Create Simple GUI Applications *with* Python & Qt5

The hands-on guide to making desktop apps with Python

Martin Fitzpatrick

Create Simple GUI Applications, with Python & Qt5

The hands-on guide to building desktop apps with Python.

Martin Fitzpatrick

This book is for sale at http://leanpub.com/create-simple-gui-applications

This version was published on 2019-08-11

Tweet This Book!

Please help Martin Fitzpatrick by spreading the word about this book on Twitter!

The suggested hashtag for this book is #createsimpleguis.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#createsimpleguis

Contents

Introduction

Welcome to *Create Simple GUI Applications* the practical guide to building professional desktop applications with Python & Qt.

If you want to learn how to write GUI applications it can be pretty tricky to get started. There are a lot of new concepts you need to understand to get *anything* to work. A lot of tutorials offer nothing but short code snippets without any explanation of the underlying systems and how they work together. But, like any code, writing GUI applications requires you to learn to think about the problem in the right way.

In this book I will give you the real useful basics that you need to get building functional applications with the PyQt framework. I'll include explanations, diagrams, walkthroughs and code to make sure you know what you're doing every step of the way. In no time at all you will have a fully functional Qt application - ready to customise as you like.

The source code for each step is included, but don't just copy and paste and move on. You will learn much more if you experiment along the way!

So, let's get started!

Book format

This book is formatted as a series of coding exercises and snippets to allow you to gradually explore and learn the details of PyQt5. However, it is not possible to give you a *complete* overview of the Qt system in a book of this size (it's huge, this isn't), so you are encouraged to experiment and explore along the way.

If you find yourself thinking "I wonder if I can do *that*" the best thing you can do is put this book down, then *go and find out!* Just keep regular backups of your code along the way so you always have something to come back to if you royally mess it up.

Throughout this books there are also boxes like this, giving info, tips and warnings. All of them can be safely skipped over if you are in a hurry, but reading them will give you a deeper and more rounded knowledge of the Qt framework.

Qt and PyQt

When you write applications using PyQt what you area *really* doing is writing applications in Qt. The PyQt library is simply[1] a wrapper around the C++ Qt library, to allow it to be used in Python.

Because this is a Python interface to a C++ library the naming conventions used within PyQt do not adhere to PEP8 standards. Most notably functions and variables are named using `mixedCase` rather than `snake_case`. Whether you adhere to this standard in your own applications based on PyQt is entirely up to you, however you may find it useful to help clarify where the PyQt code ends and your own begins.

Further, while there is PyQt specific documentation available, you will often find yourself reading the Qt documentation itself as it is more complete. If you do you will need to translate object syntax and some methods containing Python-reserved function names as follows:

Qt	PyQt
`Qt::SomeValue`	`Qt.SomeValue`
`object.exec()`	`object.exec_()`
`object.print()`	`object.print_()`

Python 3

This book is written to be compatible with Python 3.4+. Python 3 is the future of the language, and if you're starting out now is where you should be focusing your efforts. However, in recognition of the fact that many people are stuck supporting or developing on legacy systems, the examples and code used in this book are also tested and confirmed to work on Python 2.7. Any notable incompatibility or

[1] Not really *that* simple.

gotchas will be flagged with a meh-face to accurately portray the sentiment e.g.

Python 2.7

In Python 2.7 `map()` returns a `list`.

If you are using Python 3 you can safely ignore their indifferent gaze.

Getting Started

Before you start coding you will first need to have a working installation of PyQt and Qt on your system. The following sections will guide you through this process for the main available platforms. If you already have a working installation of PyQt on your Python system you can safely skip this part and get straight onto the fun.

The complete source code all examples in this book is available to download from here.

GPL Only

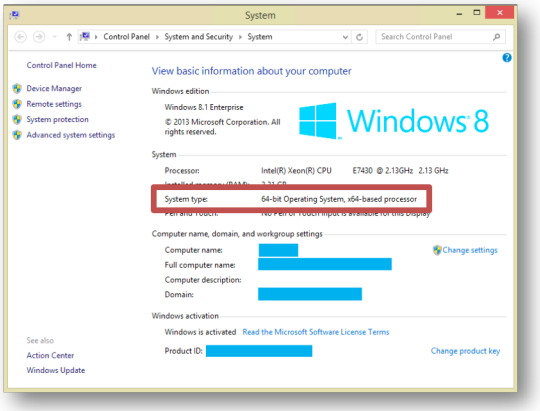
Note that the following instructions are **only** for installation of the GPL licensed version of PyQt. If you need to use PyQt in a non-GPL project you will need to purchase an alternative license from Riverbank Computing in order to release your software.

Documentation?

The PyQt packages from Riverbank do not include the Qt documentation. However this is available online at docs.qt.io. If you *do* want to download the documentation you can do so from www.qt.io.

Installation Windows

PyQt5 for Windows can be installed as for any other application or library. The only slight complication is that you must first determine whether your system supports 32bit or 64bit software. You can determine whether your system supports 32bit or 64bit by looking at the System panel accessible from the control panel.



The Windows system panel, where you can find out if you're running 64 or 32bit.

If your system *does* support 64bit (and most modern systems do) then you should also check whether your current Python install is 32 or 64 bit. Open a command prompt (Start > `cmd`):

```
C:\> python3
```

Look at the top line of the Python output, where you should be able to see whether you have 32bit or 64bit Python installed. If you want to switch to 32bit or 64bit Python you should do so at this point.

PyQt5 for Python 3

A PyQt5 installer for Windows is available direct from the developer Riverbank Computing. Download the `.exe` files from the linked page, making sure you download the currently 64bit or 32bit version for your system. You can install this file as for any other Windows application/library.

After install is finished, you should be able to run `python` and `import PyQt5`.

PyQt5 for Python 2.7

Unfortunately, if you want to use PyQt5 on Python 2.7 there are no official installers available to do this. This part of a policy by Riverbank Computing to encourage transition to Python 3 and reduce their support burden.

However, there is nothing technically stopping PyQt5 being compiled for Python 2.7 and the helpful people at Abstract Factory have done exactly that.

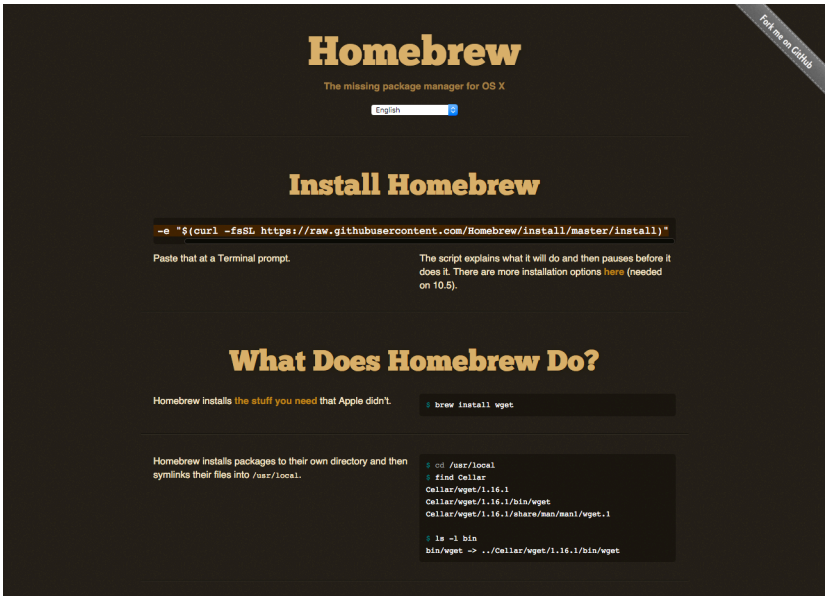
Simply download the above `.rar` file and unpack it with 7zip (or other unzip application). You can then copy the resulting folder to your Python site-packages folder — usually in `C:\Python27\lib\site-packages\`

Once that is done, you should be able to run `python` and `import PyQt5`.

Installation Mac

OS X comes with a pre-installed version of Python (2.7), however attempting to install PyQt5 into this is more trouble than it is worth. If you are planning to do a lot of Python development, and you should, it will be easier in the long run to create a distinct installation of Python separate from the system.

By far the simplest way to do this is to use Homebrew. Homebrew is a package manager for command-line software on MacOS X. Helpfully Homebrew also has a pre-built version of PyQt5 in their repositories.



Homebrew — the missing package manager for OS X

To install Homebrew run the following from the command line:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master\
/install)"
```

This is also available to copy and paste from the Homebrew homepage.

Once the Homebrew installation has completed, you can then install Python 3 and PyQt5 as follows:

```
brew install python3
brew install pyqt5 --with-python-3
```

After that has completed, you should be able to run `python3` and `import PyQt5`.

Installation Linux (Ubuntu)

Installation on Linux is very straightforward as packages for PyQt5 are available in the repositories of most distributions. In Ubuntu you can install either from the command line or via “Software Center”. The packages you are looking for are named `python3-pyqt5` or `python-pyqt5` depending on which version you are installing for.

You can also install these from the command line as follows:

```
apt-get install python3-pyqt5
```

Or for Python 2.7:

```
apt-get install python-pyqt5
```

Once the installation is finished, you should be able to run `python3` or `python` and `import PyQt5`.

Basic Qt Features

Welcome to your first steps in create graphical applications! In this chapter you will be introduced to the key basic features of Qt (PyQt) that you will find yourself using in any applications you create. We will develop a series of small applications, adding (and removing!) features step-by-step. Use the code given as your guide, and feel free to experiment around it — particularly with reference to the Qt Documentation.

My first Window

So, let's create our very first windowed application. Before getting the window on the screen, there are a few key concepts to introduce about how applications are organised in the Qt world. If you're already familiar with event loops you can safely skip to the next section.

The Event loop and `QApplication`

The core of every Qt Applications is the `QApplication` class. Every application needs one — and only one — `QApplication` object to function. This object holds the *event loop* of your application — the core loop which governs all user interaction with the GUI.

Each interaction with your application — whether a press of a key, click of a mouse, or mouse movement — generates an *event* which is placed on the *event queue*. In the event loop, the queue is checked on each iteration and if a waiting event is found, the event and control is passed to the specific *event handler* for the event. The event handler deals with the event, then passes control back to the event loop to wait for more events. There is only *one* running event loop per application.

Key Points

- QApplication holds the Qt event loop
- One QApplication instance required
- You application sits waiting in the event loop until an action is taken
- There is only *one* event loop

Creating your App

To start build your application, create a new Python file — you can call it whatever you like (e.g. `MyApp.py`).

Backup!

We'll be editing within this file as we go along, and you may want to come back to earlier versions of your code, so remember to keep regular backups along the way. For example, after each section save a file named `MyApp_<section>.py`

The source code for your very first application is shown below. Type it in verbatim, and be careful not to make mistakes. If you do mess up, Python should let you know what's wrong when you run it. If you don't feel like typing it all in, you can download the source code.

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *

# Only needed for access to command line arguments
import sys

# You need one (and only one) QApplication instance per application.
# Pass in sys.argv to allow command line arguments for your app.
# If you know you won't use command line arguments QApplication([]) works too.
app = QApplication(sys.argv)

# Start the event loop.
app.exec_()


# Your application won't reach here until you exit and the event
# loop has stopped.
```

Let's go through the code line by line.

We start by importing the PyQt5 classes that we need for the application, from the `QtWidgets`, `QtGui` and `QtCore` submodules.

This kind of global import `from <module> import *` is generally frowned upon in Python. However, in this case we know that the PyQt classnames don't conflict with one another, or with Python itself. Importing them all saves a lot of typing, and helps with PyQt4 compatibility.

Next we create an instance of `QApplication`, passing in `sys.arg` (which contains command line arguments). This allows us to pass command line arguments to our application. If you know you won't be accepting command line arguments you can pass in an empty list instead, e.g.

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

Finally, we call `app.exec_()` to start up the event loop.

The underscore is there because `exec` is a reserved word in Python and can't be used as a function name. PyQt5 handles this by appending an underscore to the name used in the C++ library. You'll also see it for `.print_()`.

To launch your application, run it from the command line like any other Python script, for example:

```
python MyApp.py
```

Or, for Python 3:

```
python3 MyApp.py
```

The application should run without errors, yet there will be no indication of anything happening, aside from perhaps a busy indicator. This is completely normal — we haven't told Qt to create a window yet!

Every application needs at least one `QMainWindow`, though you can have more than one if you need to. However, no matter how many you have, your application will always exit when the last main window is closed.

Let's add a `QMainWindow` to our application.

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *

import sys


app = QApplication(sys.argv)

window = QMainWindow()
window.show() # IMPORTANT!!!!! Windows are hidden by default.

# Start the event loop.
app.exec_()
```

`QMainWindow`

- Main focus for user of your application
- Every application needs at least one (...but can have more)
- Application will exit when last main window is closed

If you launch the application you should now see your main window. Notice that Qt automatically creates a window with the normal window decorations, and you can drag it around and resize it like any normal window.

I can't see my window!

You must *always* call `.show()` on a newly created `QMainWindow` as they are created invisible by default.

Congratulations — you've created your first Qt application! It's not very interesting at the moment, so next we will add some content to the window.

If you want to create a custom window, the best approach is to subclass `QMainWindow` and then include the setup for the window in the `__init__` block. This allows the window behaviour to be self contained. In the next step we create our own subclass of `QMainWindow` — we can call it `MainWindow` to keep things simple.

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *

import sys


# Subclass QMainWindow to customise your application's main window
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")

        # The `Qt` namespace has a lot of attributes to customise
        # widgets. See: http://doc.qt.io/qt-5/qt.html
        label.setAlignment(Qt.AlignCenter)

        # Set the central widget of the Window. Widget will expand
        # to take up all the space in the window by default.
        self.setCentralWidget(label)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec_()
```

Notice how we write the `__init__` block with a small bit of boilerplate to take the arguments (none currently) and pass them up to the `__init__` of the parent `QMainWindow` class.

When you subclass a Qt class you must *always* call the super `__init__`-function to allow Qt to set up the object.

Next we use `.setWindowTitle()` to change the title of our main window.

Then we add our first widget — a `QLabel` — to the middle of the window. This is one of the simplest widgets available in Qt. You create the object by passing in the text that you want the widget to display.

We set the alignment of the widget to the centre, so it will show up in the middle of the window.

The Qt namespace (`Qt.`) is full of all sorts of attributes that you can use to customise and control Qt widgets. We'll cover that a bit more later, it's worth a look.

Finally, we call `.setCentralWidget()` on the the window. This is a `QMainWindow` specific function that allows you to set the widget that goes in the middle of the window.

If you launch your application you should see your window again, but this time with the `QLabel` widget in the middle.

Hungry for widgets?

We'll cover more widgets in detail shortly but if you're impatient and would like to jump ahead you can take a look at the QWidget documentation. Try adding the different widgets to your window!

In this section we've covered the `QApplication` class, the `QMainWindow` class, the event loop and experimented with adding a simple widget to a window. In the next section we'll take a look at the mechanisms Qt provides for widgets and windows to communicate with one another and your own code.

