Python and Qt

The Best Parts.

Michael Herrmann

With a foreword by the creator of PyQt

Contents

Foreword

I'm very happy that Michael wrote this book. There are other resources that teach you how to write a GUI. But not all of them are up-to-date. Even fewer are as well-written. And none get you to a deployable application as quickly.

Michael's goal is to save you time learning. He does this by answering the questions you will actually encounter. And by avoiding unnecessary detail. This makes the book an engaging read with many valuable insights.

In this sense, I feel that this book is an important contribution. It is a fantastic resource, especially if you are just starting out. But also practitioners will benefit from Michael's extensive experience. In short: I think you will enjoy it!

Phil Thompson,
creator of PyQt

Chapter 1

Welcome!

1.1   About this book

This short book distills years of experience. It exposes implicit knowledge that normally takes practitioners of Python and Qt a long time to learn. By reading these pages, you will quickly be able to create better GUI applications.
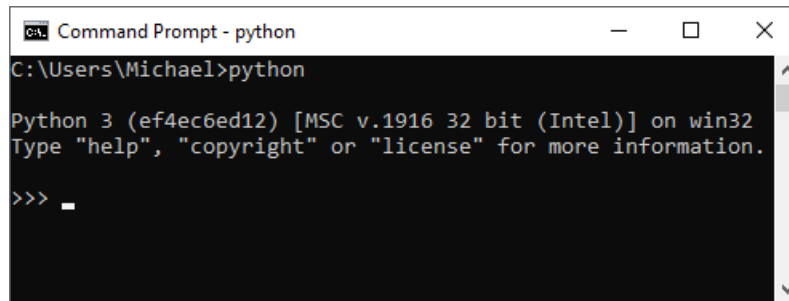
Because this book focuses on bringing you the most valuable information in the least amount of time, it is necessarily incomplete. It does not include an introduction to Python. Nor is it a reference manual for Qt. Instead, you will learn the important *principles*. And where you can find any necessary details.

That's not to say that this book is light on examples. On the contrary, every section is motivated by one or more real-world use cases. You can find a copy of their source code at https://github.com/pyqt/examples. You are cordially invited to play with and use them as a basis for your own projects.

1.2   Prerequisites

The examples in this book are based on Python 3. To run them, you need to be able to start its interpreter, as shown in Figure 1.1.

Furthermore, you will need some basic knowledge about working on the command line, such as how to change directories and what the PATH environment variable does.

Figure 1.1: Python started from the command line on Windows

You do not absolutely need to know Python. By tweaking the examples presented here, you can also learn how things work without deeply knowing the language. However: It is not amiss to say that you will find it easier to follow the examples if you know about object-oriented programming.

Chapter 2

Background

2.1 About Qt

Qt is a GUI framework with a rich history: It was initially created in 1991 by the Norwegian company Trolltech. In 2008, it was acquired by Nokia who used it for the Symbian mobile operating system. Symbian succumbed to Android, so Nokia switched to Microsoft Windows Phone and sold Qt to a company called Digia. These days, Qt is owned by *The Qt Company*, a former Digia subsidiary.

A very common misconception about Qt is the pronunciation of its name. Most people pronounce it as "cu-tee". But the official, correct way is just "cute"!

Today, Qt has many uses. It is the basis of the KDE Linux desktop environment and popular applications such as VLC. But it is also widespread in industry: Many factories contain machines with user interfaces based on Qt. Tesla and other manufacturers use it for their in-car displays. And even consumer devices such as TVs and coffee machines have Qt GUIs.

Most of Qt is open source under the LGPL. This lets typical Python projects use it for free. However, the license does have some formal requirements. For information on how to comply with them, please see Appendix A.

2.2 Using Qt from Python

Qt is written in C++. This has great performance but is more difficult to use than other programming languages. Many developers therefore prefer to work

with more high-level languages. Among those, Python has set itself apart as the clear winner, especially when working with Qt. The great benefit of combining Qt with Python is that you can build applications very quickly while not sacrificing much of the speed of C++.

Using Qt from Python requires so-called *language bindings*. These are Python libraries that allow you to interface with Qt's C++ API. We'll use the most prominent Python binding in this book, PyQt. For an alternative, see Appendix B.

2.2.1   PyQt

PyQt was first released in 1998 and has been in continuous development ever since. It is maintained by Riverbank Computing Limited, a company independent of Qt. By now, PyQt has been the most stable and widely used Python binding for Qt for decades.

PyQt is open source under the GPL, which unlike the *L*GPL does not permit dynamic linking into proprietary applications. For projects where this is too restrictive, Riverbank Computing offer a commercial license. If you use PyQt for a proprietary application, then you will typically have to buy a license for it. At the time of this writing, a license costs $550 for one developer.

2.2.2   Naming conventions

PyQt has virtually the same API as Qt: Classes such as `QApplication` etc. have the same names in Python. Similarly for constants and method names: `QCheckBox::isChecked()` is called `QCheckbox.isChecked()` in Python.

Some people consider this "unpythonic". In Python, the convention would be to use underscores in everything but class names: `is_checked()`. Developers coming to Qt from a Python background sometimes find this difference foreign, especially when the two writing styles are mixed in one project.

But mirroring Qt's naming convention has enormous benefits. It allows Python developers to use Qt's excellent documentation. When searching for solutions to a problem online, you don't have to translate snake_case to CamelCase. You can use any answers you find almost literally. Finally, you can seamlessly mix Python and C++ if this becomes necessary for performance. All these points easily trump the comparatively superficial effects of a naming convention.

A natural question to ask at this point is which convention you should adopt in your own projects. The answer is surprisingly easy: By using Qt, you will have CamelCase in your code. But virtually all Python libraries use snake_case. So you will have that as well. The best approach I have found is to use snake_case, except in cases where you don't have a choice such as when overwriting a method from Qt. This has the added benefit of letting you see immediately whether a method is from your own code or imposed by Qt.

2.3   Widgets vs. Qt Quick / QML

Qt offers two broad technologies for creating user interfaces: The first, referred to as *Widgets*, is the old core of Qt. It renders desktop GUIs typical for operating systems such as Windows or macOS. A somewhat more recent technology is *Qt Quick*: It offers highly customizable graphics, fluid animations and is more targeted at mobile devices and touch displays.

Qt Quick uses a markup language called *QML* for laying out UIs. It is functionally similar to HTML, CSS and JavaScript. In fact, it supports JavaScript as a scripting language. Its overall syntax is similar to JSON. Qt Quick applications often use QML for the UI and C++ or Python for the business logic.

Widgets on the other hand are often laid out directly in code. That is, there are C++ or Python statements that say "draw a button here". This can be avoided by using Qt Designer (see Section 3.6), a visual editor that outputs special XML files describing a UI. But it lacks the dynamic features of JavaScript.

In general, you should use Widgets if you want a pretty standard desktop GUI. Otherwise, if you want highly customizable graphics and / or support for mobile devices and touch displays, use Qt Quick.

Because Qt Quick already supports JavaScript, a dynamic scripting language, combining it with Python yields fewer benefits than doing so for Widgets. For this reason, this book is mostly about Widgets.

2.4   Reference materials

As briefly mentioned in Section 2.2.2, Qt's documentation is fantastic. You should use it more than any other resource. Don't be put off by the fact that

it is about C++. Because the Python APIs are so similar to Qt's own, you will easily be able to make sense of everything.

As your usage of Qt becomes more advanced, you may encounter the need to read Qt's C++ source code. This too is nothing to be afraid of. Simply download the source archive called "single" from Qt's home page. Unpack and open the entire folder (!) with an editor that lets you navigate to files by name. (A great choice is Sublime Text with its shortcut `Ctrl/Cmd` + `P`.) Then, you can find the implementation of any class such as `QWidget` by opening `qwidget.cpp`.

Chapter 3

A whirlwind tour of Qt with Python

This chapter gives you a quick tour of Qt with Python, so you get a feeling for how to use the two, and what they can achieve. Later chapters then show more detailed examples.

3.1   Installing PyQt

The cleanest way to install PyQt is via a virtual environment. A virtual environment is simply a folder that contains all libraries for a project. This is unlike a system-wide installation of those libraries, which would affect your other projects as well.

To create a virtual environment in the current directory, execute the following command:

```
python -m venv venv
```

This creates the `venv/` folder. To activate the virtual environment on Windows, run:
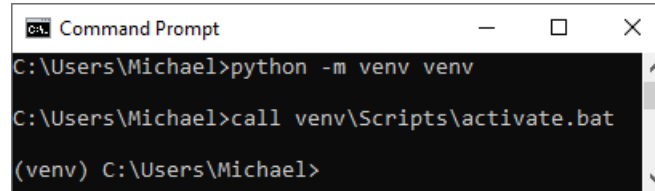
```
call venv/scripts/activate.bat
```

On Mac and Linux, use:

```
source venv/bin/activate
```

You can see that the virtual environment is active by the `(venv)` prefix in your

shell. This is shown in Figure 3.1.

Figure 3.1: Creating and activating a virtual environment
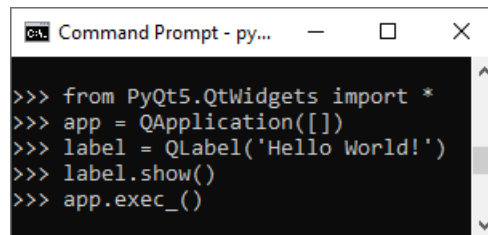
To then install PyQt, issue the following command:

```
pip install PyQt6
```

That's it! You've successfully set up PyQt.

3.2   A first GUI

Time to write our very first GUI app! With the virtual environment still active, start Python. We will execute the commands shown in Example 1.

Example 1: Commands for a Hello World app with PyQt

First, we tell Python to load PyQt via the import statement:

```
from PyQt6.QtWidgets import *
```

Next, we create a `QApplication` with the command:

```
app = QApplication([])
```

This is a requirement of Qt: Every GUI app must have exactly one instance of `QApplication`. Many parts of Qt don't work until you have executed the above line. You will therefore need it in virtually every (Py)Qt app you write.

The brackets `[]` represent the command line arguments passed to the application. Because our app doesn't use any parameters, we leave them empty.

Now, to actually see something, we create a simple label:

```
label = QLabel('Hello World!')
```

Then, we tell Qt to show the label on the screen:

```
label.show()
```

Depending on your operating system, this already opens a tiny little window as shown in Figure 3.2.
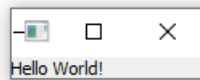
Figure 3.2: A very simple Hello World app on Windows

The last step is to hand control over to Qt and ask it to "run the application until the user closes it". This is done via the command:

```
app.exec()
```

If all this worked as expected then well done! You've just built your first GUI app with Python and Qt.

3.3   Widgets

Everything you see in a (Py)Qt app is a *widget*: Buttons, labels, windows, dialogs, progress bars etc. Like HTML elements, widgets are often nested. For example, a window can contain a button, which in turn contains a label.

Example 2 shows the most common Qt widgets. Top-to-bottom, left-to-right, they are:

- QLabel
- QComboBox
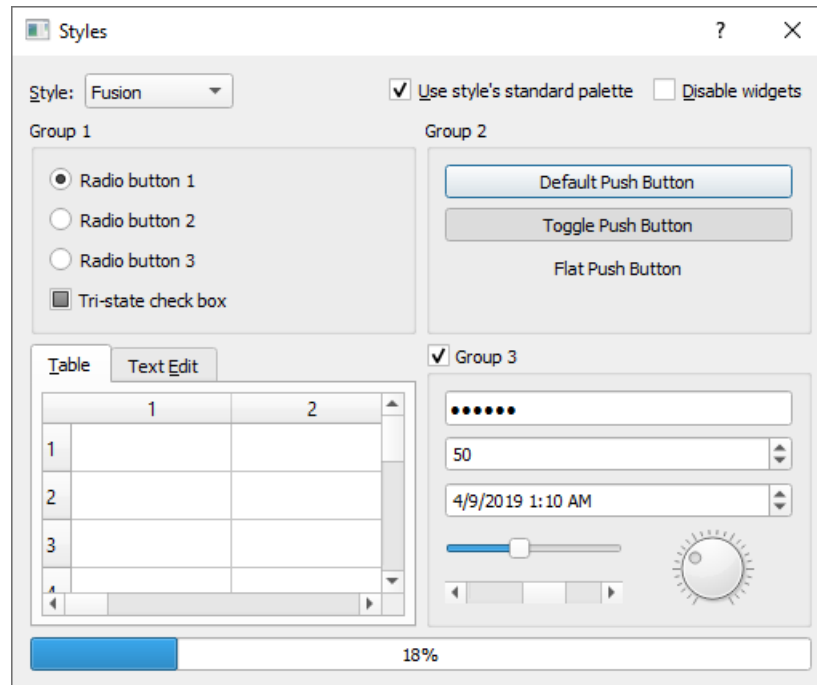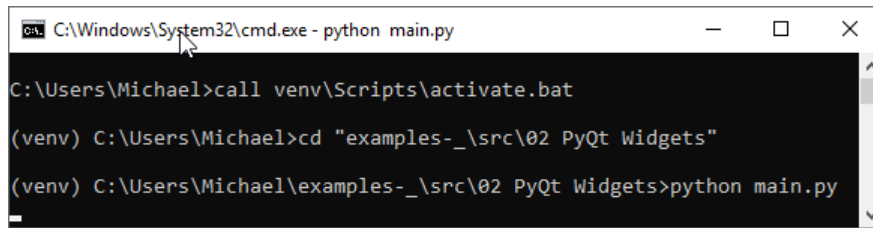
Example 2: The most important Qt widgets

- QCheckBox
- QRadioButton
- QPushButton
- QLineEdit
- QTableWidget
- QSlider
- QProgressBar

Note how the screenshot is labeled as an "Example". This indicates that you can find its source code in the GitHub repository mentioned in Section 1.1.

To run the example yourself, download the repository and use `cd` to change into the sample's subdirectory. Then invoke `python main.py` while the virtual environment is active. These steps are shown in Figure 3.3.

Figure 3.3: Commands for executing the Widgets example

3.4 Layouts

Like the example above, your GUI will most likely consist of multiple widgets. In this case, you need to tell Qt how to position them. For instance, you can use `QVBoxLayout` to stack widgets vertically. This is shown in Example 3.
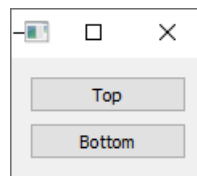
Example 3: Two buttons laid out vertically using a QVBoxLayout

The code for this example is:

```python
from PyQt6.QtWidgets import *
app = QApplication([])
window = QWidget()
layout = QVBoxLayout()
layout.addWidget(QPushButton("Top"))
layout.addWidget(QPushButton("Bottom"))
window.setLayout(layout)
window.show()
app.exec()
```

As before, we instantiate a `QApplication`. Then, we create a window. We use the most basic type `QWidget` for it because it merely acts as a container and we don't want it to have any special behavior. Next, we create the layout and add two `QPushButtons` to it. Finally, we tell the window to use this layout (and thus its contents). As in our first application, we end with calls to `.show()` and

```
app.exec().
```

There are many other kinds of layouts (eg. `QHBoxLayout` to lay out items horizontally). For a quick overview, please see Qt's documentation.

3.5   Signals and Slots

Qt uses a mechanism called *signals* to let you react to events such as the user clicking a button. Example 4 illustrates this. It contains a button that, when clicked, shows a message box.
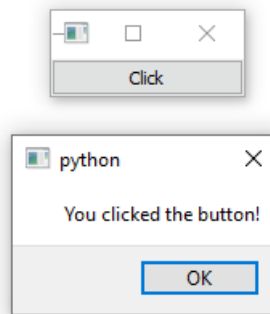
Example 4: Responding to user input with signals

The code for this example is:

```python
from PyQt6.QtWidgets import *

app = QApplication([])
button = QPushButton("Click")

def on_button_clicked():
    alert = QMessageBox()
    alert.setText("You clicked the button!")
    alert.exec()

button.clicked.connect(on_button_clicked)
```

```
button.show()
app.exec()
```

(If you've never used Python before: Indentation is important. Put four spaces before each line in on_button_clicked and follow it with an empty line.)

The crucial step is highlighted: button.clicked is a signal, .connect(...) lets us install a so-called *slot* on it. This is simply a function that gets called when the signal occurs. In the above example, our slot shows a message box.

The term slot is important when using Qt from C++, because slots must be declared in a special way in C++. In Python however, any function can be a slot – we saw this above. For this reason, the distinction between slots and "normal" functions has little relevance for us.

Signals are ubiquitous in Qt. And of course, you can also define your own. An example of this can be found in a later section, on pages 75ff.

3.6   Creating GUIs visually with Qt Designer

Instead of writing code to lay out your widgets, you can also use a graphical tool: Qt Designer gives you a simple drag-and-drop interface for building GUIs. Figure 3.4 shows it in a screenshot.

