as are Shutterbox Labs, the camera names, and all sample data. The sample "photos" are simple pictures generated with code; no real photographs are used.

Preface

Many of the most useful programs people run every day are not websites. They are desktop tools: a viewer for a folder of images, an editor for a configuration file, a small data-entry app for a team, a utility that talks to a device on the desk. They start instantly, work offline, open local files without uploading anything, and fit naturally into the operating system with menus, keyboard shortcuts, drag and drop, and the clipboard.

Python is an excellent language for building them. You already know it from scripts and services; with the right toolkit, the same code can grow a real window, a responsive interface, and a polished experience. What is usually missing is a guided path from "hello window" to an application you would be proud to hand to other people: one that does not freeze, keeps its data safe, remembers its settings, looks good in light and dark themes, works from the keyboard, has tests, and can be packaged for distribution.

This book is that path. We start with the ideas every GUI toolkit shares — events, callbacks, layouts — and learn them first with Tkinter, which ships with Python. Then we move to PySide6, the official Python binding for Qt, and use it to build a complete desktop application step by step, introducing each feature at the moment it solves a real problem.

Who This Book Is For

- **Python developers** who are comfortable with the language and want to build desktop applications and internal tools with a real user interface.
- **Engineers, scientists, and analysts** who have scripts that colleagues would use if only they had a window, a few buttons, and a progress bar.
- **Developers coming from web or command-line work** who want to understand event-driven programming, model/view design, and threading in a desktop context.
- **Teams maintaining Tkinter or Qt applications** who want sound architecture, tests, and a release process instead of one giant window class.

You do not need any previous GUI experience. You should know Python itself: functions, classes, modules, virtual environments, and installing packages.

What You Will Learn

- How event-driven programs work, why interfaces freeze, and the rules that keep them responsive.
- Tkinter and ttk: widgets, control variables, themes, the pack and grid layout managers, events and bindings, menus, dialogs, the Canvas, images with Pillow, and the Treeview — ending with a complete Tkinter app.

- PySide6 fundamentals: QApplication, QWidget, and QMainWindow, layouts and everyday widgets, signals and slots, actions, menus, toolbars, keyboard shortcuts, and dialogs with validation.
- Model/view programming with QAbstractTableModel, QTableView, QListView, proxy models for sorting and filtering, and delegates that draw custom cells.
- Real application plumbing: reading EXIF data and building thumbnails with Pillow, storing data in SQLite, running work in background threads with QThread and QThreadPool, and saving settings with QSettings.
- Polish: Qt Style Sheets and dark mode, Qt Designer and .ui files, custom-painted widgets, drag and drop, the clipboard, and accessibility and keyboard-first design.
- Professional practice: MVC and MVP architecture, testing GUIs with pytest and offscreen Qt, error handling and logging, packaging with PyInstaller and Nuitka, and the licensing questions every Qt developer should understand.

How This Book Is Organized

Part I — Foundations (Chapters 1–4) explains what desktop apps are good at, how event-driven programming works, how to set up your tools, and how we plan the running example. **Part II — Tkinter** (Chapters 5–9) teaches Python's built-in toolkit and finishes with Shutterbox Lite, a complete Tkinter photo browser. **Part III — PySide6 Fundamentals** (Chapters 10–14) introduces Qt: windows, layouts, widgets, signals and slots, actions and shortcuts, and dialogs. **Part IV — Building Shutterbox** (Chapters 15–19) builds the core of the full application: photos, EXIF, thumbnails, SQLite storage, model/view, thumbnail grids, and background threads. **Part V — Polish and User Experience** (Chapters 20–24) covers settings, styling and dark mode, Qt Designer, custom widgets and drag and drop, and accessibility. **Part VI — Architecture, Testing, and Shipping** (Chapters 25–28) covers architecture patterns, automated GUI testing, packaging and licensing, and a final case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Shutterbox

Throughout the book we build **Shutterbox**, a fictional desktop photo organizer. You point it at a folder, and it imports the photos, reads their EXIF data (the date taken and the camera), and builds thumbnails. You browse the library in a thumbnail grid or a sortable table, rate photos from zero to five stars, tag them, collect them into albums, and search by file name. Everything is stored in a local SQLite library, imports run in the background with a progress bar, the window remembers its size and layout, and the app offers light and dark themes and full keyboard control. In Part II we build a smaller cousin in Tkinter, **Shutterbox Lite**, which browses one folder and saves star ratings. A set of

twelve sample "photos" — simple pictures generated with code, complete with EXIF metadata — makes every result in the book reproducible, and the screenshots you see were captured from the real programs.