Save a copy of your file as `MyApp_window.py` as we'll need it again later.

Signals, Slots, Events

As already described, every interaction the user has with a Qt application causes an Event. There are multiple types of event, each representing a difference type of interaction — e.g. mouse or keyboard events.

Events that occur are passed to the event-specific handler on the widget where the interaction occurred. For example, clicking on a widget will cause a `QMouseEvent` to be sent to the `.mousePressEvent` event handler on the widget. This handler can interrogate the event to find out information, such as what triggered the event and where specifically it occurred.

You can intercept events by subclassing and overriding the handler function on the class, as you would for any other function. You can choose to filter, modify, or ignore events, passing them through to the normal handler for the event by calling the parent class function with `super()`.

```
    class CustomButton(Qbutton):

        def keyPressEvent(self, e):
            # My custom event handling
            super(CustomButton, self).keyPressEvent(e)
```

However, imagine you want to catch an event on 20 different buttons. Subclassing like this now becomes an incredibly tedious way of catching, interpreting and handling these events.

```
    class CustomButton99(Qbutton)

        def keyPressEvent(self, e):
            # My custom event handling
            super(CustomButton99, self).keyPressEvent(e)
```

Thankfully Qt offers a neater approach to receiving notification of things happening in your application: *Signals*.

Signals

Instead of intercepting raw events, signals allow you to 'listen' for notifications of specific occurrences within your application. While these can be similar to events — a click on a button — they can also be more nuanced — updated text in a box. Data can also be sent alongside a signal - so as well as being notified of the updated text you can also receive it.

The receivers of signals are called *Slots* in Qt terminology. A number of standard slots are provided on Qt classes to allow you to wire together different parts of your application. However, you can also use any Python function as a slot, and therefore receive the message yourself.

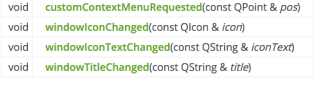
Load up a fresh copy of `MyApp_window.py` and save it under a new name for this section.

Basic signals

First, let's look at the signals available for our `QMainWindow`. You can find this information in the Qt documentation. Scroll down to the Signals section to see the signals implemented for this class.

Qt 5 Documentation — QMainWindow Signals

As you can see, alongside the two `QMainWindow` signals, there are 4 signals inherited from `QWidget` and 2 signals inherited from `Object`. If you click through to the `QWidget` signal documentation you can see a `.windowTitleChanged` signal implemented here. Next we'll demonstrate that signal within our application.



Qt 5 Documentation — Widget Signals

The code below gives a few examples of using the `windowTitleChanged` signal.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        # SIGNAL: The connected function will be called whenever the window
        # title is changed. The new title will be passed to the function.
        self.windowTitleChanged.connect(self.onWindowTitleChange)

        # SIGNAL: The connected function will be called whenever the window
        # title is changed. The new title is discarded in the lambda and the
        # function is called without parameters.
        self.windowTitleChanged.connect(lambda x: self.my_custom_fn())

        # SIGNAL: The connected function will be called whenever the window
        # title is changed. The new title is passed to the function
        # and replaces the default parameter
        self.windowTitleChanged.connect(lambda x: self.my_custom_fn(x))

        # SIGNAL: The connected function will be called whenever the window
        # title is changed. The new title is passed to the function
        # and replaces the default parameter. Extra data is passed from
        # within the lambda.
        self.windowTitleChanged.connect(lambda x: self.my_custom_fn(x, 25))

        # This sets the window title which will trigger all the above signals
        # sending the new title to the attached functions or lambdas as the
        # first parameter.
        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)


```

```
37      # SLOT: This accepts a string, e.g. the window title, and prints it
38      def onWindowTitleChange(self, s):
39          print(s)
40
41      # SLOT: This has default parameters and can be called without a value
42      def my_custom_fn(self, a="HELLLO!", b=5):
43          print(a, b)
```

Try commenting out the different signals and seeing the effect on what the slot prints.

We start by creating a function that will behave as a 'slot' for our signals.

Then we use .connect on the `.windowTitleChanged` signal. We pass the function that we want to be called with the signal data. In this case the signal sends a string, containing the new window title.

If we run that, we see that we receive the notification that the window title has changed.

Events

Next, let's take a quick look at events. Thanks to signals, for most purposes you can happily avoid using events in Qt, but it's important to understand how they work for when they are necessary.

As an example, we're going to intercept the `.contextMenuEvent` on `QMainWindow`. This event is fired whenever a context menu is *about to be* shown, and is passed a single value `event` of type `QContextMenuEvent`.

To intercept the event, we simply override the object method with our new method of the same name. So in this case we can create a method on our `MainWindow` subclass with the name `contextMenuEvent` and it will receive all events of this type.

```
1  def contextMenuEvent(self, event):
2      print("Context menu event!")
```

If you add the above method to your `MainWindow` class and run your program you will discover that right-clicking in your window now displays the message in the print statement.

Sometimes you may wish to intercept an event, yet still trigger the default (parent) event handler. You can do this by calling the event handler on the parent class using `super` as normal for Python class methods.

```
1  def contextMenuEvent(self, event):
2      print("Context menu event!")
3      super(MainWindow, self).contextMenuEvent(event)
```

This allows you to propagate events up the object hierarchy, handling only those parts of an event handler that you wish.

However, in Qt there is another type of event hierarchy, constructed around the UI relationships. Widgets that are added to a layout, within another widget, may opt to pass their events to their UI parent. In complex widgets with multiple sub-elements this can allow for delegation of event handling to the containing widget for certain events.

However, if you have dealt with an event and do not want it to propagate in this way you can flag this by calling `.accept()` on the event.

```
class CustomButton(Qbutton):

    def event(self, e):
        e.accept()
```

Alternatively, if you do want it to propagate calling `.ignore()` will achieve this.

```
class CustomButton(Qbutton):
    def event(self, e):
        e.ignore()
```

In this section we've covered signals, slots and events. We've demonstrated some simple signals, including how to pass less and more data using lambdas. We've created custom signals, and shown how to intercept events, pass on event handling and use `.accept()` and `.ignore()` to hide/show events to the UI-parent widget. In the next section we will go on to take a look at two common features of the GUI — toolbars and menus.

Actions, Toolbars and Menus

Next we'll look at some of the common user interface elements, that you've probably seen in many other applications — toolbars and menus. We'll also explore the neat system Qt provides for minimising the duplication between different UI areas — `QAction`.

Toolbars

One of the most commonly seen user interface elements is the toolbar. Toolbars are bars of icons and/or text used to perform common tasks within an application, for which accessing via a menu would be cumbersome. They are one of the most common UI features seen in many applications. While some complex applications, particularly in the Microsoft Office suite, have migrated to contextual 'ribbon' interfaces, the standard toolbar is usually sufficient for the majority of applications you will create.

Standard GUI elements - The toolbar

Qt toolbars support display of icons, text, and can also contain any standard Qt widget. However, for buttons the best approach is to make use of the `QAction` system to place buttons on the toolbar.

Let's start by adding a toolbar to our application.

Load up a fresh copy of `MyApp_window.py` and save it under a new name for this section.

In Qt toolbars are created from the QToolBar class. To start you create an instance of the class and then call `.addToolbar` on the QMainWindow. Passing a string in as the first parameter to `QToolBar` sets the toolbar's name, which will be used to identify the toolbar in the UI.

<<(code/toolbars_and_menus_1.py)

Run it!

You'll see a thin grey bar at the top of the window. This is your toolbar. Right click and click the name to toggle it off.

I can't get my toolbar back!?

Unfortunately once you remove a toolbar there is now no place to right click to re-add it. So as a general rule you want to either keep one toolbar unremoveable, or provide an alternative interface to turn toolbars on and off.

We should make the toolbar a bit more interesting. We could just add a `QButton` widget, but there is a better approach in Qt that gets you some cool features — and that is via `QAction`. `QAction` is a class that provides a way to describe abstract user interfaces. What this means in English, is that you can define multiple interface elements within a single object, unified by the effect that interacting with that element has. For example, it is common to have functions that are represented in the toolbar but also the menu — think of something like Edit->Cut which is present both in the Edit menu but also on the toolbar as a pair of scissors, and also through the keyboard shortcut `Ctrl-X` (`Cmd-X` on Mac).

Without `QAction` you would have to define this in multiple places. But with `QAction` you can define a single `QAction`, defining the triggered action, and then add this action to both the menu and the toolbar. Each `QAction` has names, status messages, icons and signals that you can connect to (and much more).

In the code below you can see this first `QAction` added.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)

        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        toolbar.addAction(button_action)


```

```
23      def onMyToolBarButtonClick(self, s):
24          print("click", s)
```

To start with we create the function that will accept the signal from the QAction so we can see if it is working. Next we define the `QAction` itself. When creating the instance we can pass a label for the action and/or an icon. You must also pass in any `QObject` to act as the parent for the action — here we're passing `self` as a reference to our main window. Strangely for `QAction` the parent element is passed in as the final parameter.

Next, we can opt to set a status tip — this text will be displayed on the status bar once we have one. Finally we connect the `.triggered` signal to the custom function. This signal will fire whenever the `QAction` is *triggered* (or activated).

Run it!

You should see your button with the label that you have defined. Click on it and the our custom function will emit "click" and the status of the button.

Why is the signal always false?

The signal passed indicates whether the button is *checked*, and since our button is not checkable — just clickable — it is always false. We'll show how to make it checkable shortly.

Next we can add a status bar.

We create a status bar object by calling `QStatusBar` to get a new status bar object and then passing this into `.setStatusBar`. Since we don't need to change the statusBar settings we can also just pass it in as we create it, in a single line:

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)

        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))


    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

Run it!

Hover your mouse over the toolbar button and you will see the status text in the status bar.

Next we're going to turn our `QAction` toggleable — so clicking will turn it on, clicking again will turn it off. To do this, we simple call `setCheckable(True)` on the `QAction` object.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)

        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))


    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

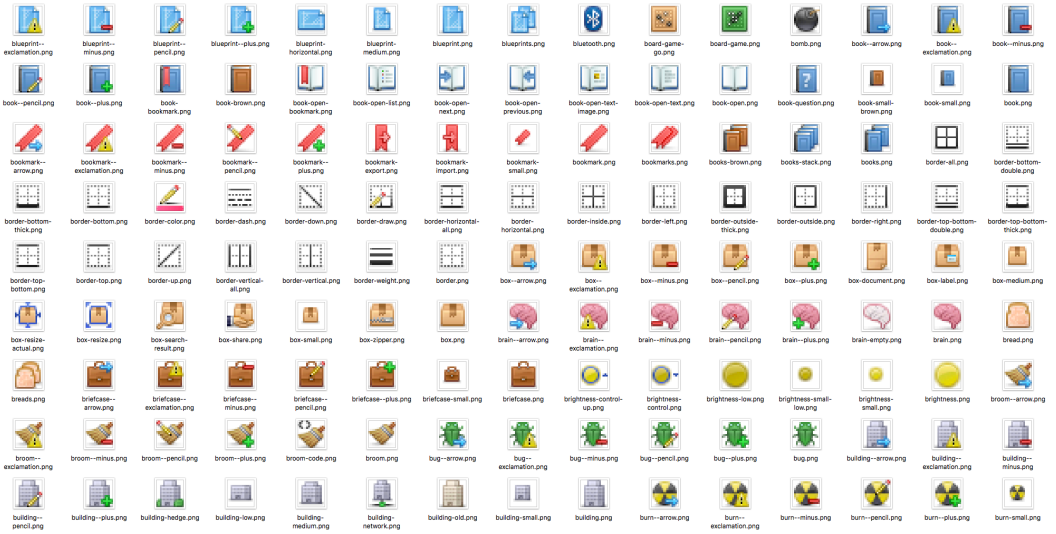
Run it!

Click on the button to see it toggle from checked to unchecked state. Note that custom slot function we create now alternates outputting `True` and `False`.

`.toggled`

There is also a `.toggled` signal, which only emits a signal when the button is toggled. But the effect is identical so it is mostly pointless.

Things look pretty shabby right now — so let's add an icon to our button. For this I recommend you download the fugue icon set by designer Yusuke Kamiyamane. It's a great set of beautiful 16x16 icons that can give your apps a nice professional look. It is freely available with only attribution required when you distribute your application — although I am sure the designer would appreciate some cash too if you have some spare.



Fugue Icon Set — Yusuke Kamiyamane

Select an image from the set (in the examples here I've selected the file `bug.png`) and copy it into the same folder as your source code. To add the icon to the `QAction` (and therefore the button) we simply pass it in as the first parameter when creating the `QAction`. If the icon is in the same folder as your source code you can just copy it to

You also need to let the toolbar know how large your icons are, otherwise your icon will be surrounded by a lot of padding. You can do this by calling `.setIconSize()` with a `QSize` object.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16,16))
        self.addToolBar(toolbar)

        button_action = QAction(QIcon("bug.png"), "Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))


    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

Run it!

The QAction is now represented by an icon. Everything should function exactly as it did before.

Note that Qt uses your operating system default settings to determine whether to show an icon, text or an icon and text in the toolbar. But you can override this by using `.setToolButtonStyle`. This slot accepts any of the following flags from the `Qt.` namespace:

Flag	Behaviour
`Qt.ToolButtonIconOnly`	Icon only, no text
`Qt.ToolButtonTextOnly`	Text only, no icon
`Qt.ToolButtonTextBesideIcon`	Icon and text, with text beside the icon
`Qt.ToolButtonTextUnderIcon`	Icon and text, with text under the icon
`Qt.ToolButtonIconOnly`	Icon only, no text
`Qt.ToolButtonFollowStyle`	Follow the host desktop style

Which style should I use?

The default value is `Qt.ToolButtonFollowStyle`, meaning that your application will default to following the standard/global setting for the desktop on which the application runs. This is generally recommended to make your application feel as *native* as possible.

Finally, we can add a few more bits and bobs to the toolbar. We'll add a second button and a checkbox widget. As mentioned you can literally put any widget in here, so feel free to go crazy. Don't worry about the `QCheckBox` type, we'll cover that later.

```
1  class MainWindow(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(MainWindow, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle("My Awesome App")
7
8          label = QLabel("THIS IS AWESOME!!!")
9          label.setAlignment(Qt.AlignCenter)
10
11         self.setCentralWidget(label)
12
13         toolbar = QToolBar("My main toolbar")
14         toolbar.setIconSize(QSize(16,16))
15         self.addToolBar(toolbar)
16
17         button_action = QAction(QIcon("bug.png"), "Your button", self)
```

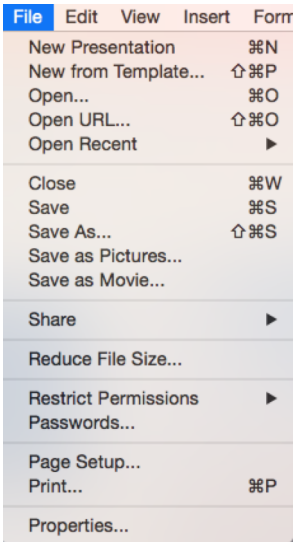
```
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addSeparator()

        button_action2 = QAction(QIcon("bug.png"), "Your button2", self)
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))


    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

Run it!

Now you see multiple buttons and a checkbox.

Menus

Menus are another standard component of UIS. Typically they are on the top of the window, or the top of a screen on a Mac. They allow access to all standard application functions. A few standard menus exist — for example File, Edit, Help. Menus can be nested to create hierarchical trees of functions and they often support and display keyboard shortcuts for fast access to their functions.



Standard GUI elements - Menus

To create a menu, we create a menubar we call `.menuBar()` on the QMainWindow. We add a menu on our menu bar by calling `.addMenu()`, passing in the name of the menu. I've called it `'&File'`. The ampersand defines a quick key to jump to this menu when pressing Alt.