Qt Designer produces .ui files. This is a special XML-based format that stores widgets as a tree. You can either load these files at runtime, or have them translated into programming languages such as C++ or Python.

To load a .ui file into Python, you can use PyQt's loadUiType(...) function as follows:

```python
from PyQt6 import uic
from PyQt6.QtWidgets import QApplication

Form, Window = uic.loadUiType("dialog.ui")

app = QApplication([])
window = Window()
form = Form()
```
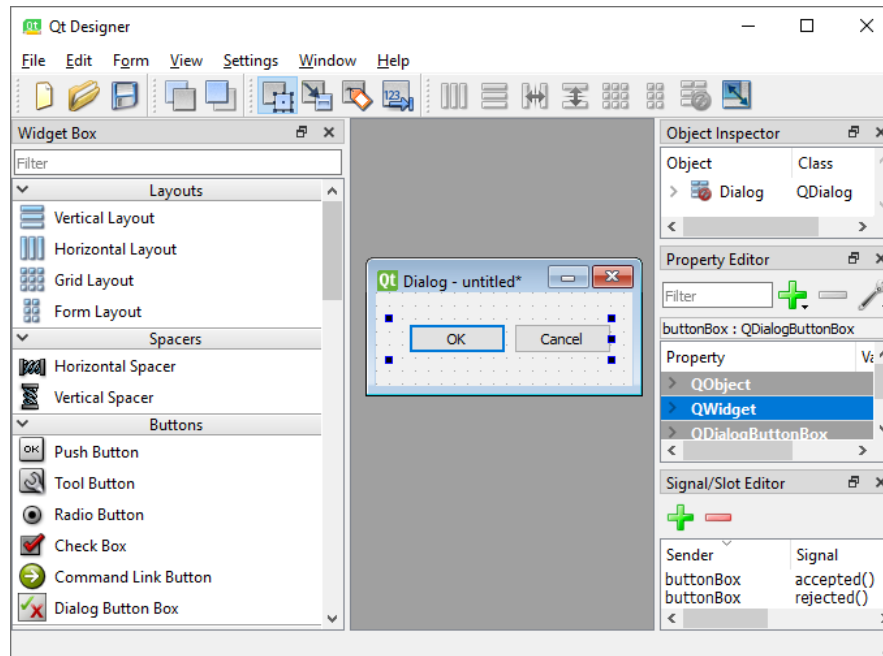
Figure 3.4: Qt Designer

```
form.setupUi(window)
window.show()
app.exec()
```

This assumes that you have saved the file as `dialog.ui`. The output is shown in Example 5.
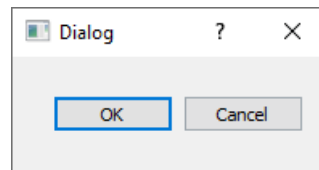
Example 5: A sample GUI created with Qt Designer

3.6.1 Qt Designer vs. Qt Creator

In addition to Qt Designer, there is also a tool called Qt *Creator*. This is a full-fledged and very powerful C++ IDE. It *contains* Qt Designer. Its power comes

at a price however: The download for Creator is gigabytes in size!

You can download Qt Designer from https://build-system.fman.io/qt-designer-download. It only weighs in at around 40 MB.

3.7   Using Qt Quick / QML with Python

As mentioned in Section 2.3, Qt can roughly be split into *Qt Quick* and *Widgets*. Most of this book is about Widgets. This section gives you a quick feel for how Qt Quick works, and how it can be combined with Python.

The code below is based on an example from the open source QML Book at qmlbook.github.io. You can find its license text in Appendix C. The QML Book is a good next read if you want to learn more about Qt Quick.

Example 6 shows the QML app we will build here. It displays a pin wheel in front of a pretty background. When the user clicks the mouse, the pin wheel slowly turns by 90 degrees. This is illustrated in Figure 3.5.
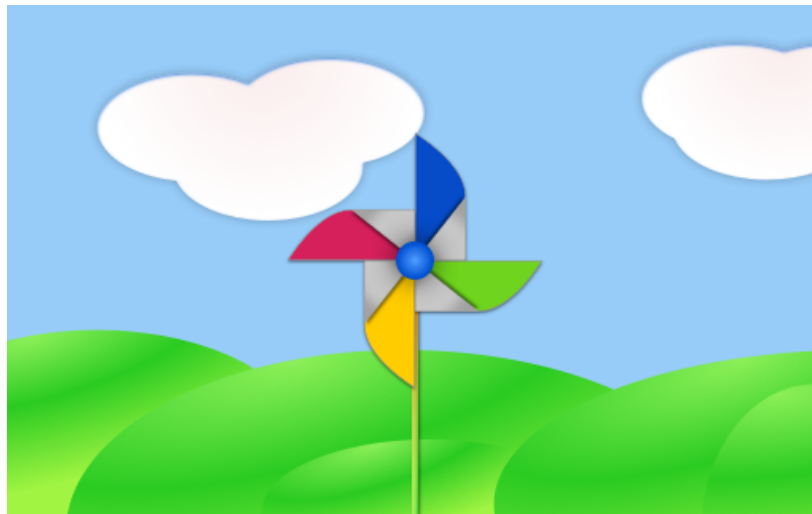
Example 6: A simple QML application

The necessary QML code is surprisingly easy to read:

```
import QtQuick 2.2
import QtQuick.Window 2.2
```

Figure 3.5: A turning pin wheel, animated with QML

```
Window {
    Image {
        id: background
        source: "background.png"
    }
    Image {
        id: wheel
        anchors.centerIn: parent
        source: "pinwheel.png"
        Behavior on rotation {
            NumberAnimation {
                duration: 250
            }
        }
    }
    MouseArea {
        anchors.fill: parent
        onPressed: {
            wheel.rotation += 90
        }
```

```
    }
    visible: true
    width: background.width
    height: background.height
}
```

In words, we create a window. It contains two images: the background and a pin wheel. These are shown in Figures 3.6 and 3.7. The pin wheel is centered inside the window. Rotations of it are animated with a duration of 250 ms. A `MouseArea` fills the entire window. When clicked, it rotates the pin wheel. Finally, the code displays the window and ensures it has the correct size.
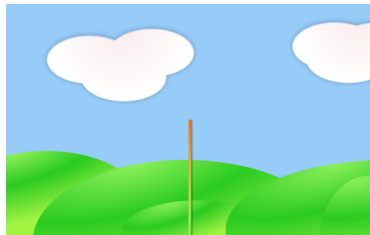
Figure 3.6: background.png

Figure 3.7: pinwheel.png

Running QML from Python is even easier: Save the above as `main.qml` and download the two images from the GitHub repository for this book. Then the application can be run with the following Python code:

```python
from PyQt6.QtQml import QQmlApplicationEngine
from PyQt6.QtWidgets import QApplication

app = QApplication([])
engine = QQmlApplicationEngine()
engine.load("main.qml")
app.exec()
```

Of course, Python and QML can be integrated more deeply than the above. For more information about this, please see the associated PyQt documentation.

Chapter 4

A text editor

In this chapter, we will create a simple text editor.
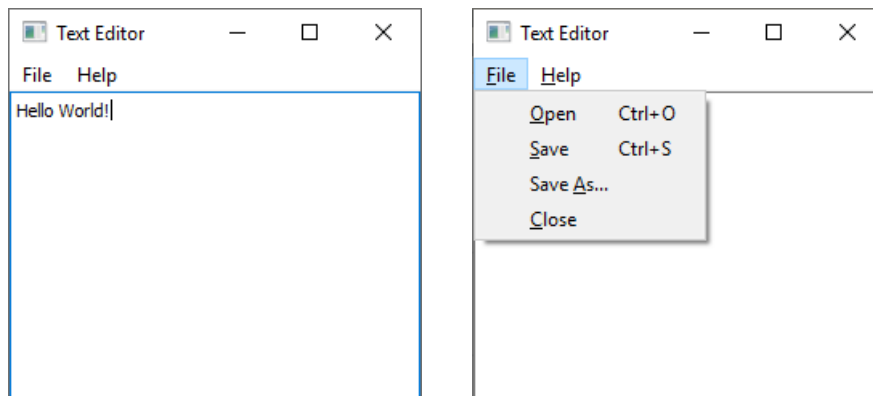
Example 7: A simple text editor

You might think this is not a very exciting use case. But! It's a great template for your own GUIs. You will learn how to implement:

- A main window
- Menus (*File* and *Help*)
- Keyboard shortcuts
- File dialogs (*Open* and *Save As*)
- A confirmation dialog: *'Save before closing?'*
- An *About* dialog.

23
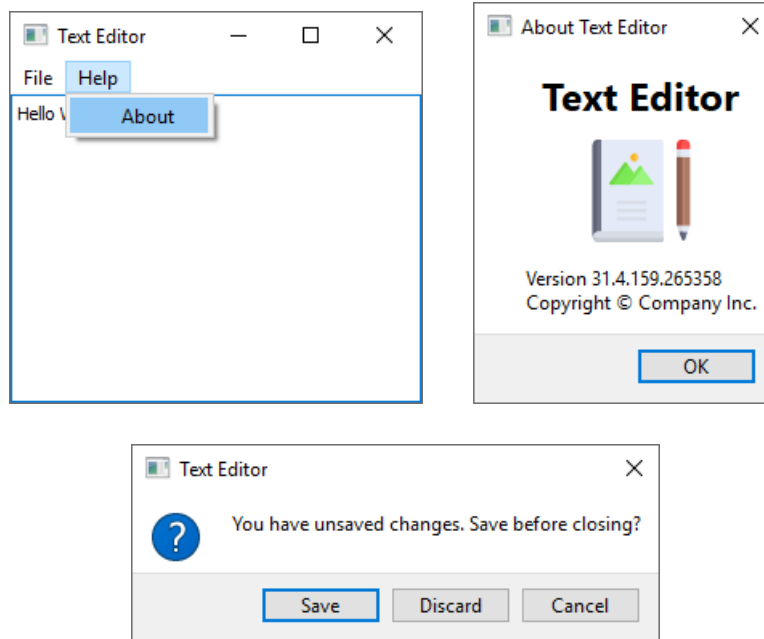
In the next chapter, we will turn the source code presented here into a standalone executable and an installer, so you can distribute it to other people.

4.1 Menus

We begin similarly to the examples in the previous chapter. This time however, we show a `QPlainTextEdit`. It is Qt's widget for (plain) multi-line text.

To make development easier, save the code below to a file called `main.py`. This way, you can then just invoke

```
python main.py
```

on the command line to run your app.

```python
from PyQt6.QtWidgets import *
app = QApplication([])
text = QPlainTextEdit()
text.show()
app.exec()
```
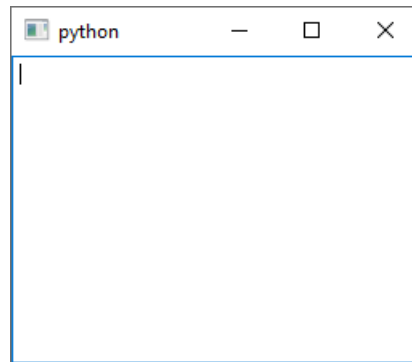
Figure 4.1: QPlainTextEdit

Figure 4.1 shows a screenshot of the resulting window. You can already type into it. Even undo (via `Ctrl/Cmd` + `Z`) and the context menu work.

A small shortcoming of this first version is that the window title is "python". We can set a more descriptive name via app.setApplicationName(...):

```python
from PyQt6.QtWidgets import *
app = QApplication([])
app.setApplicationName("Text Editor")
text = QPlainTextEdit()
text.show()
app.exec()
```

The resulting window with a title is shown in Figure 4.2.
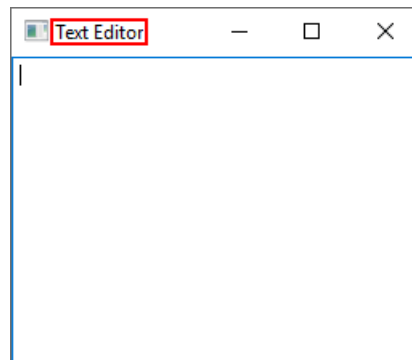
Figure 4.2: Changing the window title

(Please note that this doesn't work on macOS. Until we package our application in Chapter 5, you will still see the application title "Terminal" or "python".)

To create a menu, we need to introduce another widget: `QMainWindow`. This acts as a container and adds support for common features such as a menu, a tool bar, or a status bar.

The new code is:

```python
from PyQt6.QtWidgets import *
app = QApplication([])
app.setApplicationName("Text Editor")
text = QPlainTextEdit()
window = QMainWindow()
window.setWindowTitle("Text Editor")
window.setCentralWidget(text)
menu = window.menuBar().addMenu("&File")
close = QAction("&Close")
close.triggered.connect(window.close)
menu.addAction(close)
window.show()
app.exec()
```

Figure 4.3 shows how this gives us a *File* menu with a *Close* entry. When selected, this entry closes the window. (Note for macOS: If you don't see the menu straight away, use $\boxed{\text{Cmd}}$ + $\boxed{\text{Tab}}$ to briefly switch away from, and then back to, the window.)

So how does the new code work? First, we create a `QMainWindow`. We set its title to *Text Editor*. Then we use `setCentralWidget(...)` to make it display the text field. Next we add a menu. The ampersand in `"&File"` gives it the shortcut $\boxed{\text{Alt}}$ + $\boxed{\text{F}}$ on Windows and Linux. The character following the ampersand defines the shortcut. Eg. `"E&xit"` would be accessible with $\boxed{\text{x}}$.

When the user selects *Close*, the `.triggered` signal of our `QAction` is emitted. Just as in Section 3.5, where we first covered signals, we use `.connect(...)` to specify what should happen when this event occurs. In the present case, we want to close the window.
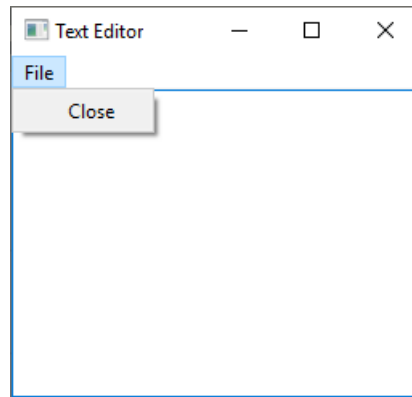
Figure 4.3: A simple File menu

You may wonder why the signal is called `triggered` and not (say) `clicked`. The reason is that a `QAction` can be invoked in many ways. An example we will see later is a *Save* action. Users can invoke it from the *File* menu. But they can also press `Ctrl/Cmd` + `S`. It's not really relevant which of the two paths was taken. `QAction` therefore abstracts away this information.

As the final step, we add the `QAction` to the menu and show the window. Note how this is different from the first version of the code, where we invoked `.show()` on the text field. This is no longer necessary because the window now contains (and thus displays) the text field.

4.1.1 Menus on macOS

Some platforms such as macOS and Ubuntu Unity display menus at the top of the screen instead of inside application windows. Figure 4.4 shows what this looks like on macOS. Note how the menu is *outside* the application window.
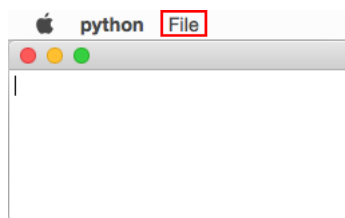
Figure 4.4: Our app's menu on macOS

There are a few other pecularities on macOS: Some names for menu entries are reserved. If we had named our entry `"Exit"` instead of `"Close"`, then our menu would not have been displayed at all! The reason for this is that there is already a `"Quit"` entry in the *python* menu in the screenshot. Because of its special name, our `QAction` would have automatically been tied to that existing entry. To prevent such behaviour and display the menu inside the window instead, you can call `window.menuBar().setNativeMenuBar(False)`.

4.2 An About dialog

Let's implement the Help menu next. Add the following code before the call to `window.show()`:

```python
help_menu = window.menuBar().addMenu("&Help")
about_action = QAction("&About")
help_menu.addAction(about_action)

def show_about_dialog():
    text = "<center>" \
            "<h1>Text Editor</h1>" \
            "&#8291;" \
            "<img src=icon.svg>" \
            "</center>" \
            "<p>Version 31.4.159.265358<br/>" \
            "Copyright &copy; Company Inc.</p>"
    QMessageBox.about(window, "About Text Editor", text)

about_action.triggered.connect(show_about_dialog)
```

This requires an image called `icon.svg` in your current directory. You can for instance use the one provided in the GitHub repository of examples for this book. It produces the dialog shown in Figure 4.5. Note that again, on macOS, the *About* entry will be shown in the *python* menu due to its special name.