Versions and Conventions

The examples target **Python 3.12** or later, **Tkinter** with Tk 8.6, and **PySide6** (Qt 6), together with Pillow and pytest. Everything was run on Windows 11; where macOS or Linux behave differently in a way that matters, a Note says so. Qt and PySide6 evolve quickly, so when a feature arrived in a recent Qt 6 release, the text says so; if something looks different in your version, check the Qt for Python documentation. Dates and numbers in the examples are formatted explicitly (for example `2025-02-14 22:41` and `27.0 KB`) so the output is the same on every system.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version or platform differences, or a useful aside.

> **WARNING**
> A common mistake or something that can freeze your app, lose data, or cause security or licensing problems.

Code appears in `monospace`. Each listing is labeled with its file name, such as `shutterbox/ui/main_window.py`, or with **Terminal** for commands; program output follows in a separate block labeled **Output**. Keyboard shortcuts are written in bold, such as **Ctrl+I**. Long lines are wrapped to fit the page.

Contents

Preface

PART I — FOUNDATIONS

1 Desktop Apps with Python: Toolkits, Trade-offs, and Your First Window

2 Event-Driven Programming: Event Loops, Callbacks, and Responsive Interfaces

3 Setting Up: Python 3.12, Virtual Environments, PySide6, and Project Layout

4 Planning Shutterbox: Requirements, Screens, Data, and Architecture

PART II — TKINTER

5 Tkinter and ttk Essentials: Widgets, Variables, and Themes

6 Tkinter Layout: pack, grid, place, and Resizable Windows

7 Events, Bindings, Menus, and Dialogs in Tkinter

8 Canvas, Images, and Treeview: Working with Pillow in Tkinter

9 Project: Shutterbox Lite, a Complete Tkinter App

PART III — PYSIDE6 FUNDAMENTALS

10 Qt and PySide6: QApplication, QWidget, and QMainWindow

11 Layouts and Everyday Widgets

12 Signals and Slots in Depth

13 Actions, Menus, Toolbars, Status Bars, and Keyboard Shortcuts

14 Dialogs: Standard Dialogs, Custom Dialogs, and Input Validation

PART IV — BUILDING SHUTTERBOX

15 The Shutterbox Core: Photos, EXIF, and Thumbnails with Pillow

16 Persistence with SQLite: Schema, Queries, and the Photo Library

17 Model/View Programming: QAbstractTableModel and QTableView

18 Thumbnail Grids, Proxy Models, Delegates, and the Sidebar

19 Threads Without Freezing: QThread, QThreadPool, and Worker Signals

PART V — POLISH AND USER EXPERIENCE

20 Settings and State: QSettings, QStandardPaths, and Preferences

21 Styling with QSS, Palettes, Icons, and Dark Mode

22 Qt Designer and .ui Files

23 Custom Widgets, Drag and Drop, and the Clipboard

24 Accessibility and Keyboard-First Design

PART VI — ARCHITECTURE, TESTING, AND SHIPPING

25 App Architecture: MVC, MVP, and Thin Widgets

26 Testing GUI Apps with pytest and Offscreen Qt

27 Packaging and Distribution: PyInstaller, Nuitka, and Licensing

28 Case Study: Shutterbox 1.0 — Errors, Logging, and Release

APPENDICES

A Desktop GUI Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What desktop applications are good at and which Python toolkits exist, how event-driven programs work and why interfaces freeze, setting up Python, PySide6, and a clean project, and planning Shutterbox before writing its first window.

1 · Desktop Apps with Python
2 · Event-Driven Programming
3 · Setting Up
4 · Planning Shutterbox

CHAPTER 1 · PART I — FOUNDATIONS

Desktop Apps with Python: Toolkits, Trade-offs, and Your First Window

Before we write a single widget, we need to know what a desktop application is good at, which Python toolkits can build one, and what actually happens when a window appears on the screen. By the end of this chapter you will have opened your first window twice, once with Tkinter and once with PySide6, and you will know exactly where the rest of the book is heading.