```
1  class MainWindow(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(MainWindow, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle("My Awesome App")
7
8          label = QLabel("THIS IS AWESOME!!!")
9          label.setAlignment(Qt.AlignCenter)
10
11         self.setCentralWidget(label)
12
13         toolbar = QToolBar("My main toolbar")
14         toolbar.setIconSize(QSize(16,16))
15         self.addToolBar(toolbar)
16
```

```
17          button_action = QAction(QIcon("bug.png"), "&Your button", self)
18          button_action.setStatusTip("This is your button")
19          button_action.triggered.connect(self.onMyToolBarButtonClick)
20          button_action.setCheckable(True)
21          toolbar.addAction(button_action)
22
23          toolbar.addSeparator()
24
25          button_action2 = QAction(QIcon("bug.png"), "Your &button2", self)
26          button_action2.setStatusTip("This is your button2")
27          button_action2.triggered.connect(self.onMyToolBarButtonClick)
28          button_action2.setCheckable(True)
29          toolbar.addAction(button_action)
30
31          toolbar.addWidget(QLabel("Hello"))
32          toolbar.addWidget(QCheckBox())
33
34          self.setStatusBar(QStatusBar(self))
35
36          menu = self.menuBar()
37
38          file_menu = menu.addMenu("&File")
39          file_menu.addAction(button_action)
40
41
42      def onMyToolBarButtonClick(self, s):
43          print("click", s)
```

Quick Keys on Mac

This won't be visible on Mac. Note that this is different to a keyboard shortcut — we'll cover that shortly.

Next we add something to menu. This is where the power of `QAction` comes in to play. We can reuse the already existing `QAction` to add the same function to the menu. Click it and you will notice that it is toggleable — it inherits the features of the `QAction`.

Now let's add some more things to the menu. Here we'll add a separator to the menu, which will appear as a horizontal line in the menu, and then add the second `QAction` we created.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16,16))
        self.addToolBar(toolbar)

        button_action = QAction(QIcon("bug.png"), "&Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addSeparator()

        button_action2 = QAction(QIcon("bug.png"), "Your &button2", self)
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))
```

```
35
36          menu = self.menuBar()
37
38          file_menu = menu.addMenu("&File")
39          file_menu.addAction(button_action)
40          file_menu.addSeparator()
41          file_menu.addAction(button_action2)
42
43
44      def onMyToolBarButtonClick(self, s):
45          print("click", s)
```

Run it!

You should see two menu items with a line between them.

You can also use ampersand to add accelerator keys to the menu to allow a single key to be used to jump to a menu item when it is open. Again this doesn't work on Mac.

To add a submenu, you simply create a new menu by calling `addMenu()` on the parent menu. You can then add actions to it as normal. For example:

```
1   class MainWindow(QMainWindow):
2
3       def __init__(self, *args, **kwargs):
4           super(MainWindow, self).__init__(*args, **kwargs)
5
6           self.setWindowTitle("My Awesome App")
7
8           label = QLabel("THIS IS AWESOME!!!")
9           label.setAlignment(Qt.AlignCenter)
10
11          self.setCentralWidget(label)
12
13          toolbar = QToolBar("My main toolbar")
14          toolbar.setIconSize(QSize(16,16))
```

```
15          self.addToolBar(toolbar)
16
17          button_action = QAction(QIcon("bug.png"), "&Your button", self)
18          button_action.setStatusTip("This is your button")
19          button_action.triggered.connect(self.onMyToolBarButtonClick)
20          button_action.setCheckable(True)
21          toolbar.addAction(button_action)
22
23          toolbar.addSeparator()
24
25          button_action2 = QAction(QIcon("bug.png"), "Your &button2", self)
26          button_action2.setStatusTip("This is your button2")
27          button_action2.triggered.connect(self.onMyToolBarButtonClick)
28          button_action2.setCheckable(True)
29          toolbar.addAction(button_action)
30
31          toolbar.addWidget(QLabel("Hello"))
32          toolbar.addWidget(QCheckBox())
33
34          self.setStatusBar(QStatusBar(self))
35
36          menu = self.menuBar()
37
38          file_menu = menu.addMenu("&File")
39          file_menu.addAction(button_action)
40          file_menu.addSeparator()
41
42          file_submenu = file_menu.addMenu("Submenu")
43          file_submenu.addAction(button_action2)
44
45
46      def onMyToolBarButtonClick(self, s):
47          print("click", s)
```

Finally we'll add a keyboard shortcut to the `QAction`. You define a keyboard shortcut by passing setKeySequence() and passing in the key sequence. Any defined key sequences will appear in the menu.

Hidden shortcuts

Note that the keyboard shortcut is associated with the `QAction` and will still work whether or not the `QAction` is added to a menu or a toolbar.

Key sequences can be defined in multiple ways - either by passing as text, using key names from the Qt namespace, or using the defined key sequences from the Qt namespace. Use the latter wherever you can to ensure compliance with the operating system standards.

The completed code, showing the toolbar buttons and menus is shown below.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        label = QLabel("THIS IS AWESOME!!!")
        label.setAlignment(Qt.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16,16))
        self.addToolBar(toolbar)

        button_action = QAction(QIcon("bug.png"), "&Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        # You can enter keyboard shortcuts using key names (e.g. Ctrl+p)
        # Qt.namespace identifiers (e.g. Qt.CTRL + Qt.Key_P)
        # or system agnostic identifiers (e.g. QKeySequence.Print)
        button_action.setShortcut( QKeySequence("Ctrl+p") )

        toolbar.addAction(button_action)

```

```
28          toolbar.addSeparator()
29
30          button_action2 = QAction(QIcon("bug.png"), "Your &button2", self)
31          button_action2.setStatusTip("This is your button2")
32          button_action2.triggered.connect(self.onMyToolBarButtonClick)
33          button_action2.setCheckable(True)
34          toolbar.addAction(button_action)
35
36          toolbar.addWidget(QLabel("Hello"))
37          toolbar.addWidget(QCheckBox())
38
39          self.setStatusBar(QStatusBar(self))
40
41          menu = self.menuBar()
42
43          file_menu = menu.addMenu("&File")
44          file_menu.addAction(button_action)
45          file_menu.addSeparator()
46
47          file_submenu = file_menu.addMenu("Submenu")
48          file_submenu.addAction(button_action2)
49
50
51      def onMyToolBarButtonClick(self, s):
52          print("click", s)
```

Save a copy of your file as `MyApp_menus.py` as we'll need it again later.

Widgets

In Qt (and most User Interfaces) 'widget' is the name given to a component of the UI that the user can interact with. User interfaces are made up of multiple widgets, arranged within the window.

Qt comes with a large selection of widgets available, and even allows you to create your own custom and customised widgets.

Load up a fresh copy of `MyApp_window.py` and save it under a new name for this section.

Big ol' list of widgets

A full list of widgets is available on the Qt documentation, but let's have a look at them quickly in action.

All the Qt5 widgets.

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *

# Only needed for access to command line arguments
import sys


# Subclass QMainWindow to customise your application's main window
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")


        layout = QVBoxLayout()
        widgets = [QCheckBox,
            QComboBox,
            QDateEdit,
            QDateTimeEdit,
            QDial,
            QDoubleSpinBox,
            QFontComboBox,
            QLCDNumber,
            QLabel,
            QLineEdit,
            QProgressBar,
            QPushButton,
            QRadioButton,
            QSlider,
            QSpinBox,
            QTimeEdit]

        for w in widgets:
            layout.addWidget(w())
```

```
38
39
40          widget = QWidget()
41          widget.setLayout(layout)
42
43          # Set the central widget of the Window. Widget will expand
44          # to take up all the space in the window by default.
45          self.setCentralWidget(widget)
46
47
48  # You need one (and only one) QApplication instance per application.
49  # Pass in sys.argv to allow command line arguments for your app.
50  # If you know you won't use command line arguments QApplication([]) works too.
51  app = QApplication(sys.argv)
52
53  window = MainWindow()
54  window.show() # IMPORTANT!!!!! Windows are hidden by default.
55
56  # Start the event loop.
57  app.exec_()
58
59
60  # Your application won't reach here until you exit and the event
61  # loop has stopped.
```

To do this we're going to take the skeleton of our application and replace the `QLabel` with a `QWidget`. This is the generic form of a Qt widget.

Here we're not using it directly. We apply a list of widgets - in a layout, which we will cover shortly - and then add the `QWidget` as the central widget for the window. The result is that we fill the window with widgets, with the `QWidget` acting as a container.

Compound widgets

Note that it's possible to use this `QWidget` layout trick to create custom compound widgets. For example you can take a base `QWidget` and overlay a layout containing multiple widgets of different types. This 'widget' can then be inserted into other layouts as normal. We'll cover custom widgets in more detail later.

Lets have a look at all the example widgets, from top to bottom:

Widget	What it does
`QCheckbox`	A checkbox
`QComboBox`	A dropdown list box
`QDateEdit`	For editing dates and datetimes
`QDateTimeEdit`	For editing dates and datetimes
`QDial`	Rotateable dial
`QDoubleSpinbox`	A number spinner for floats
`QFontComboBox`	A list of fonts
`QLCDNumber`	A quite ugly LCD display
`QLabel`	Just a label, not interactive
`QLineEdit`	Enter a line of text
`QProgressBar`	A progress bar
`QPushButton`	A button
`QRadioButton`	A toggle set, with only one active item
`QSlider`	A slider
`QSpinBox`	An integer spinner
`QTimeEdit`	For editing times

There are actually more widgets than this, but they don't fit so well! You can see them all by checking the documentation. Here we're going to take a closer look at the a subset of the most useful.

QLabel

We'll start the tour with `QLabel`, arguably one of the simplest widgets available in the Qt toolbox. This is a simple one-line piece of text that you can position in your application. You can set the text by passing in a str as you create it:

```
widget = QLabel("Hello")
```

Or, by using the `.setText()` method:

```
widget = QLabel("1")  # The label is created with the text 1.
widget.setText("2")   # The label now shows 2.
```

You can also adjust font parameters, such as the size of the font or the alignment of text in the widget.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = QLabel("Hello")
        font = widget.font()
        font.setPointSize(30)
        widget.setFont(font)
        widget.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)

        self.setCentralWidget(widget)
```

Font tips

Note that if you want to change the properties of a widget font it is usually better to get the *current* font, update it and then apply it back. This ensures the font face remains in keeping with the desktop conventions.

The alignment is specified by using a flag from the `Qt.` namespace. The flags available for horizontal alignment are:

Flag	Behaviour
`Qt.AlignLeft`	Aligns with the left edge.
`Qt.AlignRight`	Aligns with the right edge.
`Qt.AlignHCenter`	Centers horizontally in the available space.
`Qt.AlignJustify`	Justifies the text in the available space.

The flags available for vertical alignment are:

Flag	Behaviour
`Qt.AlignTop`	Aligns with the top.
`Qt.AlignBottom`	Aligns with the bottom.
`Qt.AlignVCenter`	Centers vertically in the available space.

You can combine flags together using pipes (`|`), however note that you can only use vertical or horizontal alignment flag at a time.

```
1 align_top_left = Qt.AlignLeft | Qt.AlignTop
```

Qt Flags

Note that you use an *OR* pipe (`|`) to combine the two flags (not `A & B`). This is because the flags are non-overlapping bitmasks. e.g. `Qt.AlignLeft` has the hexadecimal value `0x0001`, while `Qt.AlignBottom` is `0x0040`. By ORing together we get the value `0x0041` representing 'bottom left'. This principle applies to all other combinatorial Qt flags.

If this is gibberish to you, feel free to ignore and move on. Just remember to use |!

Finally, there is also a shorthand flag that centers in both directions simultaneously:

Flag	Behaviour
`Qt.AlignCenter`	Centers horizontally *and* vertically

Weirdly, you can also use `QLabel` to display an image using `.setPixmap()`. This

accepts an *pixmap*, which you can create by passing an image filename to `QPixmap`. In the example files provided with this book you can find a file `otje.jpg` which you can display in your window as follows:

```
1  widget.setPixMap(QPixmap('otje.jpg'))
```

Otgon “Otje” Ginge the cat.

What a lovely face. By default the image scales while maintaining its aspect ratio. If you want it to stretch and scale to fit the window completely you can set `.setScaledContents(True)` on the `QLabel`.

```
1 widget.setScaledContents(True)
```

QCheckBox

The next widget to look at is `QCheckBox()` which, as the name suggests, presents a checkable box to the user. However, as with all Qt widgets there are number of configurable options to change the widget behaviours.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = QCheckBox()
        widget.setCheckState(Qt.Checked)

        # For tristate: widget.setCheckState(Qt.PartiallyChecked)
        # Or: widget.setTriState(True)
        widget.stateChanged.connect(self.show_state)

        self.setCentralWidget(widget)


    def show_state(self, s):
        print(s == Qt.Checked)
        print(s)
```

You can set a checkbox state programmatically using `.setChecked` or `.setCheckState`. The former accepts either `True` or `False` representing checked or unchecked respectively. However, with `.setCheckState` you also specify a particular checked state using a `Qt.` namespace flag:

Flag	Behaviour
`Qt.Unchecked`	Item is unchecked
`Qt.PartiallyChecked`	Item is partially checked
`Qt.Checked`	Item is unchecked

A checkbox that supports a partially-checked (`Qt.PartiallyChecked`) state is com-

monly referred to as ‘tri-state’, that is being neither on nor off. A checkbox in this state is commonly shown as a greyed out checkbox, and is commonly used in hierarchical checkbox arrangements where sub-items are linked to parent checkboxes.

If you set the value to `Qt.PartiallyChecked` the checkbox will become tristate. You can also `.setTriState(True)` to set tristate support on a You can also set a checkbox to be tri-state without setting the current state to partially checked by using `.setTriState(True)`

> You may notice that when the script is running the current state number is displayed as an `int` with checked = `2`, unchecked = `0`, and partially checked = `1`. You don’t need to remember these values, the `Qt.Checked` namespace variable `== 2` for example. This is the value of these state’s respective flags. This means you can test state using `state == Qt.Checked`.

QComboBox

The `QComboBox` is a drop down list, closed by default with an arrow to open it. You can select a single item from the list, with the currently selected item being shown as a label on the widget. The combo box is suited to selection of a choice from a long list of options.

You have probably seen the combo box used for selection of font faces, or size, in word processing applications. Although Qt actually provides a specific font-selection combo box as `QFontComboBox`.