The code for adding the menu and `QAction` is similar to before. What's different is that we now use `QMessageBox` to display a dialog. `QMessageBox` is the main class for showing alerts or asking the user simple questions such as "Do you really want to quit?". We use its `about(...)` method to display an
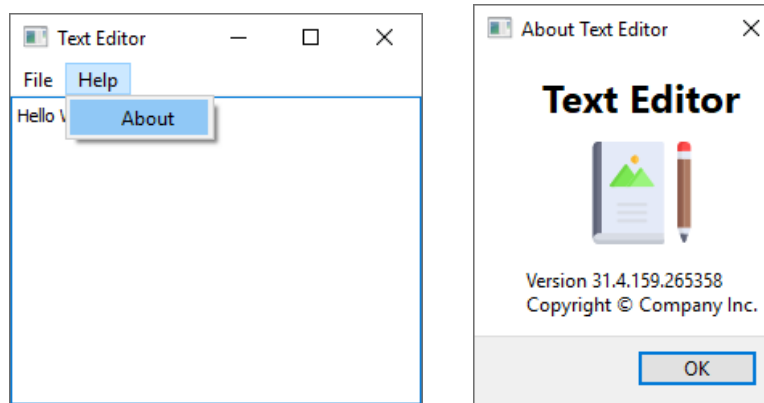
Figure 4.5: An About dialog

informative message.

As you can look up in its documentation, `QMessageBox.about(...)` takes several parameters. The first is the window over which the dialog should be centered. Then come the title and text. You can see that we supply HTML here.

Actually, it isn't HTML but a Qt dialect called *rich text*. It sometimes behaves differently from what you would expect in a browser. For example: Browsers display an `<img>` after a `<h1>` on a new line. Qt doesn't. That's why our code needs the special character #8291 ("invisible separator") to break the two apart. (We could have also used `<br/>` or `<p>...</p>` but they add too much space.)

4.2.1   From Qt's C++ docs to Python code

As mentioned in Section 2.4, Qt's documentation is so good that it should be your main reference. It is for `C++`. We will see now how you can use it for Python.

In the previous section, we used `QMessageBox` to display an alert. When you open its documentation, you find the link *Static Public Members* on the left (see Figure 4.6). It is always worthwhile to look at this link because Qt's most frequently used features are often exposed there.

The static members consist of a list of functions, truncated in Figure 4.7 to save space.
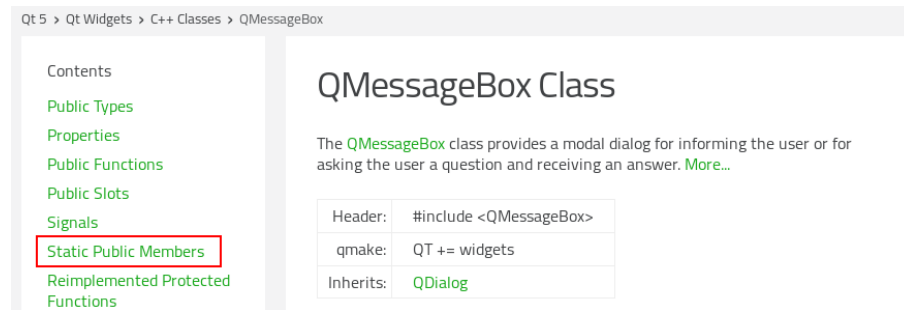
Figure 4.6: QMessageBox documentation

Figure 4.7: Static public members of QMessageBox

Already the first one, `about(...)`, is the one we used in the previous section. When you read its documentation, you quickly realise that it fits our use case perfectly.

Now, because `about(...)` is a *static* member of `QMessageBox`, you can invoke it in Python as `QMessageBox.about(...)`. With a little more reading, you can figure out the meaning of the parameters `parent`, `title` and `text` shown in Figure 4.7. Because the latter two are `QStrings`, you can simply supply them as Python strings `"..."`.

That is pretty much all there is to it. Don't be put off by the fact that Qt's documentation is for C++. With a little bit of translation, you can easily use it for Python. As mentioned before, it is an excellent resource. Make use of it!

4.3 Opening files

Let us now make it possible to open files. Here's a little challenge: If you're told that `QFileDialog` shows file dialogs in Qt, can you figure out from its documentation how to use it? (Solution in the next paragraph.)

Similarly to before, we look at the static members of `QFileDialog`. There, we

find `getOpenFileName(...)`. It "returns an existing file selected by the user". Perfect. Only its signature, shown in Figure 4.8, looks a little intimidating.

QString QFileDialog::getOpenFileName(QWidget *parent = nullptr, const `[static]`
QString &caption = QString(), const QString &dir = QString(), const QString
&filter = QString(), QString *selectedFilter = nullptr, QFileDialog::Options options = ...)

This is a convenience static function that returns an existing file selected by the user. If the user presses Cancel, it returns a null string.

```
QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"),
                                                "/home",
                                                tr("Images (*.png *.xpm *.jpg)"));
```

Figure 4.8: Signature of QFileDialog.getOpenFileName

Fortunately, all parameters are optional, as indicated by the `=...` after each one. We do want to give some of them, however. As before, we want to center the dialog over our window, so will supply the `parent`. Similarly, we'd like to give the dialog a caption. So we will use:

```
QFileDialog.getOpenFileName(window, "Open")
```

Then we need to process the result of `getOpenFileName(...)`. If you look at Figure 4.8 again, you'll see that `getOpenFileName` returns a string. But there's more: The asterisk in the parameter `QString * selectedFilter` marks it as a *pointer* to a string. This makes the Python binding return it as well. So we actually get a tuple, `(path, selectedFilter)`.

We're only interested in the `path`, so will do `getOpenFileName(...)[0]`. This is the empty string `""` when the user presses Cancel, so we'll use `if` ... to catch this case. Otherwise, we can use Python's `open(...).read()` to read the file into memory, and `text.setPlainText(...)` to update the text field.

The new code then looks as follows:

```
open_action = QAction("&Open")


def open_file():
    path = QFileDialog.getOpenFileName(window, "Open")[0]
    if path:
```

```
        text.setPlainText(open(path).read())
```

```
open_action.triggered.connect(open_file)
menu.addAction(open_action)
```

Add this before the line `close = QAction("&Close")`. This produces the screen-shots shown in Figure 4.9.
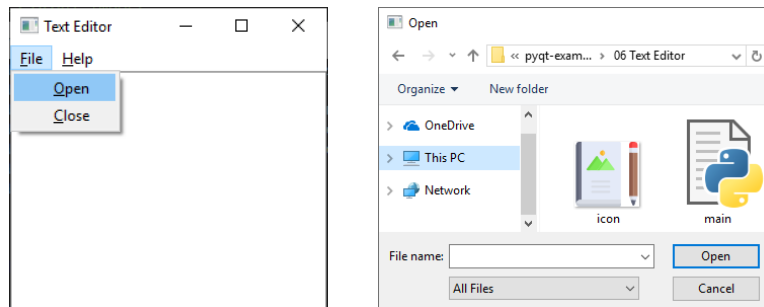
Figure 4.9: Open dialog

Two concluding notes about Figure 4.8: It contains several double colons `::`. You can generally replace these by single dots in Python. We did this when `QFileDialog::getOpenFileName` became `QFileDialog.getOpenFileName`. Finally, `tr(...)` is for Qt's translation mechanism. You don't have to use it.

4.4 Keyboard shortcuts

Every application on this planet supports `Ctrl/Cmd` + `O` for opening files, so we should have this feature as well. In our case of `QAction`, this is especially easy: We can simply call `action.setShortcut(...)`. The parameter we pass should be a `QKeySequence`. We can for instance do:

```
open_action.setShortcut(QKeySequence("Ctrl+O"))
```

But there is an even better way. `QKeySequence` defines many standard short-cuts. One of them is `QKeySequence.Open`. This stands for whichever keys are normally used to open files on the user's current operating system. So add the following before the line `menu.addAction(open_action)`:

```
from PyQt6.QtGui import QKeySequence
```

```
open_action.setShortcut(QKeySequence.Open)
```

This way, the dialog in Figure 4.9 also appears when you press `Ctrl/Cmd` + `O` .

You may wonder why we need the new **import**. Recall from the beginning of this chapter that the first line of our code is:

```
from PyQt6.QtWidgets import *
```

The classes we have used until now – QApplication, QPlainTextEdit, ... – lie in the QtWidgets module. QKeySequence on the other hand lies in QtGui. That's why the new import is needed.

Here is how you can find out which import you need to use a class such as QKeySequence. Open its documentation. Figure 4.10 shows that it says:

```
qmake: QT += gui
```


Figure 4.10: Documentation of the QKeySequence class

Because it mentions gui (and not, say, widgets), this means that you have to import QtGui.

4.5 Saving files

Let's implement the *Save As...* dialog next. Another challenge for you: We saw above that QFileDialog can display a file open dialog. Where in its documentation would you expect to find information about a dialog for saving files?

The answer is boringly repetitive at this point: In the public static members of QFileDialog. There, you'll find the method getSaveFileName(...) that does

what we want. Using it is very similar to `getOpenFileName(...)` from before:

```python
save_as_action = QAction("Save &As...")


def save_as():
    path = QFileDialog.getSaveFileName(window, "Save As")[0]
    if path:
        with open(path, "w") as f:
            f.write(text.toPlainText())


save_as_action.triggered.connect(save_as)
menu.addAction(save_as_action)
```

Again, add this before the line `close = QAction("&Close")`. Then you'll get the screenshots shown in Figure 4.11.
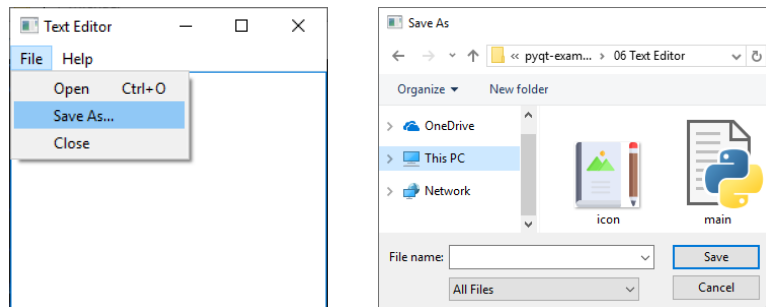
Figure 4.11: *Save As...* dialog

What's different is that we now save (instead of load) a file. We do this in the standard Python way:

```python
with open(..., "w") as f:
    f.write(...)
```

We save to the *path* selected by the user. To get the contents of the text field, we call its `.toPlainText()` method. The end result is that the text is saved to the chosen destination.

Before we continue, here is the current code for your reference:

```python
from PyQt6.QtWidgets import *
```

```python
from PyQt6.QtGui import QKeySequence

app = QApplication([])
app.setApplicationName("Text Editor")
text = QPlainTextEdit()
window = QMainWindow()
window.setCentralWidget(text)

menu = window.menuBar().addMenu("&File")
open_action = QAction("&Open")
def open_file():
    path = QFileDialog.getOpenFileName(window, "Open")[0]
    if path:
        text.setPlainText(open(path).read())
open_action.triggered.connect(open_file)
open_action.setShortcut(QKeySequence.Open)
menu.addAction(open_action)

save_as_action = QAction("Save &As...")
def save_as():
    path = QFileDialog.getSaveFileName(window, "Save As")[0]
    if path:
        with open(path, "w") as f:
            f.write(text.toPlainText())
save_as_action.triggered.connect(save_as)
menu.addAction(save_as_action)

close = QAction("&Close")
close.triggered.connect(window.close)
menu.addAction(close)

help_menu = window.menuBar().addMenu("&Help")
about_action = QAction("&About")
help_menu.addAction(about_action)
def show_about_dialog():
    text = "<center>" \
```

```
            "<h1>Text Editor</h1>" \
            "&#8291;" \
            "<img src=icon.svg>" \
            "</center>" \
            "<p>Version 31.4.159.265358<br/>" \
            "Copyright &copy; Company Inc.</p>"
    QMessageBox.about(window, "About Text Editor", text)
about_action.triggered.connect(show_about_dialog)

window.show()
app.exec()
```

4.6   Variable state

Another shortcut that every text editor should have is `Ctrl/Cmd` + `S`. This opens the *Save As...* dialog if we're editing a new file, or saves the current text otherwise.

To implement this behaviour, we need to remember the path of the current file. When we're creating a new document, the path is empty. But once we're editing an existing file, the path should tell us where it lies, so we can write to it when the user saves.

Let us introduce a variable to store the current file path. Add the following line to the top of your code, eg. above `menu = ...`:

```
file_path = None
```

Then add the following above `save_as_action = ...`:

```
save_action = QAction("&Save")

def save():
    if file_path is None:
        save_as()
    else:
        with open(file_path, "w") as f:
            f.write(text.toPlainText())
```

```
save_action.triggered.connect(save)
save_action.setShortcut(QKeySequence.Save)
menu.addAction(save_action)
```

This implements the *Save* functionality: We add a menu entry with the appropriate shortcut. When the user invokes it, we first check if `file_path` is **None**. If yes, this means we're editing a new file so we show the *Save As...* dialog. Otherwise, we use code we have already seen to write the current text to the remembered path.

All that remains is to make *Open* and *Save As...* update the `file_path` variable. Here's how to change `open_file()`. New lines are highlighted in bold:

```
def open_file():
    global file_path
    path = QFileDialog.getOpenFileName(window, "Open")[0]
    if path:
        text.setPlainText(open(path).read())
        file_path = path
```

So when the user opens a file, we set the `file_path` variable to the path he chose. The **global** keyword is required so our modification to the variable is visible outside the `open_file()` function. (If that doesn't make sense, please try a quick online search for "Python global".)

The changes to `save_as()` are slightly more involved. The main reason is that we want to reuse `save()` instead of duplicating **open**`...write...` from above.

```
def save_as():
    global file_path
    path = QFileDialog.getSaveFileName(window, "Save As")[0]
    if path:
        file_path = path
        save()
```

So when the user invokes *Save As...* and selects a file, the new implementation first updates the `file_path` variable. Then it simply lets `save(...)` handle the business of actually writing to disk.

4.7 An exit prompt

The final feature we want to implement is a warning if the user attempts to close the editor when there are unsaved changes. This is shown in Figure 4.12.
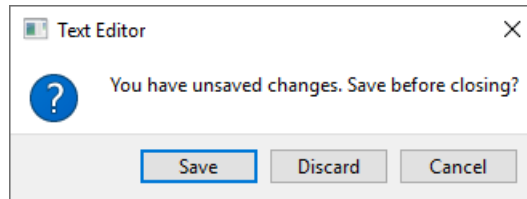
Figure 4.12: A prompt on exit

We don't just do this to make our text editor more complete. It also gives us a chance to talk about how you can implement your own prompts, how Qt's event system works, and how to extend Qt's classes.

4.7.1 Qt's event system

In Section 3.5, we saw how *signals* let you react to events such as the user clicking a button. However, signals are just the tip of the iceberg. Internally, Qt uses a different mechanism for processing events.

In any GUI application, many things can happen at the same time. A "Loading..." animation might play as data is being received. At any moment in between, the user can press Escape. Qt has to coordinate these activities in an orderly way.

The easiest way to ensure consistency is to process events one-by-one instead of in parallel. For example: Data coming in and the "Loading..." animation being updated could be a first event. The user pressing Escape another.

Qt keeps track of events in a *queue*. Whenever something happens (data coming in, the user pressing a key, etc.), it is not handled immediately. Instead, it is added to the queue. Qt then has a single *event loop*, in which it processes the items in the queue one-by-one.

As a matter of fact, we have already seen Qt's event loop numerous times. When you invoke

```
app.exec()
```

what you are doing is precisely to run Qt's event loop. It exits only when the application closes.

Events in Qt are represented by instances of the `QEvent` class (or one of its subclasses). For example, `QKeyEvent` contains information about which key was pressed, and whether a modifier such as `Ctrl` was pressed along with it.

To process an event, Qt delivers it to one or more objects. For example: Suppose the user presses `Esc` while the dialog on page 38 is open. Qt first delivers the corresponding `QKeyEvent` to the *Save* button, because it currently has the keyboard focus. However, this button does not know how to handle `Esc`. So Qt next notifies the button's parent, which in this case is the containing dialog window. This knows how to react to `Esc` and closes itself.

Technically, Qt delivers events to objects by invoking their `event(e)` method. This takes a single parameter, the `QEvent` e, and returns **True** or **False** depending on whether the event was handled. Qt keeps invoking `.event(e)` on further objects until one of them returns **True**.

In addition to the very general `event(...)`, which handles *all* events, there are also more specialised *event handlers*. A good example is `keyPressEvent(e)`, which is only called for key presses. Its parameter e is always a `QKeyEvent`. We will see another example, `closeEvent(...)`, in the next section.

Unlike `event(...)`, the specialised methods `keyPressEvent(e)` etc. do not return a value. Instead, they can call `e.ignore()` or `e.accept()` to suppress the event (or let it pass up the chain). We'll see an example of this too below.