What We Mean by a Desktop GUI App

A *desktop GUI application* is a program that runs on the user's own computer and talks to them through windows, menus, buttons, and other graphical controls instead of a terminal prompt. It is started like any other program on the machine: from the Start menu, a desktop shortcut, the Dock on macOS, or a launcher on Linux. It reads and writes files on the local disk, and it keeps running until the user closes it.

You already use dozens of these every week: a text editor, an image viewer, a music player, a settings panel, a database client. Many of the most useful ones are small. A team writes a little tool that checks a folder of invoices, a lab writes a viewer for instrument data, an operations group writes a front end for a script that everybody is afraid to run from the command line. These tools rarely make headlines, but they save hours every day, and Python is an excellent language for building them.

The "GUI" part brings a new way of thinking. A command-line script usually runs from top to bottom: read the arguments, do the work, print the result, exit. A GUI program instead sets up a window and then *waits*. It reacts to whatever the user does next, in whatever order they choose to do it. That shift, from a program that drives the conversation to a program that responds to events, is the single most important idea in this book. We meet it at the end of this chapter and study it properly in Chapter 2.

When a Desktop App Is the Right Choice

Today the default answer to "we need a tool with a user interface" is often "build a web app". Web apps are a fine choice for many problems, so it is worth being honest about where a desktop application really wins and where it does not.

Where desktop apps shine

- **Local files.** A desktop app can open a folder with ten thousand photos, read them directly from the disk, and never upload a byte. For large files, private data, or data that simply lives on one machine, this is a big advantage.
- **Working offline.** The app does not need a server or a network connection. It works on a train, in a lab without internet access, or on a locked-down corporate laptop.
- **Operating system integration.** Native menus, keyboard shortcuts, drag and drop from the file manager, the clipboard, system dialogs for opening and saving files, notifications, and a taskbar icon all come naturally.
- **Responsiveness.** There is no round trip to a server for every click. A well-written desktop app reacts immediately, even when it manages a lot of data.
- **Hardware and local tools.** Talking to a serial device, a scanner, a local database, or another program on the same computer is straightforward from a desktop process.
- **Privacy by default.** Data stays on the user's machine unless the app deliberately sends it somewhere.

Where another kind of program fits better

- **Shared, multi-user data.** If many people must see and edit the same records at the same time, a server is usually involved anyway, and a web front end is a natural fit.
- **Zero installation.** A web page needs only a browser. A desktop app has to be installed, and on some managed machines users cannot install software at all.
- **Very frequent updates.** Updating a website updates every user at once. Desktop apps need an update story (Chapter 27 discusses it).
- **Mobile reach.** The toolkits in this book target Windows, macOS, and Linux desktops. Phones and tablets are a different world.
- **Automation.** If the main users are other programs, scheduled jobs, or power users who chain commands together, a command-line tool is simpler and more scriptable than any window.

The table below summarizes the trade-offs. It is a rule of thumb, not a law; plenty of successful products combine two of these forms, for example a desktop app that syncs with a web service, or a CLI that also offers a small GUI.

Need	Desktop GUI	Web app	CLI
Work with large local files	Excellent	Awkward (upload)	Good
Work offline	Yes	Limited	Yes
Many users, shared data	Needs a server	Natural	Needs a server
Nothing to install	No	Yes	No
Friendly for non-technical users	Yes	Yes	Rarely

Easy to automate and script	Rarely	Via an API	Yes

> **NOTE**
>
> This book is only about desktop GUIs. We assume you already know Python itself (functions, classes, modules, virtual environments, and pip). We do not cover command-line argument parsing or web frameworks, except for the few lines of `argparse` that the finished Shutterbox app uses for its options.

The Python GUI Landscape

Python does not have one official GUI toolkit. It has one toolkit in the standard library and a healthy set of third-party options, each with its own design and its own sweet spot. Here is a neutral tour; the order means nothing.