You can add items to a `QComboBox` by passing a list of strings to `.addItems()`. Items will be added in the order they are provided.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = QComboBox()
        widget.addItems(["One", "Two", "Three"])

        # The default signal from currentIndexChanged sends the index
        widget.currentIndexChanged.connect( self.index_changed )

        # The same signal can send a text string
        widget.currentIndexChanged[str].connect( self.text_changed )

        self.setCentralWidget(widget)


    def index_changed(self, i): # i is an int
        print(i)

    def text_changed(self, s): # s is a str
        print(s)
```

The `.currentIndexChanged` signal is triggered when the currently selected item is updated, by default passing the index of the selected item in the list. However, when connecting to the signal you can also request an alternative version of the signal by appending `[str]` (think of the signal as a `dict`). This alternative interface instead provides the label of the currently selected item, which is often more useful.

`QComboBox` can also be editable, allowing users to enter values not currently in the list and either have them inserted, or simply used as a value. To make the box editable:

```
widget.setEditable(True)
```

You can also set a flag to determine how the insert is handled. These flags are stored on the `QComboBox` class itself and are listed below:

Flag	Behaviour
`QComboBox.NoInsert`	No insert
`QComboBox.InsertAtTop`	Insert as first item
`QComboBox.InsertAtCurrent`	Replace currently selected item
`QComboBox.InsertAtBottom`	Insert after last item
`QComboBox.InsertAfterCurrent`	Insert after current item
`QComboBox.InsertBeforeCurrent`	Insert before current item
`QComboBox.InsertAlphabetically`	Insert in alphabetical order

To use these, apply the flag as follows:

```
widget.setInsertPolicy(QComboBox.InsertAlphabetically)
```

You can also limit the number of items allowed in the box by using `.setMaxCount`, e.g.

```
widget.setMaxCount(10)
```

QListBox

Next `QListBox`. It's very similar to `QComboBox`, differing mainly in the signals available.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = QListWidget()
        widget.addItems(["One", "Two", "Three"])

        # In QListWidget there are two separate signals for the item, and the s\
tr
        widget.currentItemChanged.connect( self.index_changed )
        widget.currentTextChanged.connect( self.text_changed )

        self.setCentralWidget(widget)


    def index_changed(self, i): # Not an index, i is a QListItem
        print(i.text())

    def text_changed(self, s): # s is a str
        print(s)
```

`QListBox` offers an `currentItemChanged` signal which sends the `QListItem` (the element of the list box), and a `currentTextChanged` signal which sends the text.

QLineEdit

The `QLineEdit` widget is a simple single-line text editing box, into which users can type input. These are used for form fields, or settings where there is no restricted list of valid inputs. For example, when entering an email address, or computer name.

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = QLineEdit()
        widget.setMaxLength(10)
        widget.setPlaceholderText("Enter your text")

        #widget.setReadOnly(True) # uncomment this to make readonly

        widget.returnPressed.connect(self.return_pressed)
        widget.selectionChanged.connect(self.selection_changed)
        widget.textChanged.connect(self.text_changed)
        widget.textEdited.connect(self.text_edited)

        self.setCentralWidget(widget)


    def return_pressed(self):
        print("Return pressed!")
        self.centralWidget().setText("BOOM!")

    def selection_changed(self):
        print("Selection changed")
        print(self.centralWidget().selectedText())

    def text_changed(self, s):
        print("Text changed...")
```

```
32          print(s)
33
34      def text_edited(self, s):
35          print("Text edited...")
36          print(s)
```

As demonstrated in the above code, you can set a maximum length for the text in a line edit.

Layouts

So far we've successfully created a window, and we've added a widget to it. However we normally want to add more than one widget to a window, and have some control over where it ends up. To do this in Qt we use *layouts*. There are 4 basic layouts available in Qt, which are listed in the following table.

Layout	Behaviour
`QHBoxLayout`	Linear horizontal layout
`QVBoxLayout`	Linear vertical layout
`QGridLayout`	In indexable grid XxY
`QStackedLayout`	Stacked (z) in front of one another

Qt Designer

You can actually design and lay out your interface graphically using the Qt designer, which we will cover later. Here we're using code, as it's simpler to understand and experiment with the underlying system.

As you can see, there are three positional layouts available in Qt. The `VBoxLayout`, `QHBoxLayout` and `QGridLayout`. In addition there is also `QStackedLayout` which allows you to place widgets one on top of the other within the same space, yet showing only one layout at a time.

Load up a fresh copy of `MyApp_window.py` and save it under a new name for this section.

Before we start experimenting with the different layouts, we're first going to create a very simple custom widget that we can use to visualise the layouts that we use. Add the following code to your file as a new class at the top level:

Custom color widget

```
class Color(QWidget):

    def __init__(self, color, *args, **kwargs):
        super(Color, self).__init__(*args, **kwargs)
        self.setAutoFillBackground(True)

        palette = self.palette()
        palette.setColor(QPalette.Window, QColor(color))
        self.setPalette(palette)
```

In this code we subclass `QWidget` to create our own custom widget `Color`. We accept a single parameter when creating the widget — `color` (a `str`). We first set `.setAutoFillBackground` to `True` to tell the widget to automatically fill its background with the window cooler. Next we get the current palette (which is the global desktop palette by default) and change the current `QPalette.Window` color to a new `QColor` described by the value `color` we passed in. Finally we apply this palette back to the widget. The end result is a widget that is filled with a solid color, that we specified when we created it.

If you find the above confusing, don't worry too much. We'll cover custom widgets in more detail later. For now it's sufficient that you understand that calling you can create a solid-filled red widget by doing the following:

```
Color('red')
```

First let's test our new `Color` widget by using it to fill the entire window in a single color. Once it's complete we can add it to the `QMainWindow` using `.setCentralWidget` and we get a solid red window.

Adding a widget to the layout

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        widget = Color('red')
        self.setCentralWidget(widget)
```

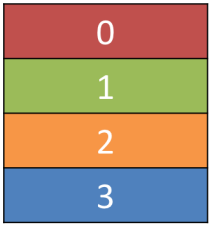
Run it!

The window will appear, filled completely with the color red. Notice how the widget expands to fill all the available space.

Next we'll look at each of the available Qt layouts in turn. Note that to add our layouts to the window we will need a dummy `QWidget` to hold the layout.

`QVBoxLayout` vertically arranged widgets

With `QVBoxLayout` you arrange widgets one above the other linearly. Adding a widget adds it to the bottom of the column.



A QVBoxLayout, filled from top to bottom.

Let's add our widget to a layout. Note that in order to add a layout to the `QMainWindow` we need to apply it to a dummy `QWidget`. This allows us to then use `.setCentralWidget` to apply the widget (and the layout) to the window. Our

coloured widgets will arrange themselves in the layout, contained within the `QWidget` in the window. First we just add the red widget as before.

QVBoxLayout

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout = QVBoxLayout()

        layout.addWidget(Color('red'))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

Run it!

Notice the border now visible around the red widget. This is the layout spacing — we'll see how to adjust that later.

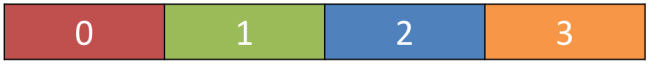
If you add a few more coloured widgets to the layout you'll notice that they line themselves up vertical in the order they are added.

QVBoxLayout

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout = QVBoxLayout()

        layout.addWidget(Color('red'))
        layout.addWidget(Color('green'))
        layout.addWidget(Color('blue'))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

`QHBoxLayout` horizontally arranged widgets

`QHBoxLayout` is the same, except moving horizontally. Adding a widget adds it to the right hand side.



A QHBoxLayout, filled from left to right.

To use it we can simply change the `QVBoxLayout` to a `QHBoxLayout`. The boxes now flow left to right.

QHBoxLayout

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout = QHBoxLayout()

        layout.addWidget(Color('red'))
        layout.addWidget(Color('green'))
        layout.addWidget(Color('blue'))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

Nesting layouts

For more complex layouts you can nest layouts inside one another using `.addLayout` on a layout. Below we add a `QVBoxLayout` into the main `QHBoxLayout`. If we add some widgets to the `QVBoxLayout`, they'll be arranged vertically in the first slot of the parent layout.

Nested layouts

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout1 = QHBoxLayout()
        layout2 = QVBoxLayout()
```

```
10          layout3 = QVBoxLayout()
11
12          layout2.addWidget(Color('red'))
13          layout2.addWidget(Color('yellow'))
14          layout2.addWidget(Color('purple'))
15
16          layout1.addLayout( layout2 )
17
18          layout1.addWidget(Color('green'))
19
20          layout3.addWidget(Color('red'))
21          layout3.addWidget(Color('purple'))
22
23          layout1.addLayout( layout3 )
24
25          widget = QWidget()
26          widget.setLayout(layout1)
27          self.setCentralWidget(widget)
```

Run it!

The widgets should arrange themselves in 3 columns horizontally, with the first column also containing 3 widgets stacked vertically. Experiment!

You can set the spacing around the layout using `.setContentMargins` or set the spacing between elements using `.setSpacing`.

```
1  layout1.setContentsMargins(0,0,0,0)
2  layout1.setSpacing(20)
```

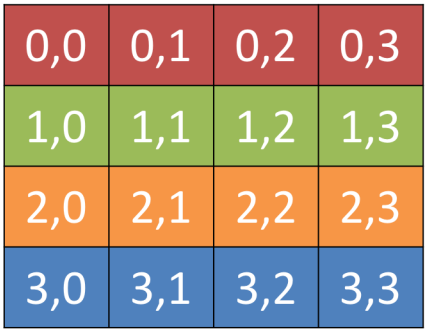
The following code shows the combination of nested widgets and layout margins and spacing. Experiment with the numbers til you get a feel for them.

Margins and spacing

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout1 = QHBoxLayout()
        layout2 = QVBoxLayout()
        layout3 = QVBoxLayout()

        layout1.setContentsMargins(0,0,0,0)
        layout1.setSpacing(20)

        layout2.addWidget(Color('red'))
        layout2.addWidget(Color('yellow'))
        layout2.addWidget(Color('purple'))

        layout1.addLayout( layout2 )

        layout1.addWidget(Color('green'))

        layout3.addWidget(Color('red'))
        layout3.addWidget(Color('purple'))

        layout1.addLayout( layout3 )

        widget = QWidget()
        widget.setLayout(layout1)
        self.setCentralWidget(widget)
```

`QGridLayout` widgets arranged in a grid

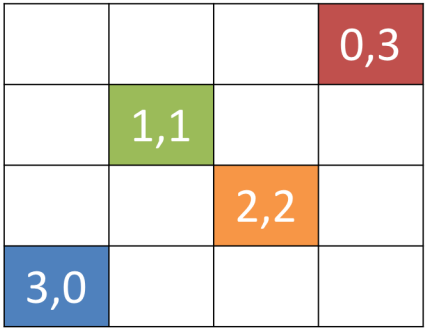
As useful as they are, if you try and using `QVBoxLayout` and `QHBoxLayout` for laying out multiple elements, e.g. for a form, you'll find it very difficult to ensure differently sized widgets line up. The solution to this is `QGridLayout`.



A QGridLayout showing the grid positions for each location.

`QGridLayout` allows you to position items specifically in a grid. You specify row and column positions for each widget. You can skip elements, and they will be left empty.

Usefully, for `QGridLayout` you don't need to fill all the positions in the grid.



A QGridLayout with unfilled slots.

QGridLayout

```
1  class MainWindow(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(MainWindow, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle("My Awesome App")
7
8          layout = QGridLayout()
```

```
9
10          layout.addWidget(Color('red'), 0, 0)
11          layout.addWidget(Color('green'), 1, 0)
12          layout.addWidget(Color('blue'), 1, 1)
13          layout.addWidget(Color('purple'), 2, 1)
14
15          widget = QWidget()
16          widget.setLayout(layout)
17          self.setCentralWidget(widget)
```

`QStackedLayout` multiple widgets in the same space

The final layout we'll cover is the `QStackedLayout`. As described, this layout allows you to position elements directly in front of one another. You can then select which widget you want to show. You could use this for drawing layers in a graphics application, or for imitating a tab-like interface. Note there is also `QStackedWidget` which is a container widget that works in exactly the same way. This is useful if you want to add a stack directly to a `QMainWindow` with `.setCentralWidget`.

QStackedLayout — in use only the uppermost widget is visible, which is by default the first widget added to the layout.

QStackedLayout, with the 2nd (1) widget selected and brought to the front.

QStackedLayout

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout = QStackedLayout()

        layout.addWidget(Color('red'))
        layout.addWidget(Color('green'))
        layout.addWidget(Color('blue'))
        layout.addWidget(Color('yellow'))

        layout.setCurrentIndex(3)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

`QStackedWidget` is exactly how tabbed views in applications work. Only one view ('tab') is visible at any one time. You can control which widget to show at any time

by using `.setCurrentIndex()` or `.setCurrentWidget()` to set the item by either the index (in order the widgets were added) or by the widget itself.

Below is a short demo using `QStackedLayout` in combination with `QButton` to to provide a tab-like interface to an application:

Tabbed interface

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        pagelayout = QVBoxLayout()
        button_layout = QHBoxLayout()
        layout = QStackedLayout()

        pagelayout.addLayout(button_layout)
        pagelayout.addLayout(layout)

        for n, color in enumerate(['red','green','blue','yellow']):
            btn = QPushButton( str(color) )
            btn.pressed.connect( lambda n=n: layout.setCurrentIndex(n) )
            button_layout.addWidget(btn)
            layout.addWidget(Color(color))

        widget = QWidget()
        widget.setLayout(pagelayout)
        self.setCentralWidget(widget)
```

Helpfully. Qt actually provide a built-in TabWidget that provides this kind of layout out of the box - albeit in widget form. Below the tab demo is recreated using `QTabWidget`:

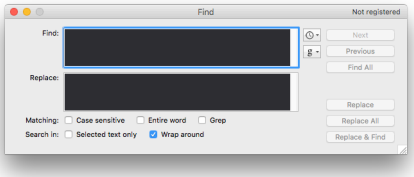
QTabWidget

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")


        tabs = QTabWidget()
        tabs.setDocumentMode(True)
        tabs.setTabPosition(QTabWidget.East)
        tabs.setMovable(True)

        for n, color in enumerate(['red','green','blue','yellow']):
            tabs.addTab( Color(color), color)

        self.setCentralWidget(tabs)
```

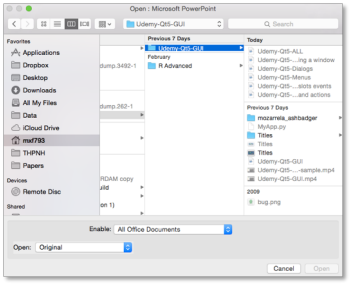
As you can see, it's a little more straightforward — and a bit more attractive! You can set the position of the tabs using the cardinal directions, toggle whether tabs are moveable with `.setMoveable` and turn a 'document mode' on and off which (on OS X) shows a slimmer tab interface. We encounter more of these advanced widgets later.