4.7.2 Handling the close event

To react to the user closing the window, one can override `closeEvent(...)` as follows:

```python
class MainWindow(QMainWindow):
    def closeEvent(self, e):
        ...
```

The close event is *accepted by default*: Unless we call `e.ignore()`, it is propagated to `QApplication`, which closes the app. We do not need to call any

special Qt function, for this to happen. We can simply do nothing.

If there are unsaved changes, we want to show the dialog from page 38. When the user cancels this dialog, we want to prevent our app from closing. If they choose *Save*, we want to invoke the usual logic for saving. If they choose *Discard*, we simply do want to close the app.

A main question is how we can tell whether there are unsaved changes. Fortunately, Qt makes this pretty easy. The `QPlainTextEdit` exposes its data model through the `.document()` method. We can call `.document().isModified()` on it to check for changes.

Let us then see how we can show the dialog. We implemented the About functionality in Section 4.2 with `QMessageBox.about(...)`. Now, we can use `QMessageBox.question(...)` to offer several choices. The relevant code is:

```python
answer = QMessageBox.question(
    window, None,
    "You have unsaved changes. Save before closing?",
    QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel
)
if answer & QMessageBox.Save:
    # the user chose "Save"
elif answer & QMessageBox.Cancel:
    # the user canceled
```

This is pretty similar to `.about(...)` from before. The first parameter is the window over which the dialog should be centered. The second parameter is the dialog title – we use `None` here to use the default title. Next comes the text in the dialog, and finally the buttons. `QMessageBox` defines many standard buttons, and we use some of them here.

To specify multiple buttons, we use Python's *bitwise or operator* "|". Then, to test which of the buttons was selected, we use the *bitwise and operator* "&". The details of how this works are beyond the scope of this book. What is interesting however is that there are also other situations where you may have to use "|". For instance, to set the horizontal *and* vertical alignment of a text label, you can call `label.setAlignment(Qt.AlignRight | Qt.AlignVCenter)`.

The final question is how we can reset `.document().isModified()` when the user saves. (Recall from above that we use this snippet to check if there are unsaved changes.) Fortunately, here too the answer is easy: We can simply call `.document().setModified(`**`False`**`)`.

4.7.3   Putting it all together

With the above knowledge, we can now complete version 1.0 of our text editor! Add the following before the line `app = QApplication([])`:

```python
class MainWindow(QMainWindow):
    def closeEvent(self, e):
        if not text.document().isModified():
            return
        answer = QMessageBox.question(
            window, None,
            "You have unsaved changes. Save before closing?",
            QMessageBox.Save | QMessageBox.Discard |
            QMessageBox.Cancel
        )
        if answer & QMessageBox.Save:
            save()
        elif answer & QMessageBox.Cancel:
            e.ignore()
```

The highlighted lines were not yet explained: If the text was not modified, we **return** immediately to close our app. Otherwise, we prompt the user. If he wants to save the file, we invoke `save()`. Else, if he canceled the dialog, we use `e.ignore()` to stop the close event - and thus prevent the app from quitting.

For the above to take effect, we actually need to use our `MainWindow` class. So change the line that currently says:

```python
window = QMainWindow()
```

to:

```python
window = MainWindow()
```

As the very last step, change `save()` to reset our text field's modified state:

```python
def save():
    if file_path is None:
        save_as()
    else:
        with open(file_path, "w") as f:
            f.write(text.toPlainText())
        text.document().setModified(False)
```

And that's it! Our text editor now checks if there are unsaved changes upon closing. If yes, it gives the user a chance to save her work.

If you have made it this far, then congratulations! You created a first non-trivial application and learned about Qt's fundamental (and thus most important) mechanisms. Here is the final code again for your reference. As usual, you can also download it from the GitHub repository mentioned in Section 1.1.

```python
from PyQt6.QtWidgets import *
from PyQt6.QtGui import QKeySequence


class MainWindow(QMainWindow):
    def closeEvent(self, e):
        if not text.document().isModified():
            return
        answer = QMessageBox.question(
            window, None,
            "You have unsaved changes. Save before closing?",
            QMessageBox.Save | QMessageBox.Discard |
            QMessageBox.Cancel
        )
        if answer & QMessageBox.Save:
            save()
        elif answer & QMessageBox.Cancel:
            e.ignore()


app = QApplication([])
app.setApplicationName("Text Editor")
```

```python
text = QPlainTextEdit()
window = MainWindow()
window.setCentralWidget(text)

file_path = None

menu = window.menuBar().addMenu("&File")
open_action = QAction("&Open")
def open_file():
    global file_path
    path = QFileDialog.getOpenFileName(window, "Open")[0]
    if path:
        text.setPlainText(open(path).read())
        file_path = path
open_action.triggered.connect(open_file)
open_action.setShortcut(QKeySequence.Open)
menu.addAction(open_action)

save_action = QAction("&Save")
def save():
    if file_path is None:
        save_as()
    else:
        with open(file_path, "w") as f:
            f.write(text.toPlainText())
        text.document().setModified(False)
save_action.triggered.connect(save)
save_action.setShortcut(QKeySequence.Save)
menu.addAction(save_action)

save_as_action = QAction("Save &As...")
def save_as():
    global file_path
    path = QFileDialog.getSaveFileName(window, "Save As")[0]
    if path:
        file_path = path
```

```python
        save()
save_as_action.triggered.connect(save_as)
menu.addAction(save_as_action)

close = QAction("&Close")
close.triggered.connect(window.close)
menu.addAction(close)

help_menu = window.menuBar().addMenu("&Help")
about_action = QAction("&About")
help_menu.addAction(about_action)
def show_about_dialog():
    text = "<center>" \
           "<h1>Text Editor</h1>" \
           "&#8291;" \
           "<img src=icon.svg>" \
           "</center>" \
           "<p>Version 31.4.159.265358<br/>" \
           "Copyright &copy; Company Inc.</p>"
    QMessageBox.about(window, "About Text Editor", text)
about_action.triggered.connect(show_about_dialog)

window.show()
app.exec()
```

Chapter 5

Packaging and deployment

Before we go into more Qt topics, let us address another question that affects all GUI applications: How do you "compile" your source code into a standalone executable, so it runs on your users' computers?

In the Python world, the process of turning source code into a self-contained executable is called *freezing*. Although there are many libraries that attempt to solve this task – such as PyInstaller, py2exe, cx_Freeze, py2app, ... – freezing Python apps that use Qt has traditionally been a surprisingly hard problem.

We will use a new library called fbs. It specializes in Python/Qt applications. To install it, enter the command:

```
pip install fbs
```

(Recall from Section 3.1 that this requires the virtual environment to be active.)

The next section gives a quick tour of fbs. Then, we will see how you can use it to freeze the text editor from the previous chapter. Finally, we will also create an installer for the text editor.

5.1   A tour of fbs

Execute the following command to start a new fbs project:

```
fbs startproject
```

This asks you a few questions. You can for instance use *Tutorial* as the app name and your name as the author.

The command creates a new folder called `src/` in your current directory. This folder contains the minimum configuration for a bare-bones PyQt app.

Run the app

To run the basic PyQt application from source, execute the following command:

```
fbs run
```

This opens an (admittedly not very exciting) window. Figure 5.1 shows screenshots on Windows/Mac/Ubuntu.
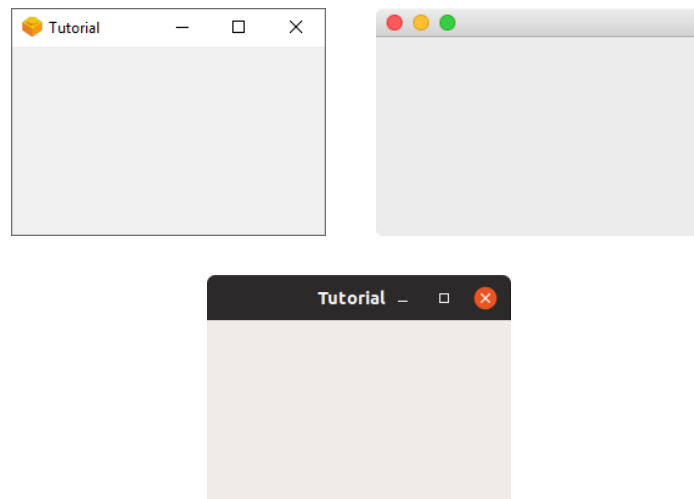

Figure 5.1: Initial GUI of fbs's startproject command

Source code of the sample app

Let's take a look at the source code of the PyQt app that was generated. It is at `src/main/python/main.py`

The first lines consist of imports:

```
from fbs_runtime.application_context.PyQt6 import \
```

```
    ApplicationContext
from PyQt6.QtWidgets import QMainWindow

import sys
```

Then comes an **if** statement:

```
if __name__ == '__main__':
    ...
```

This is a standard check in executable Python scripts. Explaining it here would unfortunately be beyond the scope of this book. Suffice it to say that it is often not absolutely necessary. To learn more, please google "python if name main".

The body of the **if** statement then does the actual work:

```
# 1. Instantiate ApplicationContext:
appctxt = ApplicationContext()

window = QMainWindow()
window.resize(250, 150)
window.show()

# 2. Invoke appctxt.app.exec():
exit_code = appctxt.app.exec()
sys.exit(exit_code)
```

The steps required for fbs are explained by comments. Highlighted in between is what the application actually does: It simply creates, resizes and then shows the window we saw in Figure 5.1.

Freezing the app

Use the following command to turn the app's source code into a standalone executable:

```
fbs freeze
```

This creates the folder `target/Tutorial`. You can copy this directory to *any* other computer (with the same OS as yours) and run the app there. It looks

easy, but this significant feat is one of fbs's primary purposes.

Creating an installer

Desktop applications are normally distributed by means of an installer. On Windows, this would be an executable called `TutorialSetup.exe`. On Mac, mountable disk images such as `Tutorial.dmg` are common. For Linux, `.deb` files are used on Ubuntu, `.rpm` on Fedora / CentOS, and `.pkg.tar.xz` on Arch.

fbs lets you generate each of the above packages via the command:

```
fbs installer
```

Depending on your operating system, this may require you to first install some tools. Please read on for OS-specific instructions.

Windows installer

Before you can use the `installer` command on Windows, you need to install NSIS and add its installation directory to your PATH environment variable.

The installer is created at `target/TutorialSetup.exe`. It lets your users pick the installation directory and adds your app to the Start Menu. It also creates an entry in Windows' list of installed programs. Your users can use this to uninstall your app. Figures 5.2 and 5.3 show these steps in action.

Mac installer

On Mac, the `installer` command generates the file `target/Tutorial.dmg`. When your users open it, they see the volume shown in Figure 5.4.

To install your app, your users simply drag its icon to the *Applications* folder (also shown in the volume).

Linux installer

On Linux, the `installer` command requires fpm. You can follow the instructions at https://fpm.readthedocs.io/en/latest/installing.html to install it.

fbs creates the installer inside the `target/` folder. Depending on your Linux distribution, it is called `Tutorial.deb`, `...pkg.tar.xz` or `...rpm`. Your users
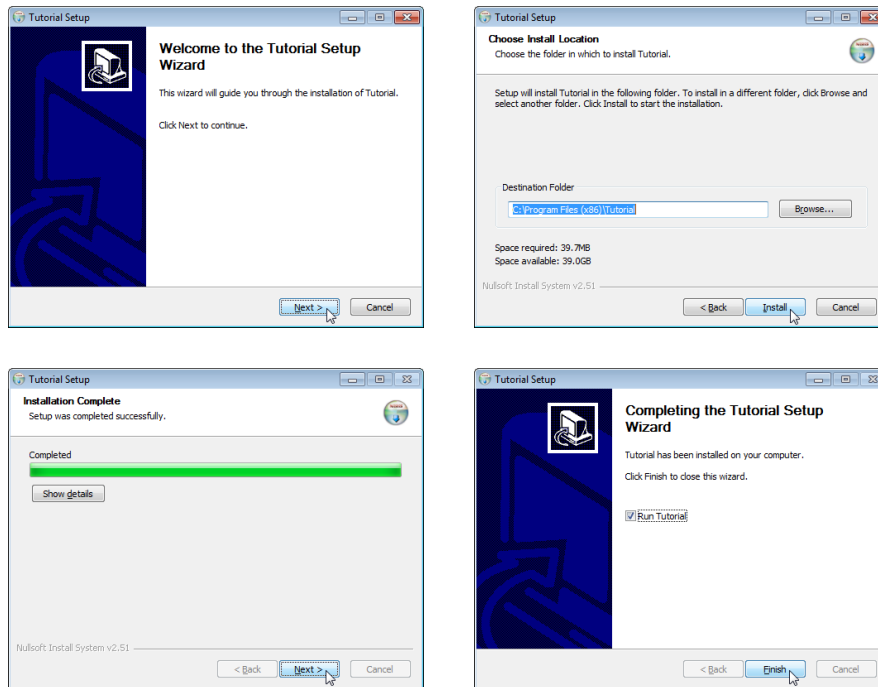
Figure 5.2: Windows installer created by fbs

can use these files to install your app with their respective package manager.

5.2 Compiling the text editor

Now that we know how to use fbs for a sample application, let us apply this knowledge to our text editor. If you executed `fbs startproject` in the previous section, please delete the `src/` directory. We want to start from scratch.

Now execute `fbs startproject`. Please enter `Text Editor` as the application name.

This creates the file `src/main/python/main.py`. Its contents were shown on page 46. We will now make a series of changes to migrate our text editor to it.

First, please remove the `if __name__ == '__main__'` statement and the indent (four spaces) of all the following lines. The `if` is not necessary for our use case. After your change, `fbs run` should still show the window in Figure 5.1.
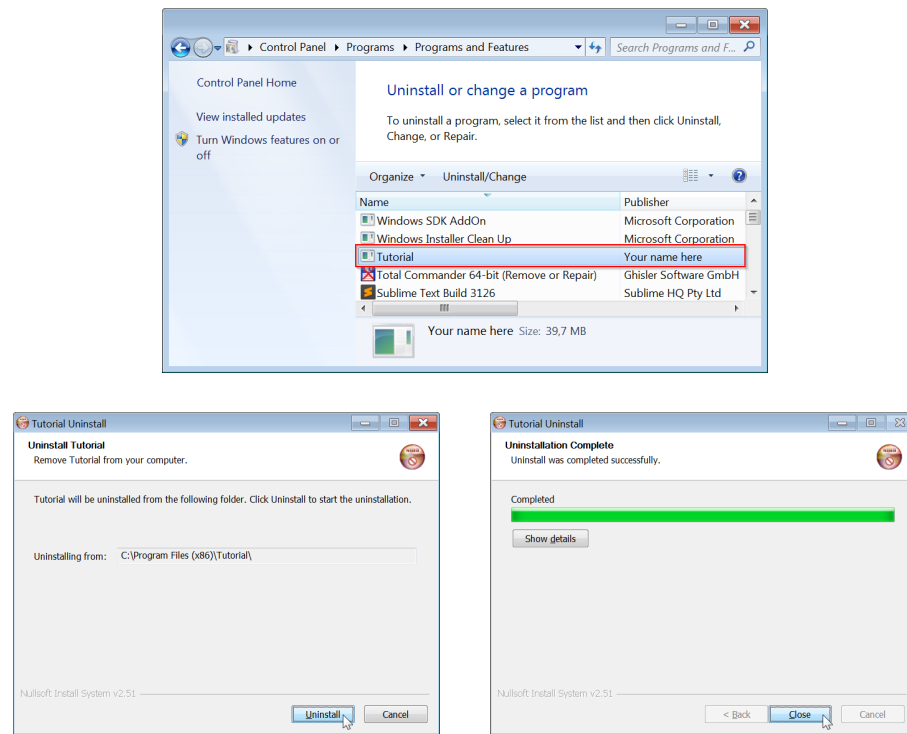
Figure 5.3: Windows uninstaller created by fbs

Next, please replace the following three lines by the text editor's *entire* code shown on pages 42ff.:

```
window = QMainWindow()
window.resize(250, 150)
window.show()
```

Then delete the following line:

```
app.exec()
```

It's no longer needed because of `appctxt.app.exec()` at the end of the code.

The previous change makes the following line obsolete. Please remove it too:

```
app = QApplication([])
```

Also delete the following line:
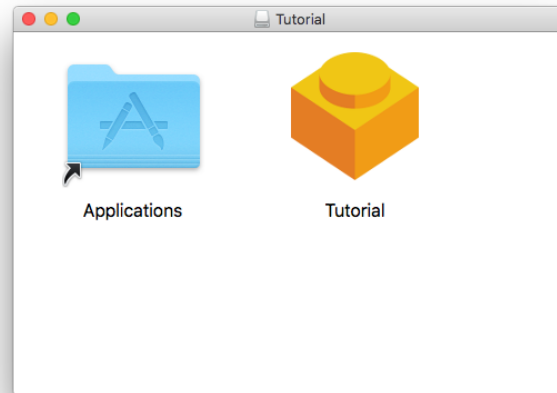
Figure 5.4: Mac installer created by fbs

```
app.setApplicationName("Text Editor")
```

fbs automatically does this for us because we entered `Text Editor` as the application name in the `startproject` command.