Toolkit	What it is	Good to know
Tkinter (with ttk)	Python's standard-library binding to Tcl/Tk	Ships with the python.org installers; simple and small
PySide6	Qt for Python, the official binding maintained by The Qt Company	Large, mature widget set; LGPLv3, GPLv3, or commercial
PyQt6	A third-party Qt binding by Riverbank Computing	Near-identical API to PySide6; GPLv3 or commercial
wxPython	A binding to the wxWidgets C++ library	Uses native controls on each platform
Kivy	A toolkit with its own rendering, aimed at touch	Also targets mobile devices
Toga (BeeWare)	A native-widget toolkit from the BeeWare project	Pairs with the Briefcase packaging tool
Dear PyGui	A GPU-accelerated, immediate-mode-style toolkit	Popular for tools, dashboards, and plots
Flet	Python apps rendered with Flutter	Desktop, web, and mobile from one codebase
pywebview and similar	A native window that hosts web technology	The UI is HTML, CSS, and JavaScript

All of these can produce useful applications. They differ in how the interface is drawn (native controls, the toolkit's own painting, or a web engine), in how much they include beyond widgets, in their licenses, and in their communities. When you choose one for a real project, read its current documentation and license yourself rather than relying on anyone's ranking, including ours.

Why this book uses Tkinter first, then PySide6

We use two toolkits on purpose, and in a deliberate order.

Tkinter is already on your machine if you installed Python from python.org on Windows or macOS. There is nothing to install, the API is small, and you can see a window after five lines. That makes it the ideal place to learn the ideas every toolkit shares: widgets, layout managers, callbacks, the event loop, and the rule that slow work must never block the interface. In Part II we build Shutterbox Lite, a complete little photo browser, with nothing but Tkinter and Pillow.

PySide6 is the official Python binding for Qt, a large C++ framework that has been used for desktop software for a long time. Qt brings things Tkinter does not offer out of the box: a model/view architecture that scales to thousands of rows, rich widgets such as dockable panels and tree views, style sheets, a resource system, a visual designer, strong accessibility support, and first-class threading tools. Parts III to VI use PySide6 to build the full Shutterbox application step by step.

The concepts transfer in both directions. Once you understand an event loop in Tkinter, Qt's event loop holds no surprises. Once you have seen Qt's model/view design, you will structure even small Tkinter apps more cleanly. And if a future project calls for wxPython, Toga, or something else, you will recognize the same patterns under different names.

> **TIP**
>
> If you later prefer PyQt6 to PySide6, most of the Qt code in this book carries over with small changes. The most visible difference is naming: PyQt6 declares custom signals with `pyqtSignal` where PySide6 uses `Signal`. Check the license difference first, though; it matters more than the API.

A first word about licenses

Licenses rarely come up in a "hello window" chapter, but with GUI toolkits they deserve an early mention because they affect how you may distribute your app. Tkinter is covered by Python's own permissive license and Tcl/Tk's BSD-style license. PySide6 is available under the LGPLv3, the GPLv3, or a commercial license from The Qt Company. Under the LGPL you can usually ship closed-source applications if you comply with its terms, which concern things like linking, letting users replace the Qt libraries, and providing notices and source for the LGPL parts. PyQt6 is available under the GPLv3 or a commercial license. Chapter 27 covers these obligations in more detail.

> **WARNING**
>
> Nothing in this book is legal advice. Licensing questions depend on how you build, bundle, and distribute your app. If you plan to sell or widely distribute a closed-source application, read the actual license texts and consult a lawyer for your situation before you commit to a toolkit.

Your First Window with Tkinter

Enough background. Let us put a window on the screen. Create a project folder (Chapter 3 calls it `shutterbox-project` and fills in the rest; for now any empty folder works), make a folder called `examples` inside it, and save this file there. Then open a terminal in the project folder, not in `examples`, and run the file as a module.

examples/ch01_hello_tk.py

```
import tkinter as tk

root = tk.Tk()
root.title("Hello, Tkinter")
root.geometry("360x120")
label = tk.Label(root, text="Hello from Tkinter!", font=("Segoe UI", 14))
label.pack(expand=True)
root.after(1000, root.destroy)  # close automatically after one second
root.mainloop()
print("mainloop() returned - the window is gone")
```

Terminal

```
py -3.12 -m examples.ch01_hello_tk
```

Output

```
mainloop() returned - the window is gone
```

A small window titled "Hello, Tkinter" appears for one second, shows the greeting, and closes; then the message is printed in the terminal. Line by line:

- `tk.Tk()` creates the *root window*, the application's main window. It also starts the Tcl interpreter that Tkinter drives behind the scenes. A Tkinter program normally has exactly one `Tk` object.
- `title()` and `geometry()` set the window title and its initial size in pixels, as a `"WIDTHxHEIGHT"` string.
- `tk.Label(root, ...)` creates a widget whose *parent* is the root window. Every widget except the root has a parent, and that parent-child tree is how both toolkits organize a window.
- `label.pack(expand=True)` hands the label to a *geometry manager*. Creating a widget does not show it; you must tell a manager where to put it. Chapter 6 covers `pack`, `grid`, and `place` in depth.
- `root.after(1000, root.destroy)` asks Tkinter to call `root.destroy` one second from now. We added it so the example closes itself; in a real app the user closes the window.
- `root.mainloop()` starts the *event loop*. This is where the program spends almost all of its life. It returns only when the root window is destroyed.

The last line is the interesting one. The `print` call comes after `mainloop()` in the file, yet it does not run until the window has closed. While the window is open, Python is busy inside `mainloop()`, waiting for things to happen and reacting to them.

> **NOTE**
> The code is the same on Windows, macOS, and Linux; only the window frame and the default font look different. On some Linux distributions Tkinter is packaged separately from Python (for example as `python3-tk`), so `import tkinter` can fail until you install that package. Chapter 3 has the setup details.

Your First Window with PySide6

Now the same idea in Qt. If PySide6 is not installed yet, Chapter 3 shows how to set up a virtual environment and install it with pip; you can read along now and run the file later.

examples/ch01_hello_qt.py

```
import sys

from PySide6.QtCore import Qt, QTimer
from PySide6.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)
label = QLabel("Hello from PySide6!")
label.setAlignment(Qt.AlignmentFlag.AlignCenter)
label.setWindowTitle("Hello, Qt")
label.resize(360, 120)
label.show()
QTimer.singleShot(1000, label.close)  # close automatically after one second
exit_code = app.exec()
print(f"exec() returned {exit_code} - the window is gone")
sys.exit(exit_code)
```

Terminal

```
py -3.12 -m examples.ch01_hello_qt
```

Output

```
exec() returned 0 - the window is gone
```

The structure is strikingly similar to the Tkinter version, with a few Qt-specific details:

- `QApplication(sys.argv)` creates the application object. Every Qt widget program needs exactly one, and it must exist before any widget is created. Passing `sys.argv` lets Qt read its own standard command-line options.

- A `QLabel` with no parent becomes a top-level window by itself. In Qt, any widget without a parent can be a window; in Chapter 10 we use `QMainWindow`, the class designed for an application's main window.
- `Qt.AlignmentFlag.AlignCenter` is written fully qualified. Qt groups its constants into enum classes, and this book always spells out the full name so you can see where each value comes from.
- `show()` makes the window visible. Unlike Tkinter, a new Qt window stays hidden until you show it.
- `QTimer.singleShot(1000, label.close)` is Qt's version of `after()`: call this function once, one second from now.
- `app.exec()` starts Qt's event loop and returns an exit code when the application quits. By default a Qt application quits when its last window closes, which is what happens here. Older code sometimes calls `exec_()`; with PySide6 you write `exec()`.

> **TIP**
>
> Many tutorials write `sys.exit(app.exec())` in one line. We split it here so you can see the return value. Passing it to `sys.exit` matters for scripts and installers that check whether a program ended successfully.

What mainloop() and exec() Really Do

Both programs end in a call that does not return until the window closes. Those calls are the heart of every GUI program, so here is a first look at what happens inside them. Chapter 2 builds a tiny event loop in plain Python to make this concrete.

The operating system sends a steady stream of messages to every window: the mouse moved, a key was pressed, the window was uncovered and must be repainted, the user clicked the close button, a timer expired. The toolkit's event loop is, at its core, a `while` loop that does three things over and over:

1. Wait until at least one event is available (sleeping while nothing happens, so an idle app uses almost no CPU).
2. Take the next event from the queue.
3. Dispatch it: find the widget it belongs to and call the code that handles it, including *your* code if you connected a function to that event.

Our `after()` and `singleShot()` calls simply put a timer into that system. When the timer expires, the loop calls the function we gave it, the window is destroyed or closed, and the loop sees that there is nothing left to run and returns.