Dialogs

Dialogs are useful GUI components that allow you to *communicate* with the user (hence the name dialog). They are commonly used for file Open/Save, settings, preferences, or for functions that do not fit into the main UI of the application. They are small modal (or *blocking*) windows that sit in front of the main application until they are dismissed. Qt actually provides a number of 'special' dialogs for the most common use-cases, allowing you to take advantage of desktop-specific tools for a better user experience.



Standard GUI features — A search dialog



Standard GUI features — A file Open dialog

In Qt dialog boxes are handled by the `QDialog` class. To create a new dialog box simply create a new object of `QDialog` type (or a subclass), passing in a parent widget, e.g. `QMainWindow` as its parent.

Let's create our own `QDialog`, we'll use our menu example code so we can start a dialog window when a button on the toolbar is pressed.

Load up a fresh copy of `MyApp_menus.py` and save it under a new name for this section.

```
class MainWindow(QMainWindow):

    # def __init__ etc.
    # ... not shown for clarity

    def onMyToolBarButtonClick(self, s):
        print("click", s)


        dlg = QDialog(self)
        dlg.setWindowTitle("HELLO!")
        dlg.exec_()

```

In the triggered function (that receives the signal from the button) we create the dialog instance, passing our `QMainWindow` instance as a parent. This will make the dialog a *modal window* of `QMainWindow`. This means the dialog will completely block interaction with the parent window.

Once we have created the dialog, we start it using `.exec_()` - just like we did for `QApplication` to create the main event loop of our application. That's not a coincidence: when you exec the `QDialog` an entirely new event loop - specific for the dialog - is created.

Remember that there can be only one Qt event loop running at any time! The `QDialog` completely blocks your application execution. Don't start a dialog and expect anything else to happen anywhere else in your application.

We'll cover how you can use multithreading to get you out of this pickle in a later chapter.

Run it! The window will display, now click the bug button and a modal window should appear. You can exit by clicking the [x].

Like our very first window, this isn't very interesting. Let's fix that by adding a dialog title and a set of OK and Cancel buttons to allow the user to *accept* or *reject* the modal.

To customise the `QDialog` we can subclass it — again you *can* customise the dialog without subclassing, but it's nicer if you do.

```
class CustomDialog(QDialog):

    def __init__(self, *args, **kwargs):
        super(CustomDialog, self).__init__(*args, **kwargs)

        self.setWindowTitle("HELLO!")

        QBtn = QDialogButtonBox.Ok | QDialogButtonBox.Cancel

        self.buttonBox = QDialogButtonBox(QBtn)
        self.buttonBox.accepted.connect(self.accept)
        self.buttonBox.rejected.connect(self.reject)

        self.layout = QVBoxLayout()
        self.layout.addWidget(self.buttonBox)
        self.setLayout(self.layout)


class MainWindow(QMainWindow):


    # def __init__ etc.
    # ... not shown for clarity


    def onMyToolBarButtonClick(self, s):
        print("click", s)


        dlg = CustomDialog(self)
        if dlg.exec_():
            print("Success!")
        else:
            print("Cancel!")
```

In the above code, we first create our subclass of `QDialog` which we've called `CustomDialog`. As for the `QMainWindow` we customise it within the `\\__init__` block to ensure that our customisations are created as the object is created. First we set a title for the `QDialog` using `.setWindowTitle()`, exactly the same as we did for our main window.

The next block of code is concerned with creating and displaying the dialog buttons. This is probably a bit more involved than you were expecting. However, this is due to Qt's flexibility in handling dialog button positioning on different platforms.

You could choose to ignore this and use a standard `QButton` in a layout, but the approach outlined here ensures that your dialog respects the host desktop standards (Ok on left vs. right for example). Breaking these expectations can be incredibly annoying to your users, so I wouldn't recommend it.

The first step in creating a dialog button box is to define the buttons want to show, using namespace attributes from `QDialogButtonBox`. Constructing a line of multiple buttons is as simple as OR-ing them together using a pipe (`|`). The full list of buttons available is below:

```
Button types
——————————
QDialogButtonBox.Ok
QDialogButtonBox.Open
QDialogButtonBox.Save
QDialogButtonBox.Cancel
QDialogButtonBox.Close
QDialogButtonBox.Discard
QDialogButtonBox.Apply
QDialogButtonBox.Reset
QDialogButtonBox.RestoreDefaults
QDialogButtonBox.Help
QDialogButtonBox.SaveAll
QDialogButtonBox.Yes
QDialogButtonBox.YesToAll
QDialogButtonBox.No
QDialogButtonBox.NoToAll|
QDialogButtonBox.Abort
```

```
QDialogButtonBox.Retry
QDialogButtonBox.Ignore
QDialogButtonBox.NoButton
```

These should be sufficient to create any dialog box you can think of. For example, to show an OK and a Cancel button we used:

```
1 buttons = QDialogButtonBox.Ok | QDialogButtonBox.Cancel
```

The variable `buttons` now contains a bit mask flag representing those two buttons. Next, we must create the `QDialogButtonBox` instance to hold the buttons. The flag for the buttons to display is passed in as the first parameter.

To make the buttons have any effect, you must connect the correct `QDialogButtonBox` signals to the slots on the dialog. In our case we've connected the `.accepted` and `.rejected` signals from the `QDialogButtonBox` to the handlers for `.accept()` and `.reject()` on our subclass of `QDialog`.

Lastly, to make the `QDialogButtonBox` appear in our dialog box we must add it to the dialog layout. So, as for the main window we create a layout, and add our `QDialogButtonBox` to it (`QDialogButtonBox` is a widget), and then set that layout on our dialog.

Run it! Click to launch the dialog and you will see a dialog box with buttons in it.

Congratulations! You've created your first dialog box. Of course, you can continue to add any other content to the dialog box that you like. Simply insert it into the layout as normal.

Qt Creator

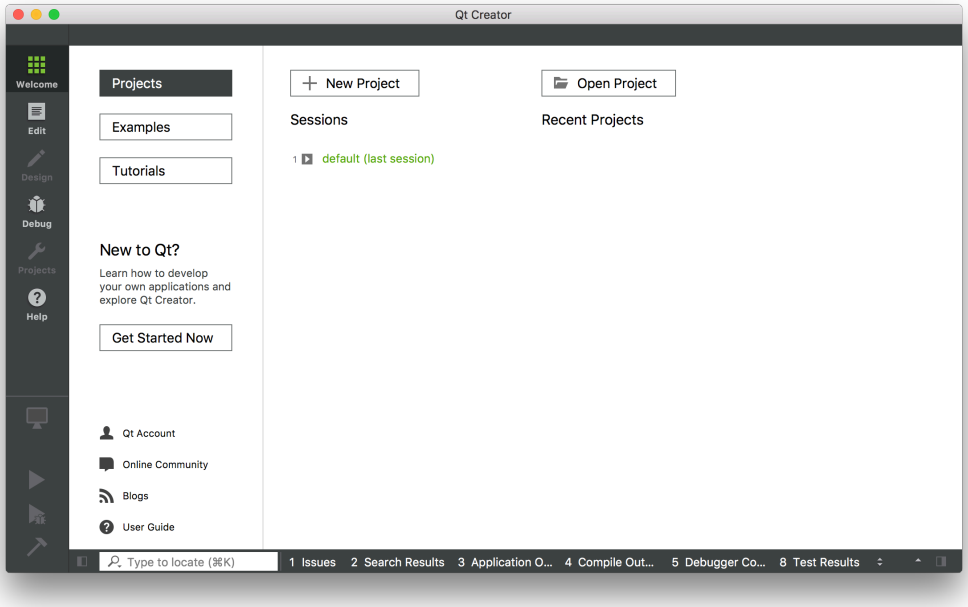
So far we have been creating apps using Python code. This is fine for simple applications, but as your applications get larger or interfaces become more complicated, it can get a bit cumbersome to define all elements programmatically in this way. The good news is that Qt comes with a graphical editor — Qt Creator — which contains a drag-and-drop UI editor.

In this chapter we'll cover the basics of creating UIs with Qt Creator. The principles, layouts and widgets are identical, so you can apply everything you've already learnt. You'll also need your knowledge of the Python API to hook up your application logical later.

You can download this from the Qt website. Just go to https://www.qt.io/download and download the Qt package. You can opt to install only Creator during the installation.

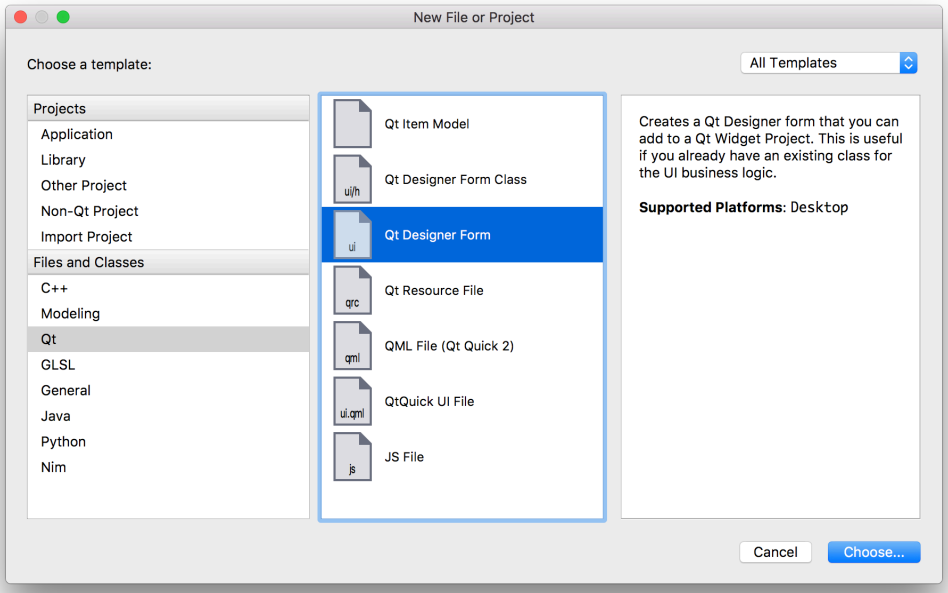
Creating a .ui file

Open up Qt Creator and you will be presented with the main window. The designer is available via the tab on the left hand side. However, to activate this you first need to start creating a `.ui` file.



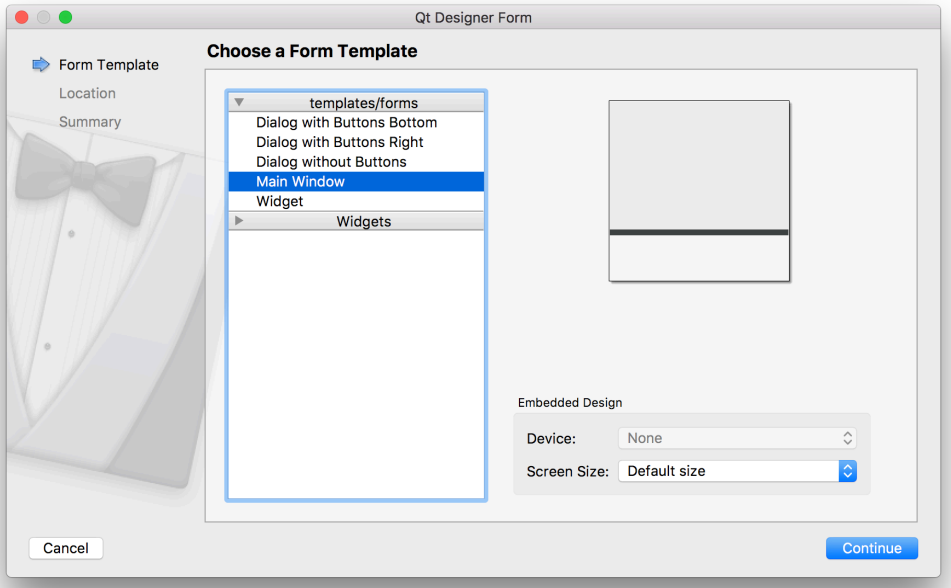
The Qt Creator interface, with the Design section shown on the left.

To create a `.ui` file go to File -> New File or Project... In the window that appears select *Qt* under *Files and Classes* on the left, then select *Qt Designer Form* on the right. You'll notice the icon has "ui" on it, showing the type of file you're creating.



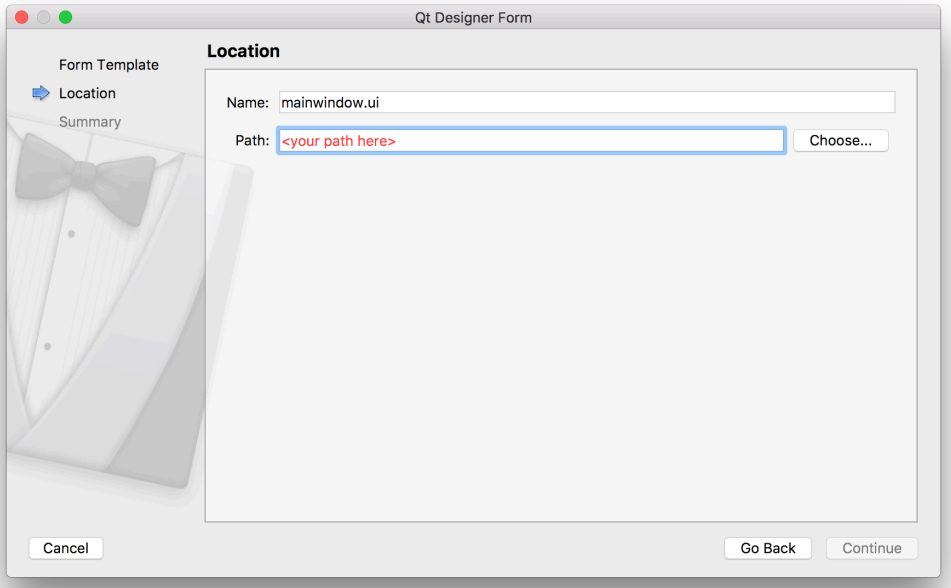
Create a new Qt .ui file.

In the next step you'll be asked what type of widget you want to create. If you are starting an application then *Main Window* is the right choice. However, you can also create `.ui` files for dialog boxes, forms and custom compound widgets.