5.2.1 Resource files

Now for a meatier change. Do you remember the text of the *About* dialog in Section 4.2?

```
text = "<center>" \
       "<h1>Text Editor</h1>" \
       "&#8291;" \
       "<img src=icon.svg>" \
       "</center>" \
       "<p>Version 31.4.159.265358<br/>" \
       "Copyright &copy; Company Inc.</p>"
```

This references an image file, `icon.svg`. Consider where this file lies in various situations: While implementing the text editor, we had it next to `main.py`. But when a Windows user runs our app, the image will be in the installation folder. And on macOS, it needs to be in a subdirectory of that, `Contents/Resources`.

In other words, the actual image path greatly depends on the environment.

Here is how fbs solves this for us: Copy `icon.svg` into the (new) directory `src/main/resources/base`. (Remember that you can download the image file from the GitHub repository mentioned in Section 1.1.) Then change the above code to:

```
text = "<center>" \
        "<h1>Text Editor</h1>" \
        "&#8291;" \
        "<img src=\"%s\">" \
        "</center>" \
        "<p>Version 31.4.159.265358<br/>" \
        "Copyright &copy; Company Inc.</p>" \
        % appctxt.get_resource("icon.svg")
```

This replaces `"icon.svg"` by the absolute path to the file. fbs transparently handles all the details. In particular, its `freeze` command ensures that the image ends up in the correct subfolder of the installation directory.

There is a problem however: When we use `fbs run` now to start the text editor, its *About* dialog shows a superfluous icon. This is shown in Figure 5.5.
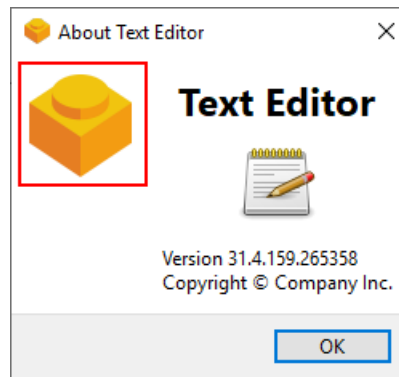

Figure 5.5: Superfluous icon in the About dialog

The reason for this is that fbs configures Qt to use an application icon. This appears in places such as the window title bar and the task switcher (`Alt/Cmd` + `Tab`). Unfortunately, `QMessageBox.about(...)` displays it as well.

To fix this, change the next line

```
QMessageBox.about(window, "About Text Editor", text)
```

to:

```
about_dialog = QMessageBox(window)
about_dialog.setText(text)
about_dialog.exec()
```

When you then do `fbs run`, you get the correct *About* dialog again.

5.2.2 Customizing the icon

fbs sets the application icon from image files in the folder `src/main/icons`. You should have this directory after `fbs startproject`. When you look inside, you will find the logo shown in Figure 5.5. This is fbs's default icon.

Let's replace it by our text editor's actual icon. Download the GitHub repository mentioned in Section 1.1 and open the directory for example 8. In it, you will find a `src/main/icons` subfolder. Copy it over the one you have locally. When you then do `fbs run`, the correct icon will be displayed in the window title and during `Alt` + `Tab`. This is shown on Windows in Figure 5.6. On macOS, please note that this will again not work until we do `fbs freeze` below.
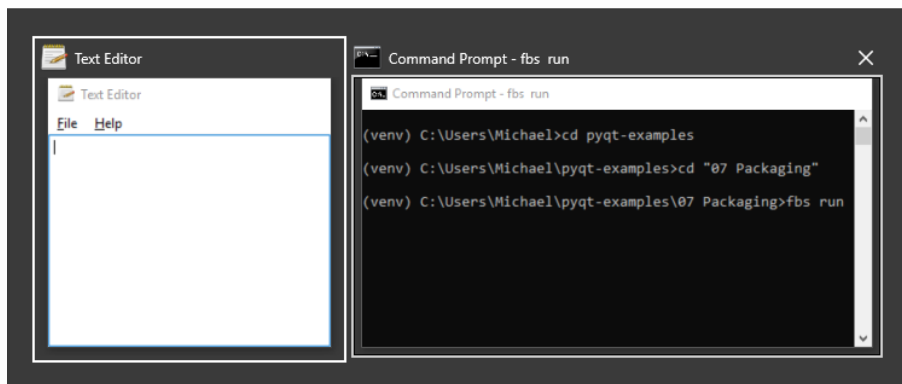
Figure 5.6: Custom icons in the window title and the Alt+Tab dialog

5.2.3   Final result

We are now in a position to package our text editor. Enter the following:

```
fbs freeze
```

This freezes the app to the `target/Text Editor` directory. As before, you can copy this folder to any other computer and run the text editor there. Our little app has grown wings!

The frozen form of the text editor finally also has all the nice features on Mac. These are shown in Figures 5.7 to 5.9.
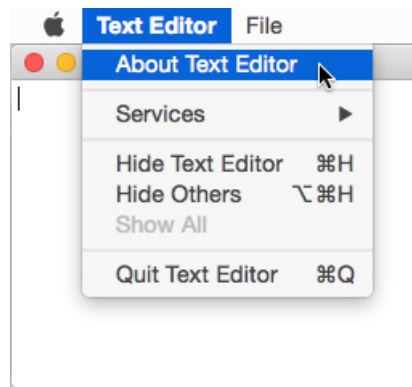
Figure 5.7: Correct menu and app titles on Mac
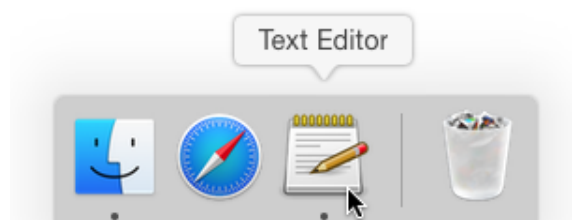
Figure 5.8: Text Editor in the Dock on macOS

5.3   Creating an installer

After the above preparations (including those in Section 5.1), creating an installer is as simple as:
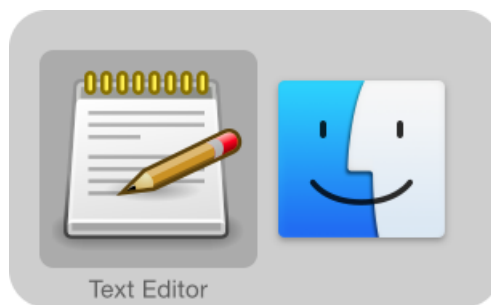
Figure 5.9: Text Editor in macOS's App Switcher

```
fbs installer
```

This places the installer inside the `target/` directory. Figures 5.10 to 5.12 show some screenshots.
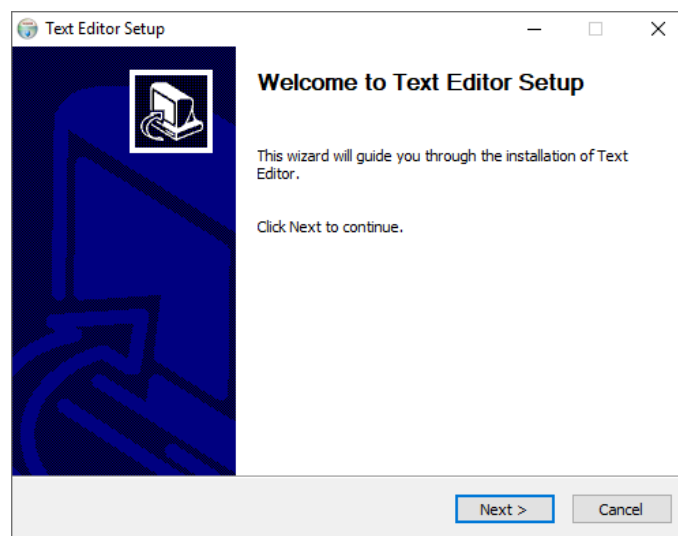
Figure 5.10: Text Editor installer on Windows
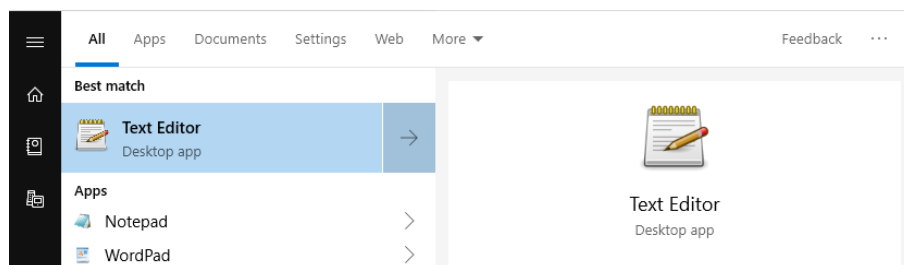
Figure 5.11: Text Editor entry in the Windows start menu
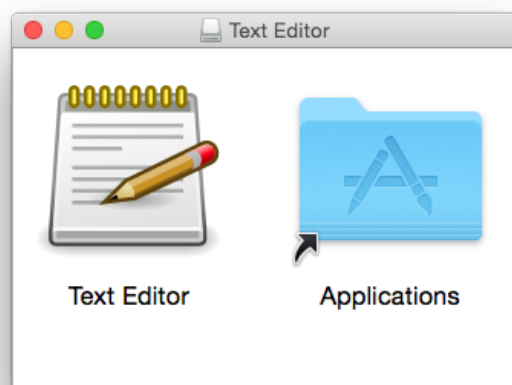
Figure 5.12: Text Editor installer on Mac

Chapter 6

Custom styles

One of Qt's strengths is its support for custom styles. There are many mechanisms that let you customize the look and feel of your application. This chapter outlines the most important ones.

6.1 Built-in styles

The coarsest way to change the appearance of your GUI is to set the global style. Recall the widgets screenshot from Section 3.3, shown here again in Figure 6.1.

This uses a style called *Fusion*. If you use the *Windows* style instead, then it looks as in Figure 6.2.

To apply a style, use `app.setStyle(...)`:

```python
from PyQt6.QtWidgets import *
app = QApplication([])
app.setStyle("Fusion")
...
```

The available styles depend on your platform. Usually, there are `"Fusion"`, `"Windows"`, `"WindowsVista"` (Windows only) and `"Macintosh"` (Mac only).
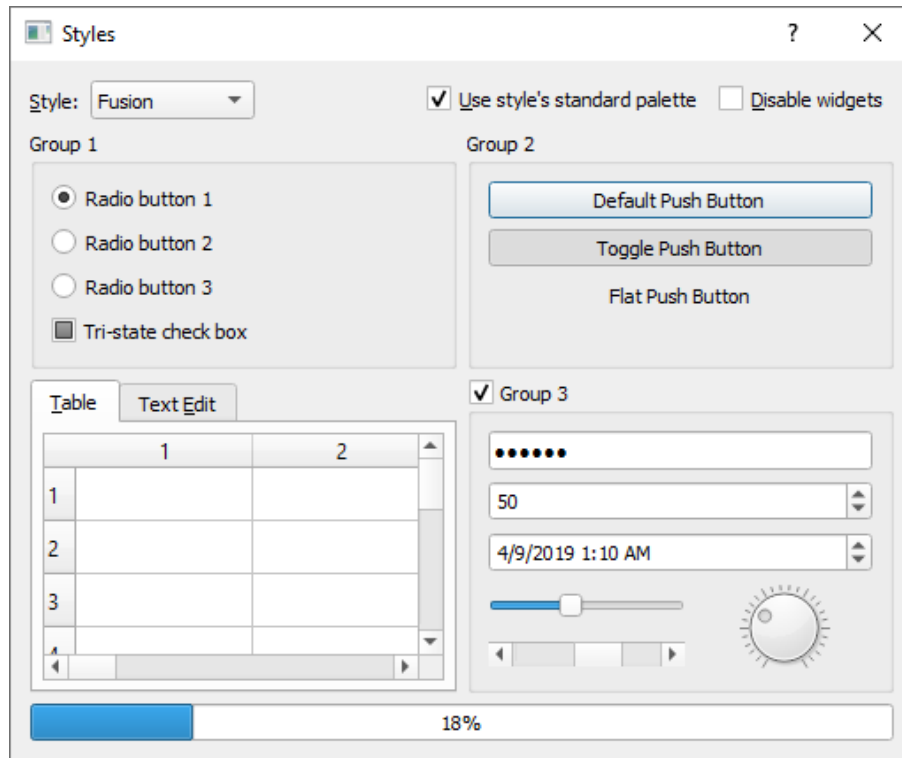
Figure 6.1: Common widgets rendered with the Fusion style

6.2   Custom colors

If you like a style but want to change its colors (eg. to a dark theme) then you
can use `QPalette` and `app.setPalette(...)`. For example:

```python
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPalette
from PyQt6.QtWidgets import QApplication, QPushButton

app = QApplication([])
app.setStyle("Fusion")
palette = QPalette()
palette.setColor(QPalette.ButtonText, Qt.red)
app.setPalette(palette)
button = QPushButton("Hello World")
```
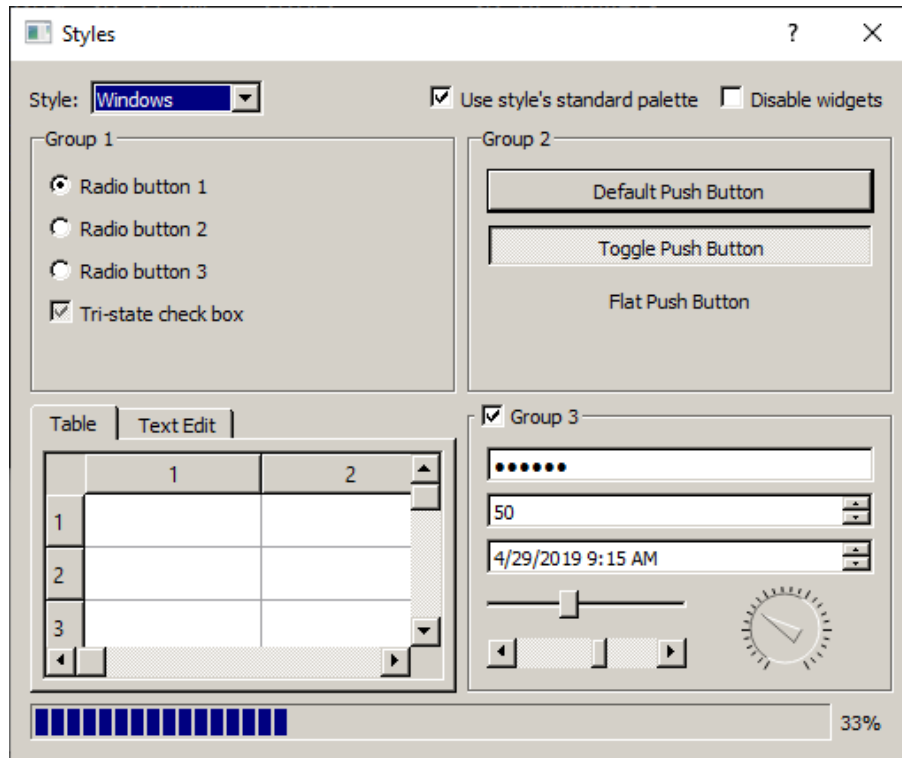
Figure 6.2: Common widgets rendered with the Windows style

```
button.show()
app.exec()
```

Figure 6.3 shows how this changes the text color in buttons to red.
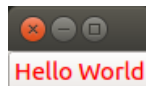
Figure 6.3: QPushButton with red text

In recent years, dark themes have also become very popular. Example 9 shows what the text editor from Chapter 4 looks like with a dark theme. For the code, please see the usual GitHub repository. It simply uses a `QPalette` like above.
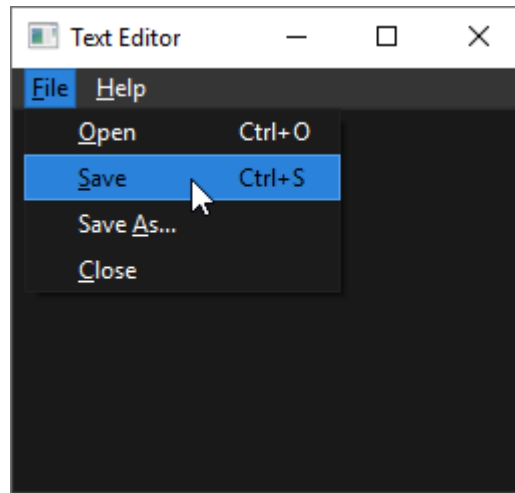
Example 9: Text editor with a dark theme

6.3 Style sheets

In addition to the above, you can change the appearance of your application via Qt style sheets (QSS). This is Qt's analogue to CSS. We can for example use it to add some spacing:

```python
from PyQt6.QtWidgets import QApplication, QPushButton
app = QApplication([])
app.setStyleSheet("QPushButton { margin:  10ex; }")
button = QPushButton("Hello World")
button.show()
app.exec()
```

Figure 6.4 shows the resulting button, surrounded by more space than usual.