This has a crucial consequence that you should remember from day one: **while your code runs, the loop is not running.** If a button handler takes five seconds, the window cannot repaint, cannot respond to clicks, and may be marked "Not Responding" by the operating system. Chapter 2 demonstrates this with real timings and shows the fixes.

A Window That Does Something

A label that disappears after a second proves the toolkit works, but it does not react to anything. Let us add the smallest possible interaction: a button that counts how often it was clicked. This time we write each program as a class, the shape most GUI code takes, because the widgets and the state they display belong together.

examples/ch01_counter_tk.py

```
import tkinter as tk
from tkinter import ttk


class CounterApp:
    def __init__(self, root: tk.Tk) -> None:
        self.count = 0
        root.title("Click Counter")
        self.label = ttk.Label(root, text="Clicked 0 times")
        self.label.pack(padx=20, pady=(20, 8))
        self.button = ttk.Button(root, text="Click me", command=self.on_click)
        self.button.pack(padx=20, pady=(0, 20))

    def on_click(self) -> None:
        self.count += 1
        self.label.configure(text=f"Clicked {self.count} times")
        print(f"on_click: count is now {self.count}")


if __name__ == "__main__":
    root = tk.Tk()
    app = CounterApp(root)
    # Our test copy presses the button three times, then closes the window.
    for delay in (300, 400, 500):
        root.after(delay, app.button.invoke)
    root.after(800, root.destroy)
    root.mainloop()
```

Output

```
on_click: count is now 1
on_click: count is now 2
on_click: count is now 3
```

The important part is `command=self.on_click`. We do not call `on_click` ourselves; we hand the method to the button, and the event loop calls it each time the button is clicked. A function used this way is called a *callback* or *event handler*. Notice that we pass `self.on_click` without parentheses: we give the button the function, not the result of calling it.

We also switched from `tk.Label` to `ttk.Label` and `ttk.Button`. The `ttk` module contains the *themed* widgets, which use the platform's native look (on Windows the default theme is called `vista`). Chapter 5 explains the difference between the classic and themed widgets; from now on we prefer ttk.

To make the example verifiable, our test copy uses `button.invoke()`, which does exactly what a click does, three times, and then closes the window. When you run it yourself, you can delete those four lines and click the button with the mouse instead.

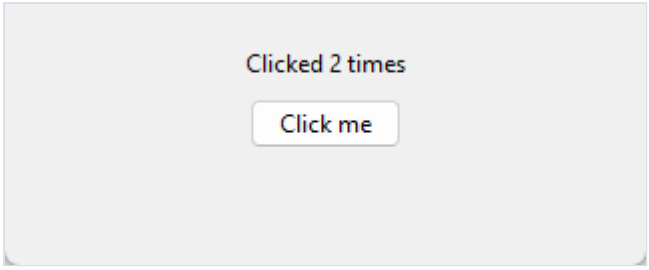


Figure 1.1 — The Tkinter counter after two clicks (client area only; the title bar is not captured).

Figure 1.1 shows the window after two clicks. The label and the button are centered because `pack()` places widgets one below the other in the middle of the available width.

Here is the same program in PySide6.

examples/ch01_counter_qt.py

```
import sys

from PySide6.QtCore import QTimer
from PySide6.QtWidgets import QApplication, QLabel, QPushButton, QVBoxLayout, QWidget


class CounterWindow(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.count = 0
        self.setWindowTitle("Click Counter")
        self.label = QLabel("Clicked 0 times")
        self.button = QPushButton("Click me")
        self.button.clicked.connect(self.on_click)
        layout = QVBoxLayout(self)
        layout.addWidget(self.label)
        layout.addWidget(self.button)

    def on_click(self) -> None:
        self.count += 1
        self.label.setText(f"Clicked {self.count} times")
        print(f"on_click: count is now {self.count}")


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = CounterWindow()
    window.show()
    # Our test copy presses the button three times, then closes the window.
    for delay in (300, 400, 500):
        QTimer.singleShot(delay, window.button.click)
    QTimer.singleShot(800, window.close)
    sys.exit(app.exec())
```

Output

```
on_click: count is now 1
on_click: count is now 2
on_click: count is now 3
```

Qt uses *signals and slots* instead of a `command=` option. The button has a `clicked` signal, and `self.button.clicked.connect(self.on_click)` connects it to our method. When the user clicks, the button *emits* the signal and Qt calls every connected function. One signal can be connected to many functions, and the button does not need to know anything about who is listening; Chapter 12 explores this design in depth.