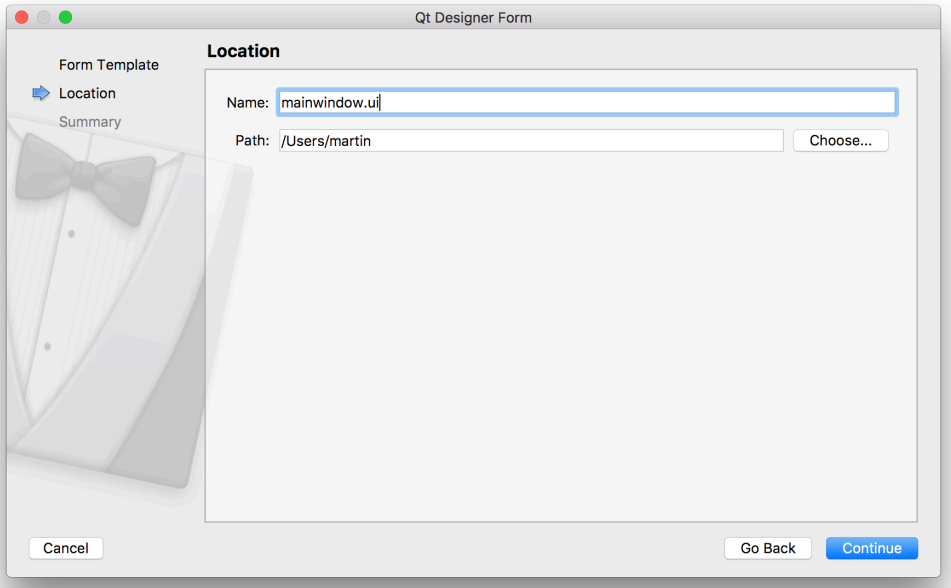
Select the type of widget to create, for most applications this will be Main Window.

Next choose a filename and save folder for your file. Save your `.ui` file with the same name as the class you'll be creating, just to make make subsequent commands simpler.



Choose save name and folder your your file.

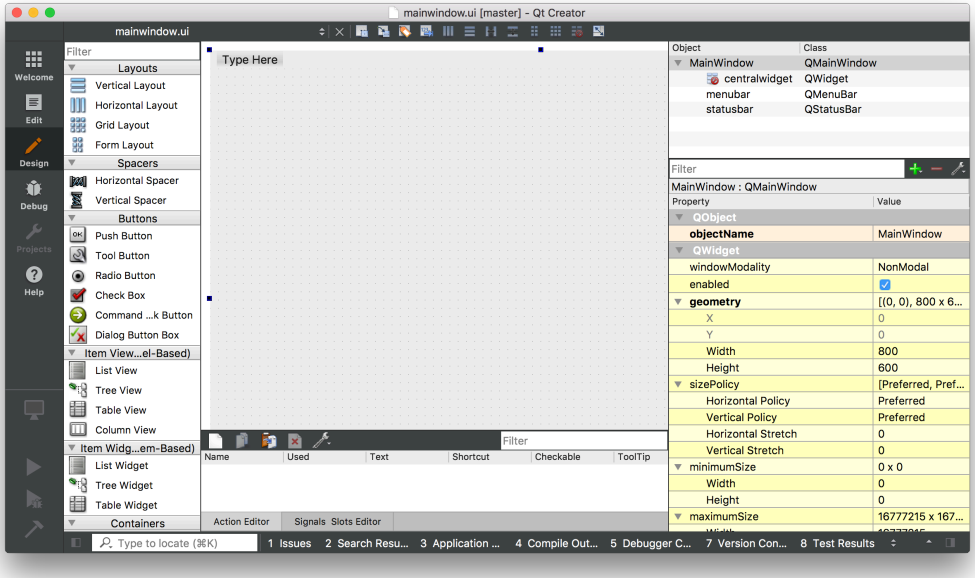
Finally, you can choose to add the file to your version control system if you're using one. Feel free to skip this step — it doesn't affect your UI.



Optionally add the file to your version control, e.g. Git.

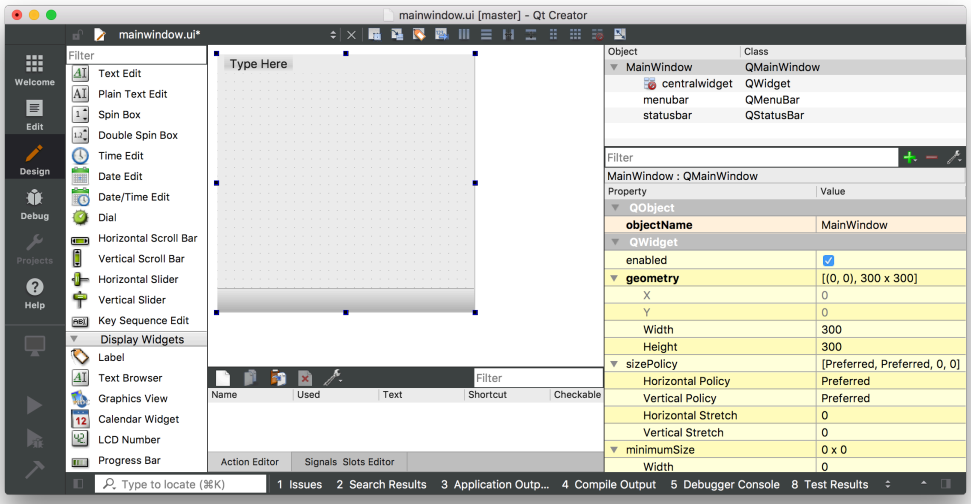
Laying out your Main Window

You'll be presented with your newly created main window in the UI designer. There isn't much to see to begin with, just a grey working area representing the window, together with the beginnings of a window menu bar.



The initial view of the created main window.

You can resize the window by clicking the window and dragging the blue handles on each corner.

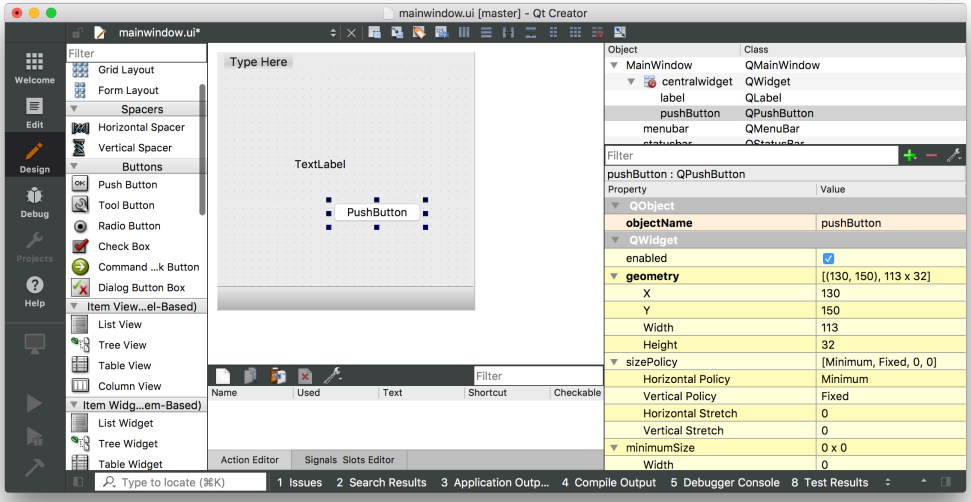


The initial view of the created main window.

The first step in building an application is to add some widgets to your window. In our first applications we learnt that to set the central widget for a `QMainWindow` we need to use `.setCentralWidget()`. We also saw that to add multiple widgets with a layout, we need an intermediary `QWidget` to apply the layout to, rather than adding the layout to the window directly.

Qt Creator takes care of this for you automatically, although it's not particularly obvious about it.

To add multiple widgets to the main window with a layout, first drag your widgets onto the `QMainWindow`. Here we're dragging 3 labels. It doesn't matter where you drop them.

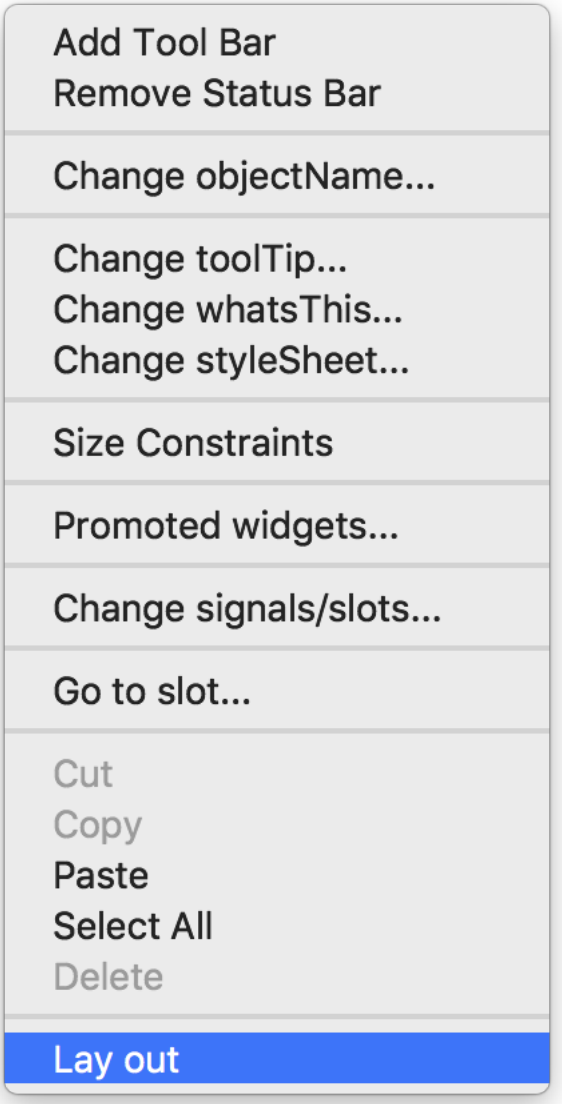


Main window with 1 labels and 1 button added.

We've created 2 widgets by dragging them onto the window, made them children of that window. We can now apply a layout.

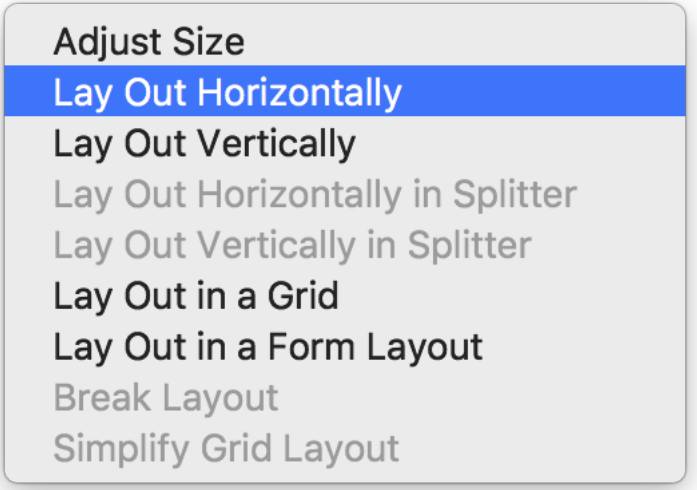
Find the `QMainWindow` in the right hand panel (it should be right at the top). Underneath you see *centralwidget* representing the window's central widget. The icon for the central widget show the current layout applied. Initially it has a red circle-cross through it, showing that there is no layout active.

Right click on the `QMainWindow` object, and find 'Layout' in the resulting dropdown.



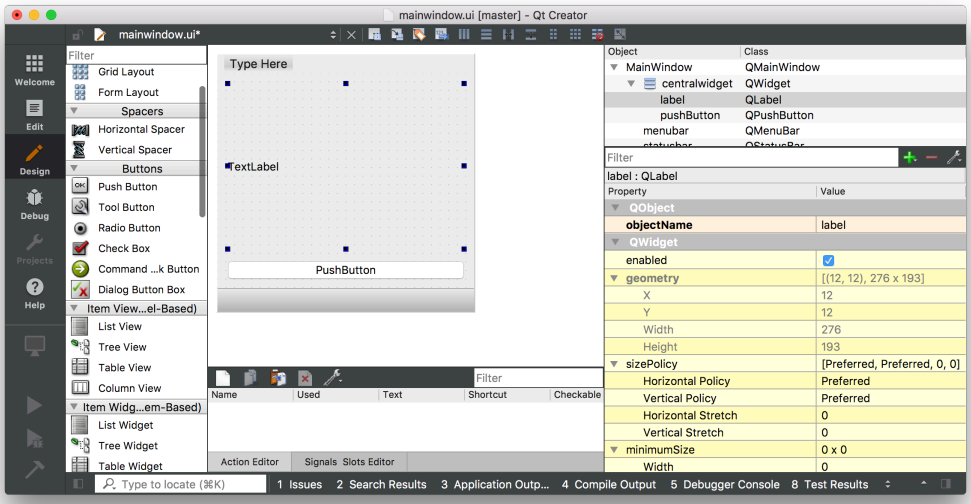
Right click on the main window, and choose layout.

Next you'll see a list of layouts which you can apply to the window. Select *Lay Out Horizontally* and the layout will be applied to the widget.



Select layout to apply to the main window.

The selected layout is applied to the the *centralwidget* of the `QMainWindow` and the widgets are added the layout, being laid out depending on the selected layout. Note that in Qt Creator you can actually drag and re-order the widgets within the layout, or select a different layout, as you like. This makes it especially nice to prototyping and trying out things.



Verticallayout applied to widgets on the main window.

Using your generated .ui file

We've created a very simple UI. The next step is to get this into Python and use it to construct a working application.

First save your `.ui` file — by default it will save at the location you chosen while creating it, although you can choose another location if you like.

The `.ui` file is in XML format. To use our UI from Python we have two alternative methods available —

1. load into into a class using the `.loadUI()` method
2. convert it to Python using the `pyuic5` tool.

These two approaches are covered below. Personally I prefer to convert the UI to a Python file to keep things similar from a programming & packaging point of view.

Loading the .ui file directly

To load `.ui` files we can use the `uic` module included with PyQt5, specifically the `uic.loadUI()`method. This takes the filename of a UI file and loads it creating a fully-functional PyQt5 object.

```
import sys
from PyQt5 import QtWidgets, uic

app = QtWidgets.QApplication(sys.argv)

window = uic.loadUi("mainwindow.ui")
window.show()
app.exec()
````

T> As the `uid.loadUI()` method turns an instance object you cannot attach cust\
om `__init__()` code. You can however handle this through a custom setup functi\
on.

```python
import sys
from PyQt5 import QtWidgets, uic

def mainwindow_setup(w):
    w.setTitle("MainWindow Title")

app = QtWidgets.QApplication(sys.argv)

window = uic.loadUi("mainwindow.ui")
mainwindow_setup(window)
window.show()
app.exec()
```

Converting your .ui file to Python

To generate a Python output file run `pyuic5` from the command line, passing the `.ui` file and the target file for output, with a `-o` parameter. The following will generate a Python file named `MainWindow.py` which contains our created UI.

```
pyuic5 mainwindow.ui -o MainWindow.py
```

If you're using PyQt4 the tool is named `pyuic4`, but is otherwise completely identical.

You can open the resulting `MainWindow.py` file in an editor to take a look, although you should *not* edit this file. The power of using Qt Creator is being able to edit, tweak and update your application while you develop. Any changes made to this file will be lost when you update it. However, you *can* override and tweak anything you like when you import and use the file in your applications.

Importing the resulting Python file works as for any other. You can import your class as follows. The `pyuic5` tool appends `Ui_` to the name of the object defined in *Qt Creator*, and it is this object you want to import.

```
from MainWindow import Ui_MainWindow
```

To create the main window in your application, create a class as normal but subclassing from both `QMainWindow` and your imported `Ui_MainWindow` class. Finally, call `self.setupUi(self)` from within the `__init__` to trigger the setup of the interface.