Once your style sheet reaches a certain size, it makes sense to put it into a separate file, rather than inline as a string like in the above code. Here is an elegant way to do this if you are using fbs as described in Chapter 5:

```python
appctxt = ApplicationContext()
with open(appctxt.get_resource("styles.qss")) as f:
    appctxt.app.setStyleSheet(f.read())
```
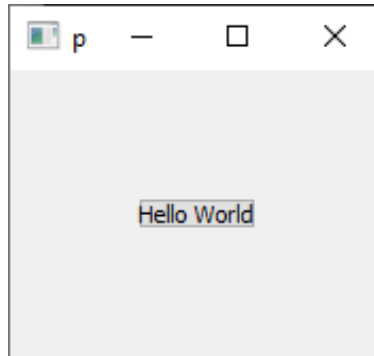
Figure 6.4: Adding spacing around a button with QSS

With these lines, you can simply write all your QSS declarations in the file `src/main/resources/base/styles.qss`.

While QSS often does not always behave exactly as one would expect from HTML and CSS, it is generally extremely powerful. For specific information on its supported elements and properties, please consult Qt's documentation.

6.4   Custom rendering: An action shooter

Sometimes, not even the above mechanisms are enough to get the appearance you want. In these cases, you can also render GUI elements manually.

Let us demonstrate this by turning our text editor into an action shooter. Add the following to the top of the code shown on page 42ff.:

```python
from PyQt6.QtGui import *
from PyQt6.QtCore import *

class PlainTextEdit(QPlainTextEdit):
    def __init__(self):
        super().__init__()
        self._holes = []
        self._bullet = QPixmap("bullet.png")
    def mousePressEvent(self, e):
        self._holes.append(e.pos())
        super().mousePressEvent(e)
```

```python
        self.viewport().update()
    def paintEvent(self, e):
        super().paintEvent(e)
        painter = QPainter(self.viewport())
        for hole in self._holes:
            painter.drawPixmap(hole, self._bullet)
```

Also replace the line

```python
text = QPlainTextEdit()
```

by

```python
text = PlainTextEdit()
```

Finally, download `bullet.png` from example directory number 10 in the GitHub repository. Place this image into your current working directory.

When you run the new code and click inside the text field, bullet holes appear. This is shown in Example 10.
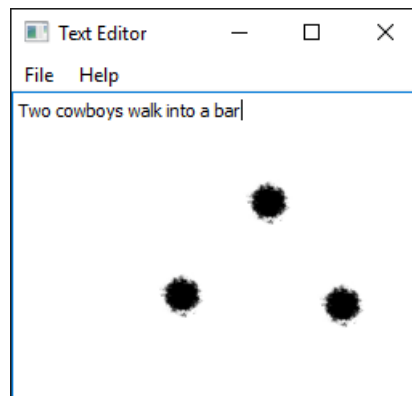
Example 10: A text editor turned action shooter

For added effect, you can add the following to the end of `mousePressEvent`:

```python
from PyQt6.QtMultimedia import QSound
QSound.play("shot.wav")
```

This requires that you also download `shot.wav` from the GitHub repository.

Once you do, there's even sound ;-)

But let us turn back to the code. The standard way to implement custom rendering in Qt is to override `paintEvent(...)` and use a `QPainter` to perform the actual drawing. This would give us:

```python
class MyWidget(QWidget):
    ...
    def paintEvent(self, e):
        painter = QPainter(self)
        # Now render - eg. painter.drawText(...).
```

However, our code is a little bit different. First, we invoke `QPlainTextEdit`'s default rendering implementation via `super().paintEvent(e)`. This draws the text field as usual. Our further steps then paint on top of it.

Next, we create a `QPainter`, but for `self.viewport()` instead of `self`. We do this because the text field is scrollable (see Figure 6.5). Qt implements this by making `QPlainTextEdit` a subclass of `QAbstractScrollArea`. The documentation for its `paintEvent(...)` tells us to draw on `.viewport()`. This is the widget that actually displays the text (and lies behind the scroll bars).
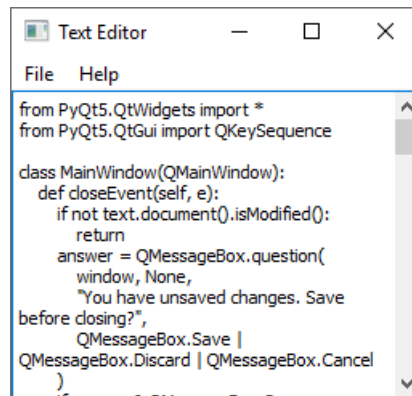
Figure 6.5: Text Editor showing its own code – with a scrollbar

Finally, our `paintEvent(...)` draws each of the holes "shot" by the user. The holes' positions were saved to a list in `mousePressEvent(...)`. This method also invokes Qt's default handler `super().mousePressEvent(e)`. And it calls `viewport().update()` to request a repaint.

It will not surprise you that custom rendering in Qt can be much more complex than the above. But the main ingredients are here: Override `paintEvent(...)`, create a `QPainter`, and call `update()` to request a repaint.

Beyond this, Qt also offers more specialised hooks for tweaking the styling. A good example of this are item delegates. These for instance let you customize the look and feel of individual cells in a table.

6.5   Tips and tricks

In a typical project, you will gradually use more and more of the above techniques. You may start out with `app.setStyle("Fusion")` for a modern look. Maybe you'll want to tweak the colors with `QPalette`. Almost certainly, you'll use style sheets to set the spacing and font sizes of your widgets. And if all else fails, you might implement your own rendering logic.

Here then are a few tips to help you do the above more effectively:

In CSS, you can use `#id` to select an element by its ID. QSS has a similar feature: If you call `.setObjectName("name")` on a widget, then you can select it with `#name`. This is very useful for styling individual GUI elements.

Qt's support for High DPI Displays is not entirely seamless. In particular, margins and font sizes can vary wildly across displays. A way to alleviate this is to **never use pixel measurements in style sheets**. Instead, use units that are independent of monitor resolution. These include **pt**, **em** and **ex**.

Even if you don't use pixel measurements, your app will likely not look consistent across operating systems. To work around this, you can use **OS-specific style sheets**. If you are using fbs as described in Chapter 5, you can for instance implement this as follows:

Place those of your style declarations that apply to all operating systems into `src/main/resources/base/styles.qss`. Then put further OS-specific styles into `os_styles.qss` in the other subdirectories `windows/`, `mac/` and `linux/` of the `resources` folder. Use the following code:

```
from fbs_runtime.application_context.PyQt6 import \
    ApplicationContext
```

```python
appctxt = ApplicationContext()
stylesheet = ""
for fname in ("styles.qss", "os_styles.qss"):
    stylesheet += open(appctxt.get_resource(fname)).read()
appctxt.app.setStyleSheet(stylesheet)
```

This concatenates the definitions from `styles.qss` and `os_styles.qss`. The trick is that `.get_resource(...)` returns the path for the user's current operating system. For example, when this method is executed on Windows, then **windows/**`os_styles.qss` from the `resources` directory is returned. As a result, only the styles for the current operating system are applied.

Chapter 7

Using threads for a chat client

Threads allow you to execute multiple strands of code in parallel. In this chapter, we will see how this can make your application more responsive.

We will create a chat client that actually works: You can use it to leave messages for other readers of this book. Figure 11 shows what it looks like.
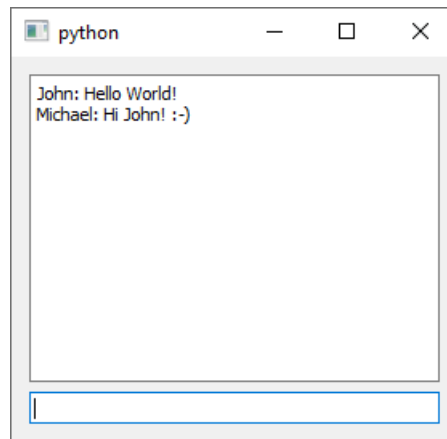
Example 11: A chat client

The easiest way to display the messages is via `QPlainTextEdit`:

```python
from PyQt6.QtWidgets import *
app = QApplication([])
text = QPlainTextEdit()
```

66

```
text.show()
app.exec()
```

Figure 7.1 shows a screenshot of the resulting window.
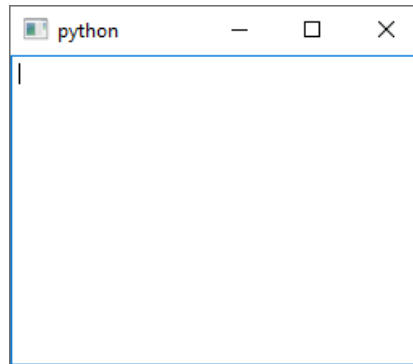
Figure 7.1: An empty text area

We now want to add a text field for typing messages below the text area. This is shown in Figure 7.2.
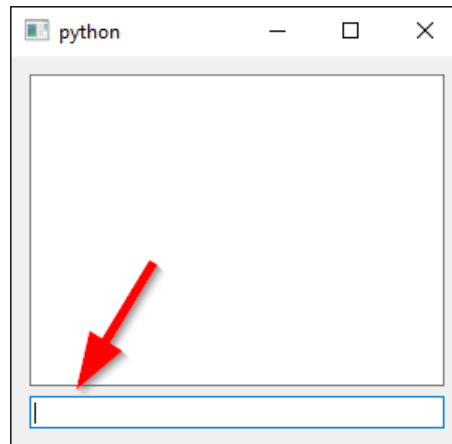
Figure 7.2: Text field for sending chat messages

We can use Qt's class `QLineEdit` to get a single-line text field. As described in Section 3.4, `QVBoxLayout` then lets us position it below the text area:

```python
from PyQt6.QtWidgets import *
app = QApplication([])
```

```
window = QWidget()
layout = QVBoxLayout()
layout.addWidget(QTextEdit())
layout.addWidget(QLineEdit())
window.setLayout(layout)
window.show()
app.exec()
```

7.1 Talking to the server

With the GUI ready, we need to connect it to a server. The easiest way to do this is via the `requests` library. You can install it via the command:

```
pip install requests
```

Once you have done this, start `python` and enter the following:

```
from requests import Session
server = Session()
chat_url = "https://build-system.fman.io/chat"
print(server.get(chat_url).text)
```

This should give you the latest chat messages.

We can also send a message. Below is an example. Be sure to use your own name and text :-)

```
server.post(chat_url, {"name": "John", "message": "Nice book"})
```

When you then call `server.get(chat_url).text` again, you should see your message.

7.2 Signals

Our chat client needs to handle certain events:
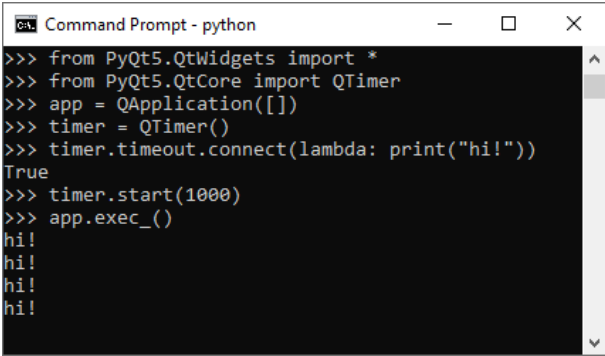
- The current message should be sent to the server when you press `Enter`.
- Once per second, we want to display any new messages from the server.

As we saw in Section 3.5, we can use *signals* to do this. For the latter case of executing code at regular intervals, Qt offers a convenient class called `QTimer`:

```python
from PyQt6.QtWidgets import *
from PyQt6.QtCore import QTimer
app = QApplication([])
timer = QTimer()
timer.timeout.connect(lambda:  print("hi!"))
timer.start(1000)
app.exec()
```

The signal in the above code is `timer.timeout`. We used its `.connect(...)` method to specify a function that gets called when the signal occurs. In the example, we gave the inline function `lambda: print("hi!")`. Our other call `timer.start(1000)` then made Qt run this code every 1,000 milliseconds.

When you execute the above code, the message `hi!` appears in your terminal once per second. This is shown in Figure 7.3.

Figure 7.3: QTimer executing code once per second

7.3   Initial implementation

We now have enough information to implement a first – single-threaded – version of the chat client. Copy the below code into `01_single_threaded.py` (or download this file from the GitHub repository mentioned in Section 1.1). Be sure to fill in your name at the top, or you won't be able to post messages. Then you can run the chat with the following command:

```
python 01_single_threaded.py
```

You'll be able to see what others have written and send messages of your own. Happy chatting! ;-)

The code should hopefully be pretty easy to follow for you by now. It only contains a few things which we have not yet seen. They are explained below.

```python
from PyQt6.QtCore import *
from PyQt6.QtWidgets import *
from requests import Session

name = "" # Enter your name here!
chat_url = "https://build-system.fman.io/chat"
server = Session()

# GUI:
app = QApplication([])
text_area = QPlainTextEdit()
# Make the text area read-only:
text_area.setFocusPolicy(Qt.NoFocus)
message = QLineEdit()
layout = QVBoxLayout()
layout.addWidget(text_area)
layout.addWidget(message)
window = QWidget()
window.setLayout(layout)
window.show()

# Event handlers:
def display_new_messages():
    new_message = server.get(chat_url).text
    if new_message:
        text_area.appendPlainText(new_message)

def send_message():
    server.post(chat_url, {
```

```
        "name": name, "message": message.text()
    })
    message.clear()

# Signals:
message.returnPressed.connect(send_message)
timer = QTimer()
timer.timeout.connect(display_new_messages)
timer.start(1000)

app.exec()
```

We use `.setFocusPolicy(Qt.NoFocus)` to prevent the text area from ever receiving the keyboard focus. This effectively makes it non-editable. After all, unlike in the text editor from Chapter 4, we don't want the user to be able to freely type into the text area.

The `.returnPressed` signal of the single-line text field for sending messages is emitted when the user presses `Enter`. We connect our `send_message()` function to it. This uses code we have already seen to send the data to the server. Then it empties the text field via `message.clear()`, so the user can type in the next message.

As we saw further above, we can use `QTimer` to invoke code periodically. Here, we make Qt call our `display_new_messages()` function every 1,000 milliseconds. This polls the server for new messages. If there is a new message, the code uses `.appendPlainText(...)` to add it to the bottom of the text area.

Note that the server actually does a lot of work for us. In particular, it keeps track of the messages we have already seen, and only sends us new ones. It does this by setting a cookie in our `Session` object. Every request we make sends this cookie back to the server, and thus lets it identify us.

7.4 Threads

When you type a long message in the above chat client, you may notice that it lags regularly. The reason for this is that Qt cannot process user input while executing the `display_new_messages()` function. Because this function takes

a few hundred milliseconds to receive a response from the server, and because it is called every second, this means that the GUI is frequently unresponsive.

To fix this, we will add parallel processing via threads. The idea is that Qt keeps handling user input in the so-called *main thread* while another thread of ours fetches new messages from the server in the background.

The necessary code is below. Simply paste it over display_new_messages above. Alternatively, you can find the entire new implementation in the file 02_multithreaded.py in the GitHub repository.

```python
from threading import Thread
from time import sleep

new_messages = []
def fetch_new_messages():
    while True:
        response = server.get(chat_url).text
        if response:
            new_messages.append(response)
        sleep(.5)

thread = Thread(target=fetch_new_messages, daemon=True)
thread.start()

def display_new_messages():
    while new_messages:
        text_area.appendPlainText(new_messages.pop(0))
```

The code starts a Thread that runs a new function, fetch_new_messages, in the background. This function uses an infinite **while** loop to load new messages from the server and to place them in a list of new_messages. The new implementation of display_new_messages then processes this list to display the messages in the GUI.

There are several things to note about the above snippet: First, remember that also the new implementation of display_new_messages is connected to timer.timeout and thus gets called every second. Further, the call sleep(.5)

prevents us from flooding the server with too many requests.

But the most subtle point is this: In `fetch_new_messages()`, why didn't we just call `text_area.appendPlainText(...)`? We could have avoided the complicated use of `new_messages`.

The reason for this is that we now have two threads running in parallel: Qt's main thread (i.e. the thread from which we call `app.exec()`) and the new thread above. At any moment, neither thread knows what the other one is currently doing. But consider: What if Qt happens to render the text area while the new thread is in the middle of calling `text_area.appendPlainText(...)`? The answer would probably be a crash.

We therefore only make changes to the GUI from the main thread. We do this by using `QTimer` to make Qt invoke our `display_new_messages()` in the main thread. This makes it safe for us to change the `text_area`, because once we're in the main thread, it is guaranteed that we are the only ones touching the GUI.