Here the window class inherits from `QWidget` instead of wrapping the root window as the Tkinter version does. Subclassing the window is the common style in Qt. The `QVBoxLayout(self)` line creates a vertical layout and installs it on the window; the widgets added to it become the window's children automatically, so Qt deletes them when the window is deleted.

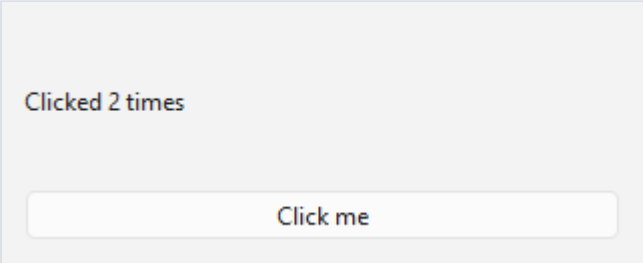


Figure 1.2 — The PySide6 counter after two clicks, captured without ever appearing on screen.

Figure 1.2 shows the Qt window at the same size as Figure 1.1. The layout stretches the button across the full width and spreads the widgets vertically, while `pack()` kept the Tkinter widgets at their natural size. Neither is right or wrong; each toolkit has defaults, and both let you change them. Chapters 6 and 11 cover layouts in depth.

The two programs side by side

Concept	Tkinter	PySide6
Application object	`tk.Tk()` (also the main window)	`QApplication(sys.argv)`
A window	The root, or a `Toplevel`	Any widget without a parent
Placing widgets	`pack` , `grid` , `place`	Layout objects such as `QVBoxLayout`
Reacting to a click	`command=` callback	`clicked.connect(...)`
Run later	`root.after(ms, fn)`	`QTimer.singleShot(ms, fn)`
Event loop	`root.mainloop()`	`app.exec()`

Notice the naming styles. Tkinter follows Python's lower-case conventions. Qt is a C++ library whose names are camelCase (`setWindowTitle`, `addWidget`), and PySide6 keeps those names. In our own code we follow PEP 8 and write methods in snake_case, such as `on_click`, so you can always tell at a glance which names come from Qt and which are ours.

Meet Shutterbox

Small examples teach individual ideas, but real understanding comes from building something complete. Throughout this book we build **Shutterbox**, a fictional desktop photo organizer. It is the kind of tool desktop apps are made for: it works with files on your own disk, it must stay responsive while it reads them, and it benefits from menus, keyboard shortcuts, and drag and drop.

Here is what the finished application does:

- **Import** a folder of photos. Shutterbox scans it in the background, reads each photo's EXIF data (the date taken and the camera) with Pillow, and builds thumbnails, while a progress bar keeps you informed.
- **Browse** the library as a thumbnail grid or as a sortable table, with a detail panel that shows a larger preview and the photo's information.
- **Rate** photos from zero to five stars with the mouse or the keys **0** to **5**.
- **Tag** photos (landscape, night, flowers) and browse by tag.
- Collect photos into **albums**, by command or by dragging them onto an album in the sidebar.
- **Search** by file name as you type, and see Favorites, the photos rated four stars or more, in one click.
- Keep everything in a local **SQLite** library, remember the window layout and preferences, and offer **light and dark themes**.

In Part II we first build **Shutterbox Lite** with Tkinter: a single-file program that browses one folder, shows a preview, and saves star ratings to a small JSON file. It is a complete, useful little app, and it lets us practice every Tkinter fundamental on a real problem. Parts III to VI then build the full PySide6 version, one feature per chapter, until we reach the tested and packaged Shutterbox 1.0 in Chapter 28.

To make every result reproducible, the book comes with twelve sample "photos": simple pictures drawn with code, complete with fictional EXIF data, in three folders called `iceland-2025`, `garden`, and `city-walk`. Whenever you see output in this book, such as a photo called `aurora-over-lake.jpg` taken on 2025-02-14 at 22:41, you will get the same output on your machine. Chapter 3 shows how to generate the photos and run the finished app; Chapter 4 plans the application in detail.

> **NOTE**
>
> Shutterbox, Shutterbox Labs, the camera names, and all sample data are fictional. The sample photos are generated pictures, so you can experiment freely without touching your own photo collection.