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)
        self.setupUi(self)
```

That's it. Your window is now fully set up.

Adding application logic

You can interact with widgets created through Qt Creator just as you would those created with code. To make things simpler `uic` adds all child widgets to the window object by their id name.

Extended Signals

We've previously covered the basics of what Qt signals are and how you can use them to make your application respond to actions and other occurrences. However, this merely scratches the surface of what you can achieve with the Qt signal/slot system.

In this chapter we'll look at ways you can extend and modify signal behaviour from within Python and how you can create custom signals yourself.

Modifying Signal Data

As you find yourself using signals more often, you'll often find that you want to be able to customise the data that is sent with them. Unfortunately there is no way to do this in Qt directly, but we can exploit some features of Python to make it work for us.

To start with, we'll look at how to send *less* data.

Imagine we have a function that accepts two parameters, with default values. However, neither of these is a string. How can we connect our `.windowTitleChanged` signal to this function?

By using a wrapper function (or a lambda) we can accept the signal's data, discard it, then call our target slot. So, for example, to discard data from a signal that emits a single value we could use the following construction.

```
def wrapper_function(x):
    real_function()  # To call target, discarding x
```

The `wrapper_function` accepts the `x` value, but does not pass it when calling `real_function`. We can also write this using a lambda as follows:

```
lambda x: fn()
```

Just like the function, the lambda accepts a single parameter `x`, then discards it calling the *real* target `fn` with no parameters:

It doesn't matter whether you use a normal function or `lambda` (anonymous function) for these. However, I tend to use the lambda syntax because it makes for tidier code.

If we want to send *more* data we can use a similar construction, but instead of discarding a parameter we add another. For example:

```
def wrapper_function(x)
    real_function(x, some_more_data)
```

Or again, with `lambda` syntax:

```
lambda x: fn(x, some_more_data)
```

There is a gotcha to be aware of here however. If you are wrapping a number of signals in turn and use a loop, you need to be aware of Python scoping behaviour for your loop variable and the wrapped function.

Here we're going to use a layout to create a list of widgets (don't worry about layouts yet, that'll be explained later). Copy the following code into a file and run it with Python.

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *


class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        layout = QVBoxLayout()

        for n in range(10):
            btn = QPushButton(str(n))
            btn.pressed.connect( lambda: self.my_custom_fn(n) )

            layout.addWidget(btn)

        widget = QWidget()
        widget.setLayout(layout)

        self.setCentralWidget(widget)

    def my_custom_fn(self, n):
        print("Button %d was clicked" % n)


app = QApplication([])

window = MainWindow()
window.show()
app.exec_()
```

You'll notice that as we iterate to add the widgets, we redirect using a lambda to our custom `clicked` function, passing the loop variable. The expectation is that clicking on each button will print a message along with the button's number in the

console. Try it.

Did you notice that clicking on all of the widgets gives the same result?

```
python3 signals_lambda.py
Button 9 was clicked
Button 9 was clicked
Button 9 was clicked
Button 9 was clicked
```

What's going on? The variable scoping rules of Python mean that when we use the loop variable inside the lambda, this is the *same object* as the loop variable. Each wrapped method will then contain a reference to the same variable, which will — once the loop is completed — contain the same final value of the loop. Each call to each wrapper will send the same value.

To prevent this we need to pass the extra data in as a named parameter to the `lambda` or wrapper function. This creates a new object, unique to that new namespace, holding the value of the loop at the time of its creation.

```
lambda x, data=data: fn(x, data)
```

So, now you should be able to pass just about anything to any function using signals! The final code now looks like this:

```
class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.setWindowTitle("My Awesome App")

        # QHBoxLayout is a horizontally stacking layout with new widgets
        # added to the right of previous widgets.
        layout = QVBoxLayout()

        for n in range(10):
            # Create a push button labeled with the loop number 0-9
            btn = QPushButton(str(n))
            # SIGNAL: The .pressed signal fires whenever the button is pressed.
            # We connect this to self.my_custom_fn via a lambda to pass in
            # additional data.
            # IMPORTANT: You must pass the additional data in as a named
            # parameter on the lambda to create a new namespace. Otherwise
            # the value of n will be bound to the final value in the parent
            # for loop (always 9).
            btn.pressed.connect( lambda n=n: self.my_custom_fn(n) )

            # Add the button to the layout. It will go to the right by default.
            layout.addWidget(btn)

        # Create a empty widget to hold the layout containing our buttons.
        widget = QWidget()

        # Set the layout containing our buttons onto the blank widget. We only
        # need to do this here because we can't set a layout on a QMainWindow.
        # So instead we're setting a layout on a widget, and then adding that
        # widget to the window(!)
        widget.setLayout(layout)

        self.setCentralWidget(widget)


    # SLOT: This function will receive the single value passed from the signal
```

```
40      def my_custom_fn(self, n):
41          print("Button %d was clicked" % n)
```

Custom Signals

The final bit of signals we're going to cover is custom signals. These allow you to use the Qt event loop to send data around your application. It's a great way to keep your app modular and responsive.

You can define your own signals using the `pyQtSignal` method provided by PyQt5. Signals must be defined as attributes of the class, passing in the type that will sent with the signals when creating it. You can choose any valid Python variable name for the name of the signal.

```
def MainWindow(QMainWindow):
    message = pyqtSignal(str)
    value = pyqtSignal(int)
```

These signals can then be used as normal:

```
window.value.emit(23)  # Signal on another object.
self.message.emit("my message")  # Signal on self.
```

You can create your own signals on any class that is a subclass of `QObject`. That includes all widgets, including the main window, dialog boxes, and so on.

You can send any Python type, including multiple types, and compound types (e.g. dictionaries, lists).

```
def MyClass(QObject):
    keyvalue = pyqtSignal(dict)
    data = pyqtSignal(tuple)
```

If you define your signal as `pyqtSignal(object)` it will be able to send any Python type. But this isn't recommend, as receiving handlers will then need to deal with all types.

QPainter and Bitmap Graphics

The first step towards creating custom widgets in PyQt5 is understanding bitmap (pixel-based) graphic operations. All standard widgets draw themselves as bitmaps on a rectangular "canvas" that forms the shape of the widget. Once you understand how this works you can draw any widget you like!

Bitmaps are rectangular grids of *pixels*, where each pixel (and its color) is represented by a number of "bits". They are distinct from vector graphics, where the image is stored as a series of line (or *vector*) drawing instructions which are repeated form the image. If you're viewing vector graphics on your screen they are being *rasterised* (i.e. converted into a bitmap image) to be displayed as pixels on the screen.

In this tutorial we'll take a look at `QPainter`, Qt's API for performing bitmap graphic operations and the basis for drawing your own widgets. We'll go through some basic drawing operations and finally put it all together to create our own little Paint app.

QPainter

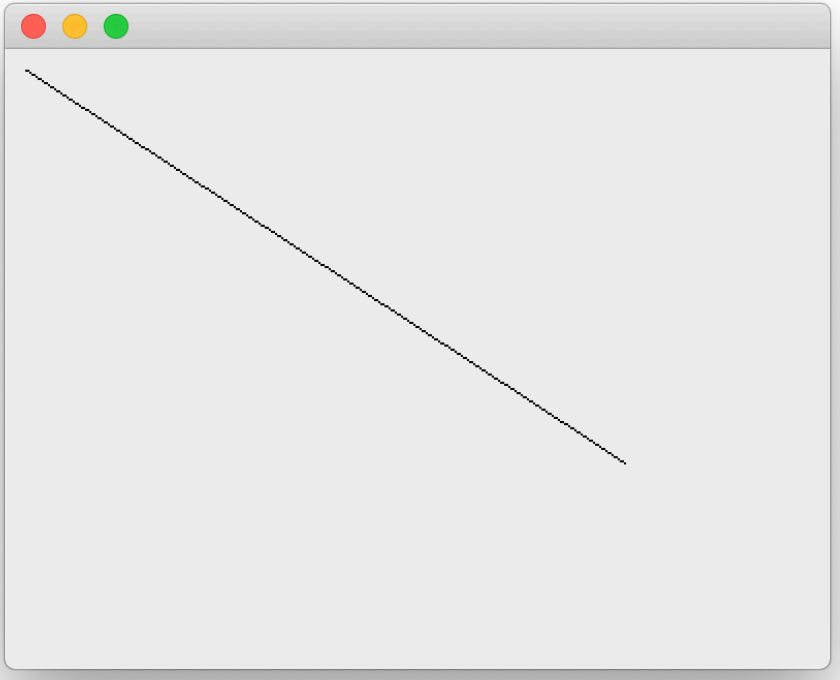
Bitmap drawing operations in Qt are handled through the `QPainter` class. This is a generic interface which can be used to draw on various *surfaces* including, for example, `QPixmap`. In this chapter we'll look at the `QPainter` drawing methods, first using primitive operations on a `QPixmap` surface, and then building a simple Paint application using what we've learnt.

To make this easy to demonstrate we'll be using the following stub application which handles creating our container (a `QLabel`) creating a pixmap canvas, setting that into the container and adding the container to the main window.

```
import sys
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel()
        canvas = QtGui.QPixmap(400, 300)
        self.label.setPixmap(canvas)
        self.setCentralWidget(self.label)
        self.draw_something()

    def draw_something(self):
        painter = QtGui.QPainter(self.label.pixmap())
        painter.drawLine(10, 10, 300, 200)
        painter.end()


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
```

Why do we use `QLabel` to draw on? The `QLabel` widget can also be used to show images, and it's the simplest widget available for displaying a `QPixmap`.

Save this to a file and run it and you should see the following — a single black line inside the window frame —



A single black line on the canvas

All the drawing occurs within the `draw_something` method — we create a `QPainter` instance, passing in the canvas (`self.label.pixmap()`) and then issue a command to draw a line. Finally we call `.end()` to close the painter and apply the changes.

You would usually also need to call `.update()` to trigger a refresh of the widget, but as we're drawing before the application window is shown a refresh is already going to occur automatically.

Drawing primitives

`QPainter` provides a huge number of methods for drawing shapes and lines on a bitmap surface (in 5.12 there are 192 `QPainter` specific non-event methods). The good news is that most of these are overloaded methods which are simply different ways of calling the same base methods.

For example, there are 5 different `drawLine` methods, all of which draw the same line, but differ in how the coordinates of what to draw are defined.

Method	Description
drawLine(const QLineF &*line*)	Draw a `QLineF` instance
drawLine(const QLine &*line*)	Draw a `QLine` instance
drawLine(int *x1*, int *y1*, int *x2*, int *y2*)	Draw a line between x1, y2 and x2, y2
drawLine(const QPoint &*p1*, const QPoint &*p2*)	Draw a line between `QPoint` 1 and `QPoint` 2
drawLine(const QPointF &*p1*, const QPointF &*p2*)	Draw a line between `QPointF` 1 and `QPointF` 2

If you're wondering what the difference is between a `QLine` and a `QLineF` , the latter has its coordinates specified as `float`. This is convenient if you have float positions as the result of other calculations, but otherwise not so much.

Ignoring the F-variants, we have 3 unique ways to draw a line — with a line object, with two sets of coordinates `(x1, y1), (x2, y2)` or with two `QPoint` objects. When you discover that a `QLine` itself is defined as `QLine(const QPoint & p1, const QPoint & p2)`or`QLine(int x1, int y1, int x2, int y2)`you see that they are all in fact, exactly the same thing. The different call signatures are simply there for convenience.

Given the x1, y1, x2, y2 coordinates, the two `QPoint`objects would be defined as `QPoint(x1, y1)` and `QPoint(x2, y2)`.

So, leaving out the duplicates we have the following draw operations —`drawArc` , `drawChord`, `drawConvexPolygon`, `drawEllipse`,`drawLine`, `drawPath`, `drawPie`, `drawPoint`, `drawPolygon`, `drawPolyline`, `drawRect`, `drawRects` and `drawRoundedRect`. To avoid get overwhelmed we'll focus first on the primitive shapes and lines first and return to the more complicated operations once we have the basics down.

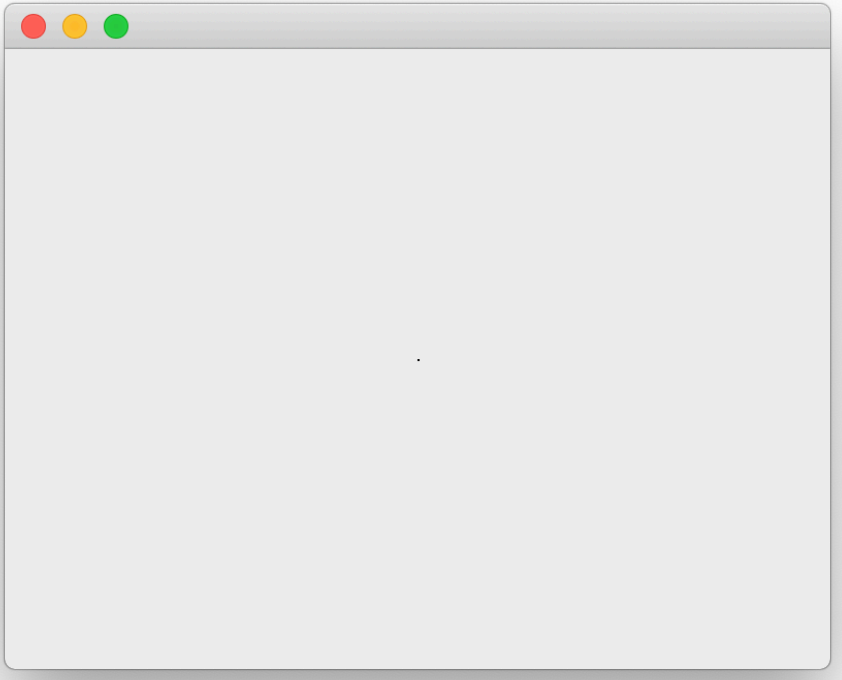
For each example, replace the `draw_something` method in your stub application and re-run it to see the output.

drawPoint

This draws a point, or *pixel* at a given point on the canvas.Each call to `drawPoint` draws one pixel. Replace your `draw_something` code with the following.

```
def draw_something(self):
    painter = QtGui.QPainter(self.label.pixmap())
    painter.drawPoint(200, 150)
    painter.end()
```

If you re-run the file you will see a window, but this time there is a single dot, in black in the middle of it. You'll probably need to move the window around to spot it.