One thing to note is that we now rely on `.append(...)` and `.pop(...)` to be safe to call from multiple threads. This is okay, because Python guarantees it.

7.4.1 The main thread

Even though it was mentioned in passing above, it makes sense to again describe the main thread explicitly. Every program starts with one thread. For example, consider the Python program consisting of the single line:

```python
print("Hello World")
```

The thread that executes this line is the main thread. It isn't special in any way, except that it's the first (and often only) thread started in a program.

In our Qt examples so far, the main thread has always been the one that executes the various `import` statements, up to the concluding `app.exec()` call. If you recall from Section 4.7.1, this last call makes Qt process events in a loop until the user closes the application.

While the main thread is "just a thread", Qt and some operating system do impose constraints on its usage. In particular, Qt requires that you only create `QApplications` in the main thread, and only call their `.exec()` method there.

As we saw above, the fact that GUI changes can only be made in the main thread requires us to communicate with it. The next section shows a more general way of doing this.

7.5   Coordinating threads with custom signals

Our `QTimer` approach above was pretty ad-hoc and required an intermediate list, `new_messages`, to pass data to the main thread. Here, we will develop a more general solution that you can also use in your own projects.

We will introduce a new function called `run_in_main_thread(...)`. It lets you execute any code in the main thread. Before we explain its implementation, here is how it lets us rewrite `fetch_new_messages()`:

```python
from threadutil import run_in_main_thread

append_message = run_in_main_thread(text_area.appendPlainText)

def fetch_new_messages():
    while True:
        response = server.get(chat_url).text
        if response:
            append_message(response)
        sleep(.5)
```

In words, we use `run_in_main_thread(...)` to create a function that adds messages to the text area *in the main thread*. Then we invoke this function in our implementation of `fetch_new_messages()`.

With these changes, `display_new_messages()` and the `QTimer` that invoked it become obsolete. As a result, the new code is shorter and easier to read. (You can find it in the file `03_with_threadutil.py` in the GitHub repository.) What's more, we can reuse `run_in_main_thread(...)` in other contexts.

The next subsection shows how `run_in_main_thread(...)` can be implemented. It starts off easy, but quickly becomes quite advanced. If you just want to use the function without having to understand how it works, copy the file `threadutil.py` from subdirectory 11 in the GitHub repository into your project.

Then, you can use it as above or as a Python decorator:

```python
@run_in_main_thread
def my_function():
    ...


class MyClass:
    @run_in_main_thread
    def my_method(self):
        ...
```

The implementation presented below is asynchronous. That is, you can only tell it "run this in the main thread", but you are not notified when this actually happens. For a more advanced implementation that allows this, look at the file `threadutil_blocking.py` in the GitHub repository.

7.5.1   Implementing run_in_main_thread(...)

We saw in Section 3.5 how you can connect to Qt's existing signals. Now we will create our own signals. They will help us invoke code in other threads.

Custom signals can only be defined in subclasses of `QObject`. This is the base class of all Qt objects, so you can actually use any Qt class. We don't need any special functionality however, so will simply use `QObject`.

Here is an example of a simple signal:

```python
from PyQt6.QtCore import pyqtSignal, QObject


class MyClass(QObject):
    signal = pyqtSignal()


instance = MyClass()
instance.signal.connect(lambda: print("Hi!"))
instance.signal.emit()
```

As described above, we create a new subclass of `QObject`. It contains a field of type `pyqtSignal`. We instantiate the class and connect this signal to a simple **lambda** function. Finally, we emit the signal. When you run the code, you will

see that this invokes the lambda function and prints the text Hi!.

Signals can have parameters. To see an example of this, change the code to:

```python
class MyClass(QObject):
    signal = pyqtSignal(int)


instance = MyClass()
instance.signal.connect(lambda x:  print(x * 2))
instance.signal.emit(7)
```

This declares that our signal has a single parameter of type int, i.e. an integer. We connect the signal to a function that actually accepts a parameter ("x"). Finally, we emit the signal by passing a value. The result is that the above code prints 14, which is seven times two.

An important feature of threading in Qt is that every QObject *lives in* a thread. Signals delivered to a QObject are executed in the thread it lives in. One can exploit this to run code in another thread: Simply create a QObject in that thread, and deliver signals to it.

The following class uses this trick:

```python
from PyQt6.QtCore import QObject, pyqtSignal


class CurrentThread(QObject):

    _on_execute = pyqtSignal(object)

    def __init__(self):
        super(QObject, self).__init__()
        self._on_execute.connect(self._execute_in_thread)

    def execute(self, f):
        self._on_execute.emit(f)

    def _execute_in_thread(self, f):
        f()
```

It defines the signal `_on_execute`. Similarly to before, this signal takes a parameter. However, this time it accepts any Python **object**, not just integers. The reason for this is that we want to be able to pass both functions and methods, and **object** is the only common superclass of these two types.

The class's constructor first invokes the **super** implementation. This gives `QObject`'s initialization code a chance to run. Then, it connects its own method `_execute_in_thread` to the signal.

If you are wondering about the leading underscores _ in the names: They are simply a Python convention to mark fields as private implementation details.

The `execute(...)` method (which is intented for outside use because it does not have a leading underscore) takes a function `f` as a parameter and simply forwards it to the `_on_execute` signal.

Finally, the `_execute_in_thread(...)` method executes the function it receives as a parameter. The trick is that this method will be called in the thread in which the `CurrentThread` instance lives!

Let's see how this works by means of a concrete example. Save the above class definition to a file called `threadutil.py`. Then start `python` and enter the following commands:

```python
from threading import *
print_thread = lambda: print(current_thread().name)
print_thread()
Thread(target=print_thread).start()
```

This should print something similar to the following:

```
MainThread
Thread-1
```

The reason for this is that the first call to `print_thread()` is performed in the main thread, while the second call happens in a separate thread.

But now consider what happens when you additionally enter the following:

```python
from PyQt6.QtWidgets import QApplication
```

```python
from threadutil import CurrentThread
app = QApplication([])
main_thr = CurrentThread()
Thread(target=lambda: main_thr.execute(print_thread)).start()
app.exec()
```

Here, we are invoking `main_thr.execute(print_thread)` in a new thread. And this prints `MainThread`! So we succeeded in making another thread execute code in the main thread.

The above implementation has a limitation: It can only run functions that do not take any parameters. Here is how we can extend it to support all functions:

```python
class CurrentThread(QObject):

    _on_execute = pyqtSignal(object, tuple, dict)

    ...

    def execute(self, f, args, kwargs):
        self._on_execute.emit(f, args, kwargs)

    def _execute_in_thread(self, f, args, kwargs):
        f(*args, **kwargs)
```

We are adding two parameters to the `_on_execute` signal. They contain the function's arguments. They're passed through to `_execute_in_thread(...)`, which then uses them to call the function. The syntax `f(*args, **kwargs)` is Python's default way of invoking a function with arbitrary parameters. If it is new to you, please google "python args kwargs".

Suppose we want to execute `print("Hi!")` in the main thread. Using the above implementation, this can now be done as follows:

```python
main_thread = CurrentThread()
# Possibly from another thread:
main_thread.execute(print, ("Hi!",), {})
```

We pass the single-element tuple `("Hi!",)` as the function's arguments, and

the empty dictionary {} as its keyword arguments. Our implementation translates this to `print`(*("Hi!",), **{}), which is equivalent to `print`("Hi!").

Now we can finally define `run_in_main_thread`:

```python
def run_in_main_thread(f):
    def result(*args, **kwargs):
        main_thread.execute(f, args, kwargs)
    return result
```

Technically, `run_in_main_thread` is a Python *decorator*: It takes a function and returns a new function. Without going into too much further detail, suffice it to say that it enables the usages shown on pages 74f.

7.6 Concluding remarks

We saw in this chapter how threads can help us make our GUI more responsive. Virtually every application beyond a certain complexity will need to use them at some point.

While threads are very powerful, they also come with a lot of subtleties. We experienced this when we could only add messages to the text area in the main thread, to avoid crashes from threads interfering with each other.

There are other issues that come with threads. One which we did not cover is error handling: If an unhandled exception occurs in the main thread, your program dies with an informative stack trace. This is not the case for other threads. fbs, described in Chapter 5, automatically works around this. But if you are not using fbs, keep in mind that you may not be seeing all errors.

Threads are one of several cases where both Qt and Python offer an implementation. We used Python's `threading.Thread` above. But Qt also offers a `QThread` class. When programming in Python, it is often easier to use its libraries. We will see this again when connecting to databases in Chapter 9.

To work with threads, you will often need primitives such as locks, events, semaphores or thread pools. Fortunately, here too Python's standard library has excellent implementations.

Chapter 8

Displaying data: Lists, tables and tree views

Many applications need to display data in the form of lists, tables or directory-style tree views. In this chapter, we will see how Qt solves this common task.

8.1 Qt's Model/View framework

Consider Example 12. It displays a *tree view* of the files and directories in your home directory. You can expand folders to drill down into their contents.
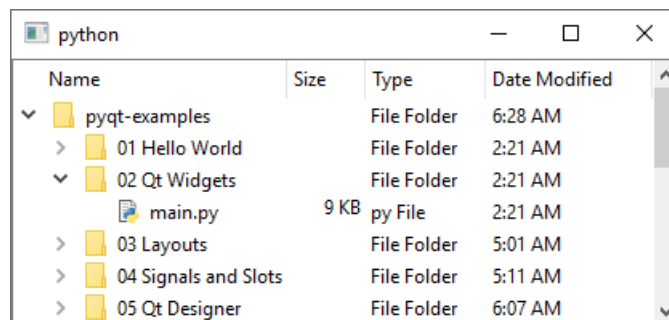
Example 12: A tree view of files and directories

The code for this example is:

```python
from os.path import expanduser
from PyQt6.QtWidgets import *
```

80

```
home_directory = expanduser("~")

app = QApplication([])
model = QDirModel()
view = QTreeView()
view.setModel(model)
view.setRootIndex(model.index(home_directory))
view.show()
app.exec()
```

Before we explain how this implementation works, let us see something cool. Change the above line
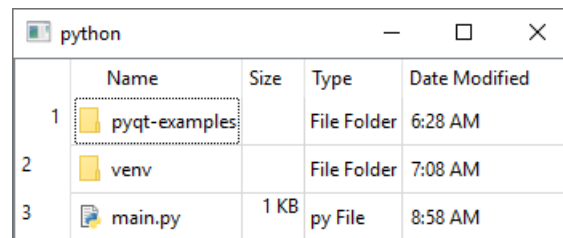
```
view = QTreeView()
```

to:

```
view = QTableView()
```

When you then run the code again, you get the *table* of files shown in Figure 8.1. This presentation no longer lets you drill down into subdirectories.

Figure 8.1: A table view of files

We can take this one step further. Change the line to:

```
view = QListView()
```

Then you only get a list (instead of a table) of files as shown in Figure 8.2.

This is pretty incredible! We changed nothing about how the code loads files. Yet, we get completely different appearances. The reason why this works is

Figure 8.2: A list view of files

that Qt's *Model/View* classes, which we used above, separate the logic for loading data from the logic that displays it.

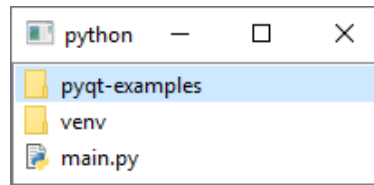So let us see how the rest of the code works. We first use `expanduser("~")` to get the path to your home directory. On Windows, this could for instance be `"C:\Users\Michael"`. Then we create a `QDirModel`. Next, we create a view and connect the model to it.

The next line `view.setRootIndex(...)` is a bit of a mystery. To understand it, we first need to know what an *index* is.

8.2   QModelIndex

Consider Example 12 again. As it turns out, the view doesn't actually know that it is displaying files. It only receives text and icons from the `model`. And information about the tree structure; Eg. that `main.py` lies inside `02 Qt Widgets/`.

The tree structure is represented via *model indices*. Every element in a model has an index consisting of a *row*, a *column* and a *parent*. For example, the size `9 KB` of `main.py` in Example 12 has `02 Qt Widgets/` as parent, row 0 (because it's the first row of its parent) and column 1 (because it's in the second column).

Parents in this context are themselves indices. So `02 Qt Widgets/` is an index that has `pyqt-examples/` as its parent, row 1 and column 0.

The parent structure gives rise to a classic chicken and egg problem: If every index has a parent, which is again an index with a parent, then is the list of "parents" never-ending?

The answer is no: Indices are represented by the `QModelIndex` class. You can invoke its `.parent()`, `.row()` and `.column()` methods to obtain the re-

spective fields. The trick is that `.parent()` is allowed to be the *invalid index* `QModelIndex()`, which indicates that the index does not have a parent.

The model and the view communicate solely using indices. Given its own root index X, the view asks the model a number of questions:

1. "How many rows are there in X?"
2. "How many columns?"
3. "Which text should I display in row 0, column 0 inside X?"
4. "Which text should I display in row 0, column 1 inside X?"

and so on until all rows and columns in X are displayed. Throughout, the view is agnostic of the kind of data it presents.

This lets us revisit our "mysterious" line:

```
view.setRootIndex(model.index(home_directory))
```

You may wonder if it is needed at all. As a matter of fact, it isn't: If you omit it, then the view's root index is `QModelIndex()` and `QDirModel` gives a list of your drives (eg. `C:/`, `D:/` ... on Windows).

But because we want to display the contents of the home directory, we do need to call `view.setRootIndex(...)` somehow. The question is: Where do we get the index which we can pass as an argument to this method?

As their name implies, model indices come from the model. Conveniently, `QDirModel` offers an `.index(...)` method that takes a file path as a parameter and returns a `QModelIndex` with a *row*, a *column* and a *parent*. We invoke this method and finally use its result to set the view's root index.

8.2.1   Indices in lists and tables

When we changed the view from a `QTreeView` to a `QTableView` and then a `QListView`, we kept the model the same. How do indices work in these cases?

The answer is that also these "flatter" presentations use `QModelIndex`. They simply ignore some of its parts. In the case of a table, the `parent` loses its relevance. And in a list, both the `parent` and the `column` have little importance.

8.3   Displaying different data roles

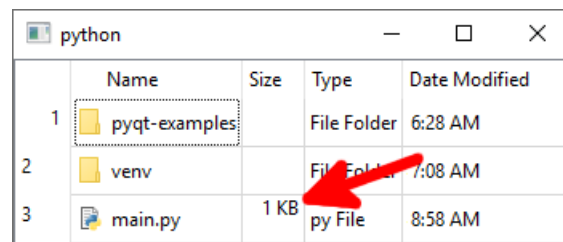Recall the following question, which the view asks the model when displaying a root index X:

Which text should I display in row 0, column 0 of X?

Besides *which text*, there are other questions the view can ask. For example: *Which color* should the cell have? *Which font*? *Which tooltip*, when the user hovers the mouse over it?

Qt supports these different pieces of information by means of *data roles*. Typical examples are `Qt.DisplayRole` for the cell's text, `Qt.FontRole` for its font, and `Qt.ForegroundRole` for its color. In these terms, the question "which text should I display" becomes "what is the DisplayRole for the index?"

Models return this information from their `data(...)` method. Its parameters are the `index` and the `role`, which indicates the kind of data to be queried.

Let's see this in practice. Did you notice in Figure 8.1 how the size `1 KB` of `main.py` was not correctly aligned with the rest of the row? It is reprinted in Figure 8.3 for your convenience.

Figure 8.3: Wrong alignment in QTableView

We can fix this by changing the model as follows:

```python
from os.path import expanduser
from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import *

home_directory = expanduser("~")
```
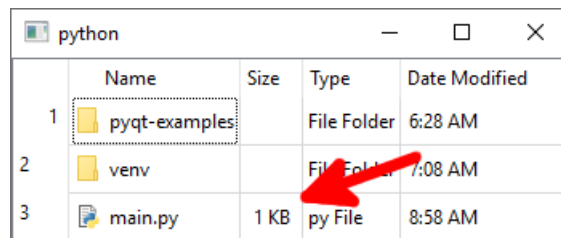
```python
class DirModel(QDirModel):
    def data(self, index, role):
        if role == Qt.TextAlignmentRole:
            return Qt.AlignVCenter
        return super(DirModel, self).data(index, role)

app = QApplication([])
model = DirModel()
view = QTableView()
view.setModel(model)
view.setRootIndex(model.index(home_directory))
view.show()
app.exec()
```

In words, we create a subclass of `QDirModel` and override `data(...)`. In its implementation, we return `AlignVCenter` whenever the `TextAlignmentRole` is requested. Otherwise, we invoke `QDirModel`'s default implementation.

The above code instructs the view to center text in cells vertically. This fixes the problem we set out to solve. The resulting screenshot is shown in Figure 8.4.

Figure 8.4: Correct alignment in QTableView

8.3.1   Headers

In addition to the `data(...)` method, models expose `headerData(...)`. This lets you change what's shown in the headers.

In the example above, *Date Modified* is a bit of a mouthful. You can change it to just `Modified` by adding the following method to the `DirModel` we had above:

```python
def headerData(self, section, orientation, role):
    if section == 3 and orientation == Qt.Horizontal \
        and role == Qt.DisplayRole:
        return "Modified"
    return super().headerData(section, orientation, role)
```

We check the `orientation` because we want to change the column header, not the row headers 1, 2, 3, ... The result is shown in Figure 8.5.

Figure 8.5: Changed column header in QTableView

8.4   Selections and the "current" index
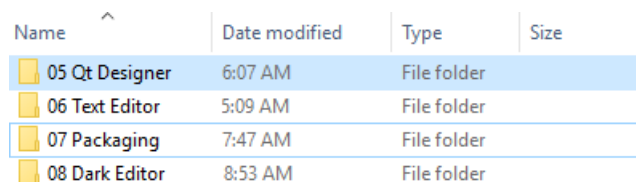
Compare the two screenshots in Figure 8.6. In the first, `main.py` is surrounded by a dotted line. In the second, it also has a blue background, clearly indicating that it is selected.

Figure 8.6: "Current" and selected cells in a QTableView

The screenshots exemplify two different concepts: The *current index* and the *selection*. You can think of the current index as a cursor that you move around the view with the arrow keys. It is always just a single cell. The selection on the other hand can contain arbitrarily many cells. In both screenshots, `main.py` is the current index. But in the second screenshot, it is also selected.

In most applications, and by default in Qt, changing the current index also updates the selection. Users can avoid this by navigating with $\boxed{\texttt{Ctrl}}$ plus the arrow keys $\boxed{\uparrow}$, $\boxed{\downarrow}$, ... This also works in Windows Explorer. See Figure 8.7.

Figure 8.7: Selection and current index in Windows Explorer

Technically, Qt stores both the current index and the selection in the class `QItemSelectionModel`. You can access it via `view.selectionModel()`. But Qt also exposes some convenience methods directly on the view, such as `view.selectAll()`, `view.currentIndex()` and `view.selectedIndices()`. The last one gives you the selected cells. You will probably use it the most.

You can customize how items are selected using view.setSelectionMode(...) and `view.setSelectionBehavior(...)`.

To always select entire rows instead of just a single cell, use:

`view.setSelectionBehavior(QAbstractItemView.SelectRows)`

To disable selections entirely, call:

`view.setSelectionMode(QAbstractItemView.NoSelection)`

There will still be a current index. To get a truly read-only presentation, execute:

`view.setFocusPolicy(Qt.NoFocus)`
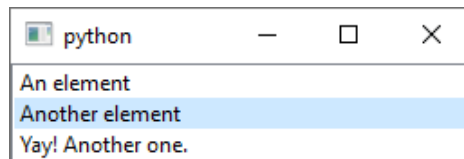
8.5 Other models

So far, the only source of data we have seen is `QDirModel`. But of course, there are others. The first one to mention is `QFileSystemModel`. It is almost identical to `QDirModel`, except that it loads files in the background and thus has better performance. The only reason we didn't use `QFileSystemModel` here is that it is a little bit more difficult to set up.

Another common model is `QStringListModel`. It lets you display a list of strings. For instance:

```
from PyQt6.QtWidgets import *
from PyQt6.QtCore import QStringListModel

app = QApplication([])
model = QStringListModel([
    "An element", "Another element", "Yay! Another one."
])
view = QListView()
view.setModel(model)
view.show()
app.exec()
```

The resulting screenshot is shown in Example 13.

Example 13: QStringListModel displayed in a QListView

We will see another model, which displays data from a database, in the next chapter. For now though, let us see how you can define your own models:

8.5.1 A custom model

Suppose you have the following data in Python:

```
headers = ["Scientist name", "Birthdate", "Contribution"]
rows = [("Newton", "1643-01-04", "Classical mechanics"),
        ("Einstein", "1879-03-14", "Relativity"),
        ("Darwin", "1809-02-12", "Evolution")]
```

To display it in Qt, we need to create a custom model. The most fitting format for this data is a table, so we will subclass `QAbstractTableModel`.

Displaying a table naturally requires answers to the following questions:

- How many rows are there?
- How many columns?
- What should be displayed in cell *x, y*?
- What are the row / column headers?

All our model does is answer these questions:

```python
class TableModel(QAbstractTableModel):
    def rowCount(self, parent):
        return len(rows)
    def columnCount(self, parent):
        return len(headers)
    def data(self, index, role):
        if role != Qt.DisplayRole:
            return QVariant()
        return rows[index.row()][index.column()]
    def headerData(self, section, orientation, role):
        if role != Qt.DisplayRole \
            or orientation != Qt.Horizontal:
            return QVariant()
        return headers[section]
```

If we then fill in the remaining code:
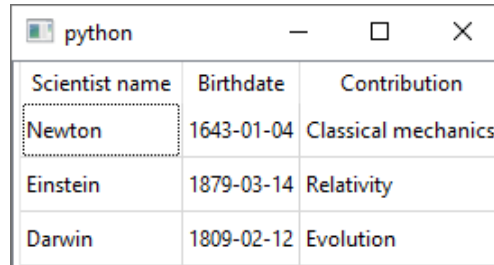
```python
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

headers = ... # as above
rows = ... # as above

class TableModel... # as above

app = QApplication([])
model = TableModel()
view = QTableView()
view.setModel(model)
view.show()
app.exec()
```

Then our data is displayed in a nice table. This is shown in Example 14.

Example 14: A custom QAbstractTableModel displayed in a QTableView

Of course, you can customize much more than just which data is displayed. You can also make the table editable for instance, and specify how data should be saved. For more information on this, please see the documentation of `QAbstractTableModel`.

Chapter 9

Databases

We have now seen plenty of ways in which data can be displayed. So let us look at a common way to obtain it: By connecting to a database.

The first database we will use in this chapter is SQLite. It is file-based and has built-in support in both Qt and Python. This makes it very easy to get started with. In Section 9.3, we will then see how you can adjust the examples presented here for other database systems such as PostgreSQL or MySQL.

As it turns out, initializing a database is slightly easier using Python's standard library than with Qt. This gives us a chance to explain Python's approach, before we later go on to using Qt's.

9.1 Accessing a database from Python

Most database libraries in Python conform to its Database API. This is a set of functions and methods that behave in the same way, irrespective of whether you're connecting to MySQL, PostgreSQL or any other database.

The Database API defines a `connect(...)` function. Its parameters depend on the respective database. In the case of SQLite, it simply requires a file name:

```python
import sqlite3
connection = sqlite3.connect("projects.db")
```

As the name of the variable implies, the function returns a `Connection` object.

91

The first thing you usually do is obtain a `Cursor` from it and execute some SQL. Here, we create a table and insert some data. To offer some insight that may be interesting for you, the data is my current monthly income in Euros (before expenses) from various projects.

```
cursor = connection.cursor()

cursor.execute("""
    CREATE TABLE projects
    (url TEXT, descr TEXT, income INTEGER)
""")

cursor.execute("""INSERT INTO projects VALUES
    ('bugfree-software.com', 'Test automation tools', 100),
    ('terminerinnerung.org', 'Appointment reminders', 1972),
    ('fman.io', 'File manager', 600),
    ('omaha-consulting.com', 'Update server hosting', 1200)
""")
```

After `cursor.execute(...)`, we can use the following to save the changes:

```
connection.commit()
```

When we later call `sqlite3.connect("projects.db")` again, the data above will still be there – provided `projects.db` has not been modified.

To read data from the database, we can use an ordinary SELECT statement:

```
cursor.execute("SELECT * from projects")
```

To actually fetch the data after this call, we can for instance use:

```
cursor.fetchall()
```

Not unexpectedly, this simply returns the rows we inserted.
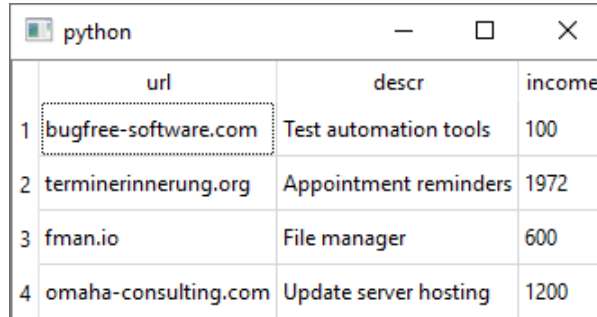
9.2 Connecting a database to Qt

Now that we have initialized the database, let us see how we can display its contents in Qt. Use the following code:

```python
from PyQt6.QtWidgets import *
from PyQt6.QtSql import *

app = QApplication([])
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName("projects.db")
db.open()
model = QSqlTableModel(None, db)
model.setTable("projects")
model.select()
view = QTableView()
view.setModel(model)
view.show()
app.exec()
```

The resulting table view is shown in Example 15. What's cool is that you can doubleclick on the cells to change their contents. Your modifications are automatically saved to the database.

Example 15: An editable database table

9.3   Connecting to other databases

Python's and Qt's APIs make it easy to use different databases interchangeably. For example, here is how you can switch to PostgreSQL:

First, set up PostgreSQL on your system. Then, install its Python library via:

```
pip install psycopg2
```

Next change the lines

```python
import sqlite3
connection = sqlite3.connect("projects.db")
```

to:

```python
import psycopg2
connection = psycopg2.connect(
    host="...", user="...", password="...", database="..."
)
```

and the lines

```python
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName("projects.db")
```

to:

```python
db = QSqlDatabase.addDatabase("QPSQL")
db.setHostName("...")
db.setUserName("...")
db.setPassword("...")
db.setDatabaseName("...")
```

(You need to fill in the appropriate values for your database. If you are running PostgreSQL on your own computer, the host name is likely `"localhost"`.)

That's all there is to it! With these changes, the rest of the examples in the previous sections work just like before.

Chapter 10

Final tips

10.1   Debugging performance

As your application matures, it is not unlikely that you will run into performance issues at some point. Here is a simple, yet very effective way to debug them.

Suppose your `main.py` script looks as follows:

```
from my_implementation import run_app
run_app()
```

Then you can very easily see where your code spends its time by using the following statements instead:

```
from my_implementation import run_app
import cProfile
cProfile.run("run_app()", sort="cumtime")
```

Once your app closes, this prints detailed statistics about how much time was spent in each function. This is invaluable for finding the root cause(s) of the problem. For more information, see the documentation of `cProfile`.

10.2   Object ownership

In Chapter 8, we used Qt's Model/View framework to display lists, tables etc. This framework also lets you implement your own drag and drop operations. For instance, you can let users drag the rows in a list.

To implement dragging in a model, one of the required steps is to override `mimeData(...)`:

```python
class MyList(QAbstractListModel):
    def mimeData(self, indexes):
        result = QMimeData()
        result.setUrls(...)
        # Note: Does not yet work. See below.
        return result
```

The `setUrls(...)` call lets you store information about whatever is being dragged. For example, if your list consists of files, then you would store a list of `file://` URLs. When the user drags a row from your list into the native file manager, the operating system then understands this data and copies the file.

As indicated by the source code comment above, this implementation does not yet work. The reason for this is the following: Qt's documentation of drag and drop says that "the `QMimeData` object should not be deleted – it will be destroyed by Qt". But consider what happens above: We create the `result` variable. When `mimeData(...)` returns, the variable goes out of scope and Python eventually deletes it. Qt later tries to delete it too – and we get a crash!

The solution is to adapt the above snippet as follows:

```python
import sip

class MyList(QAbstractListModel):
    def mimeData(self, indexes):
        result = QMimeData()
        result.setUrls(...)
        sip.transferto(result, None)
        return result
```

This essentially tells the Python interpreter not to destroy the object.

There are several places in Qt's API where ownership issues play a role. Fortunately, the Python bindings often take care of them for you automatically. However, the above shows that there are edge cases where you do need to understand what's going on.

10.3   Error handling and reporting

There are a few caveats when it comes to error handling in desktop applications in general, and in Python / Qt apps in particular.

The first is that unlike web applications, error messages that occur on your users' systems are not automatically sent to you. Services like Sentry let you track errors from your application in the cloud. This is invaluable for seeing which problems occur outside your test environment.

Another issue, mentioned in Section 7.6, is that errors that occur in background threads are often not as visible as those in the main thread. They only kill the thread, not your application, and in Python < 3.8 do not print stack traces. If your app is not working as expected and you are using threads, maybe there are some exceptions you are not being shown.

A final, and very subtle point is that both Python bindings currently cut off some stack traces: Let f and h be Python functions and g be a function of Qt. If

```
f() -> g() -> h()
```

(where "->" means "calls"), and an exception occurs in h(), then the traceback does not contain f. This can make debugging very difficult.

If you are using fbs as suggested in Chapter 5, then you get workarounds for the above issues out of the box. Otherwise however, you need to keep them in mind and perhaps create solutions of your own.

This concludes the book. I hope you have enjoyed it! May your errors be rare and their messages be ever informative ;-) Have fun writing your own apps!

Appendix A

Complying with the LGPL

Most of Qt is available under the open source LGPL license. This license permits *dynamic linking*, which is exactly what we are doing when we use Qt from Python. Because of this, projects based on Python and Qt typically do not need to buy a commercial Qt license. However, there are some obligations you do need to fulfill when using Qt under the LGPL.

The main goal of the LGPL is to let developers use LGPL-licensed libraries, while giving end users the freedom to change the library implementations. In the case of applications built with Python and Qt, this is very easy: They are typically shipped as an executable (`App.exe`, say) and shared libraries such as `Qt6Core.dll` and `Qt6Widgets.dll`. Users who want to change the Qt implementation can simply replace these DLLs. This means that we get the most important part of the LGPL "for free".

What remains are then some formal obligations of the LGPL: You need to provide users with the license texts of the GPL and the LGPL, as well as the source code of Qt. Furthermore, you need to indicate that Qt is used in your application. And finally, you need to include Qt's copyright notice.

A practical way of meeting these requirements is to include the following text in an "About" menu or a README file:

> This program uses version X.Y of the Qt Toolkit under the terms of the GNU Lesser General Public License version 3 (LGPLv3). The Qt Toolkit is Copyright (C) YEAR The Qt Company Ltd. To

obtain a copy of its source code, please go to https://qt.io or email us at <u>EMAIL</u>.

You may use, distribute and copy the Qt Toolkit under the terms of the LGPLv3, which is displayed below. This license makes reference to the GNU General Public License, which you can also find below.

GNU LESSER GENERAL PUBLIC LICENSE
<u>Version 3, 29 June 2007</u>
... text of the LGPL ...

GNU GENERAL PUBLIC LICENSE
<u>Version 3, 29 June 2007</u>
... text of the GPL ...

Please replace the underlined values above by your own. You can find Qt's copyright notice in its source code. There, you can also check that it still uses LGPL version 3.

Finally, please note that the above information was compiled by a layman and does not represent legal advice. To make sure that you are in compliance with Qt's licensing terms, you should consult a lawyer and/or The Qt Company.

Appendix B

PySide / Qt for Python

As mentioned in Section 2.2, using Qt from Python requires so-called *Python bindings*. We have used PyQt in this book. Another alternative is PySide.

PySide is the official Python binding for Qt since 2018. It is supported and maintained by The Qt Company. PySide is a *part of* Qt for Python, the umbrella term which The Qt Company use for all their Python-related projects. However, when reading online, you will often find the two names used interchangeably.

PySide was originally released in 2009 by Nokia, then owner of Qt. After Nokia sold Qt in 2011, development of PySide stalled. PySide2 was a community effort to maintain it. Finally, in 2016, the Qt company committed to officially supporting this project. It released PySide6 in 2020.

Unlike PyQt, PySide is available under the LGPL. This lets you use it for free in proprietary applications (similarly to Qt – see Appendix A). The reason we didn't use it in this book is that it is not yet as mature as its commercial rival.

B.1   Choosing between PyQt and PySide

As an experiment, I once tried to migrate the 20 kLoC code base of my file manager fman from PyQt to PySide. It only took 30 minutes to obtain a running version. Almost all changes simply replaced import statements such as

```python
from PyQt6.QtWidgets import *
```

by

```
from PySide6.QtWidgets import *
```

From an API perspective, it thus really doesn't matter which binding you use. Use PyQt if you want something mature and stable and don't mind the GPL (or buying a commercial license). Otherwise, use PySide.

Because of these similarities, you can also use PySide for the examples in this book. Simply replace `PyQt6` by `PySide6` in the code and you're good to go.

Appendix C

License text for Example 6

The example in Section 3.7 is based on code and images that are open source under the license below. The code and images are:

Copyright (c) 2012-2014, Juergen Bocklage Ryannel and Johan Thelin
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holder or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.