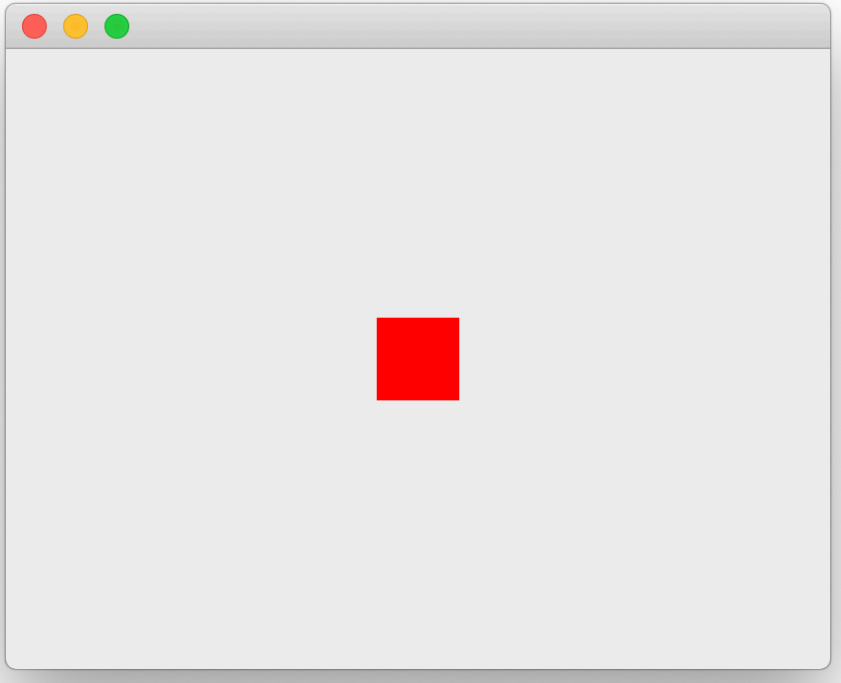


Drawing a single point (pixel) with QPainter

That really isn't much to look at. To make things more interesting we can change the colour and size of the point we're drawing. In PyQt the colour and thickness of lines is defined using the active *pen* on the QPainter. You can set this by creating a `QPen` instance and applying it.

```
    def draw_something(self):
        painter = QtGui.QPainter(self.label.pixmap())
        pen = QtGui.QPen()
        pen.setWidth(40)
        pen.setColor(QtGui.QColor('red'))
        painter.setPen(pen)
        painter.drawPoint(200, 150)
        painter.end()
```

This will give the following mildly more interesting result..

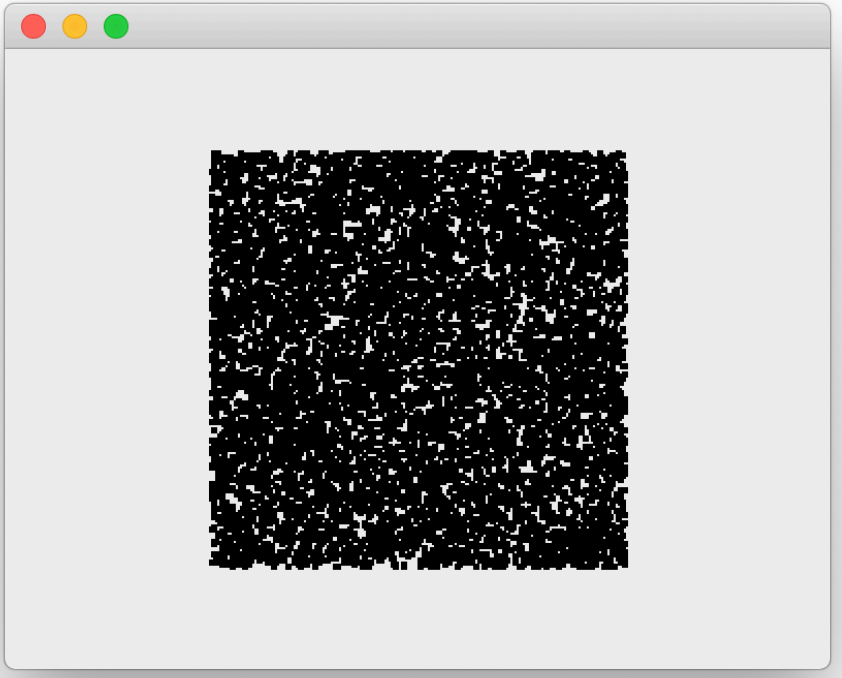


You are free to perform multiple draw operations with your `QPainter` until the painter is *ended*. Drawing onto the canvas is very quick — here we're drawing 10k dots at

random.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    painter.setPen(pen)

    for n in range(10000):
        painter.drawPoint(
            200+randint(-100, 100),  # x
            150+randint(-100, 100)   # y
            )
    painter.end()
```

The dots are 3 pixel-width and black (the default pen).

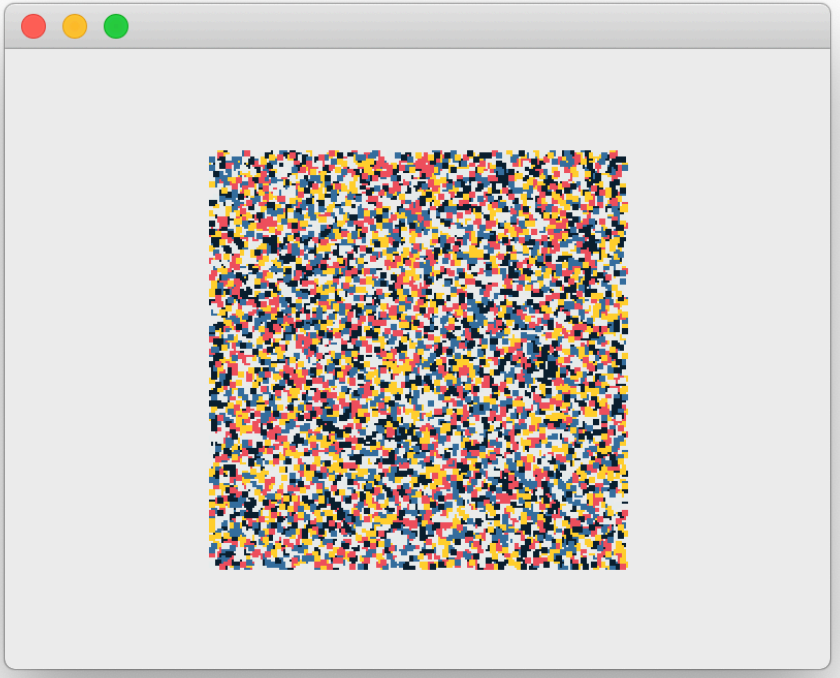


10k 3-pixel dots on a canvas

You will often want to update the current pen while drawing — e.g. to draw multiple points in different colours while keeping other characteristics (width) the same. To do this without recreating a new `QPen` instance each time you can get the current active pen from the `QPainter`using `pen = painter.pen()`. You can also re-apply an existing pen multiple times, changing it each time.

```
def draw_something(self):
    from random import randint, choice
    colors = ['#FFD141', '#376F9F', '#0D1F2D', '#E9EBEF', '#EB5160']

    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    painter.setPen(pen)

    for n in range(10000):
        # pen = painter.pen() you could get the active pen here
        pen.setColor(QtGui.QColor(choice(colors)))
        painter.setPen(pen)
        painter.drawPoint(
            200+randint(-100, 100),  # x
            150+randint(-100, 100)   # y
            )
    painter.end()
```

Will produce the following output —



Random pattern of 3 width dots

There can only ever be one `QPen` active on a `QPainter` — the current pen.

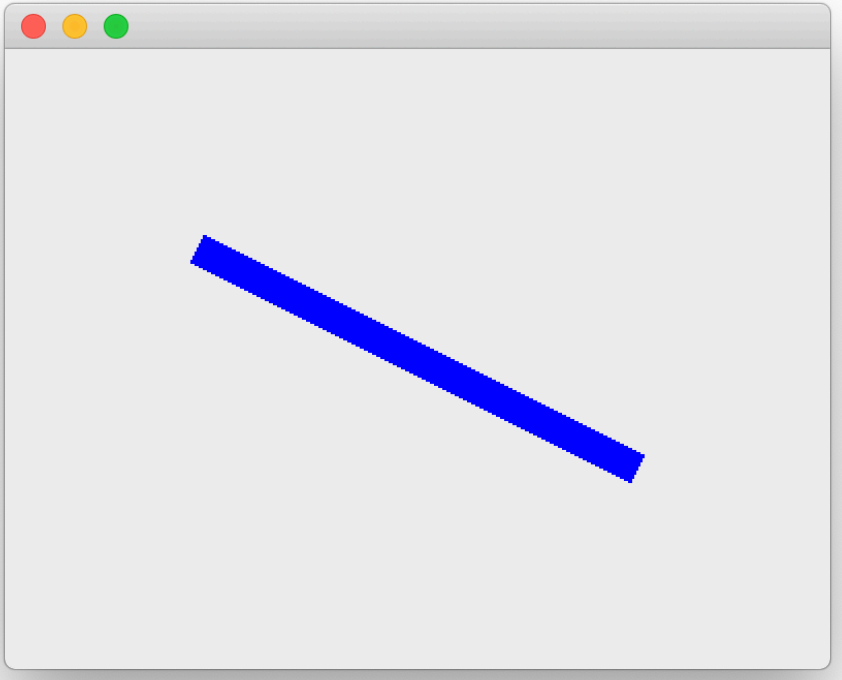
That's about as much excitement as you can have drawing dots onto a screen, so we'll move on to look at some other drawing operations.

drawLine

We already drew a line on the canvas at the beginning to test things are working. But what we didn't try was setting the pen to control the line appearance.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(15)
    pen.setColor(QtGui.QColor('blue'))
    painter.setPen(pen)
    painter.drawLine(
        QtCore.QPoint(100, 100),
        QtCore.QPoint(300, 200)
    )
    painter.end()
```

In this example we're also using `QPoint` to define the two points to connect with a line, rather than passing individual `x1, y1, x2, y2` parameters — remember that both methods are functionally identical.



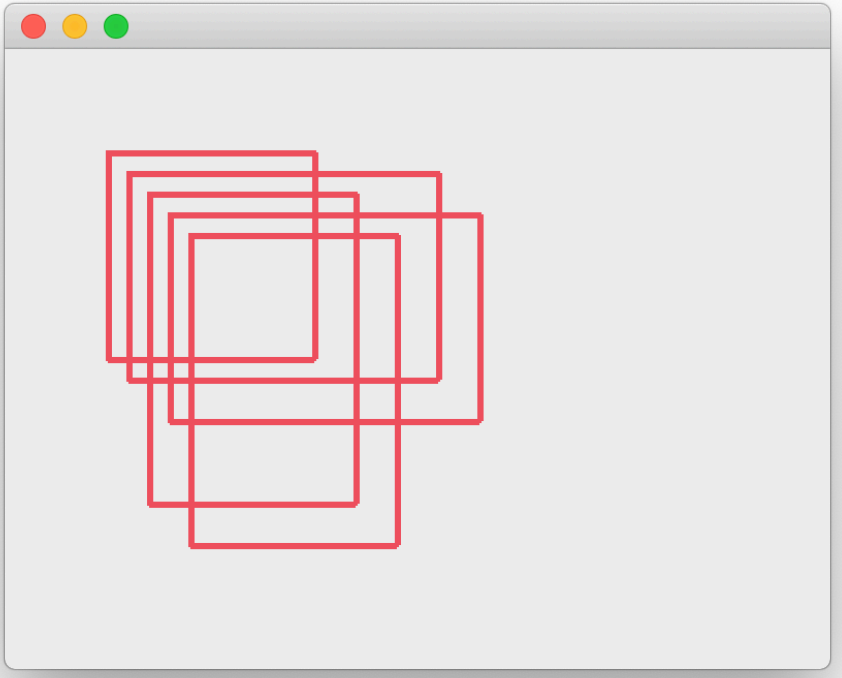
A thick blue line

drawRect, drawRects and drawRoundedRect

These functions all draw rectangles, defined by a series of points, or by `QRect` or `QRectF` instances.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    pen.setColor(QtGui.QColor("#EB5160"))
    painter.setPen(pen)
    painter.drawRect(50, 50, 100, 100)
    painter.drawRect(60, 60, 150, 100)
    painter.drawRect(70, 70, 100, 150)
    painter.drawRect(80, 80, 150, 100)
    painter.drawRect(90, 90, 100, 150)
    painter.end()
```

A square is just a rectangle with the same width and height



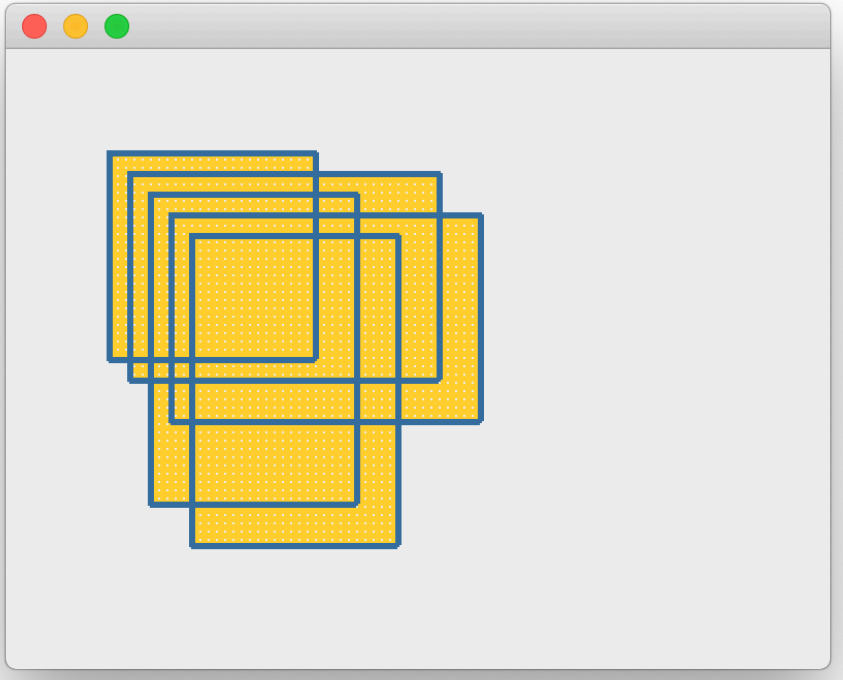
Drawing rectangles

You can also replace the multiple calls to `drawRect` with a single call to `drawRects` passing in multiple `QRect` objects. This will product exactly the same result.

```
painter.drawRects(
    QtCore.QRect(50, 50, 100, 100),
    QtCore.QRect(60, 60, 150, 100),
    QtCore.QRect(70, 70, 100, 150),
    QtCore.QRect(80, 80, 150, 100),
    QtCore.QRect(90, 90, 100, 150),
)
```

Drawn shapes can be filled in PyQt by setting the current active painter *brush*, passing in a `QBrush` instance to `painter.setBrush()`. The following example fills all rectangles with a patterned yellow colour.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    pen.setColor(QtGui.QColor("#376F9F"))
    painter.setPen(pen)

    brush = QtGui.QBrush()
    brush.setColor(QtGui.QColor("#FFD141"))
    brush.setStyle(Qt.Dense1Pattern)
    painter.setBrush(brush)

    painter.drawRects(
        QtCore.QRect(50, 50, 100, 100),
        QtCore.QRect(60, 60, 150, 100),
        QtCore.QRect(70, 70, 100, 150),
        QtCore.QRect(80, 80, 150, 100),
        QtCore.QRect(90, 90, 100, 150),
    )
    painter.end()
```

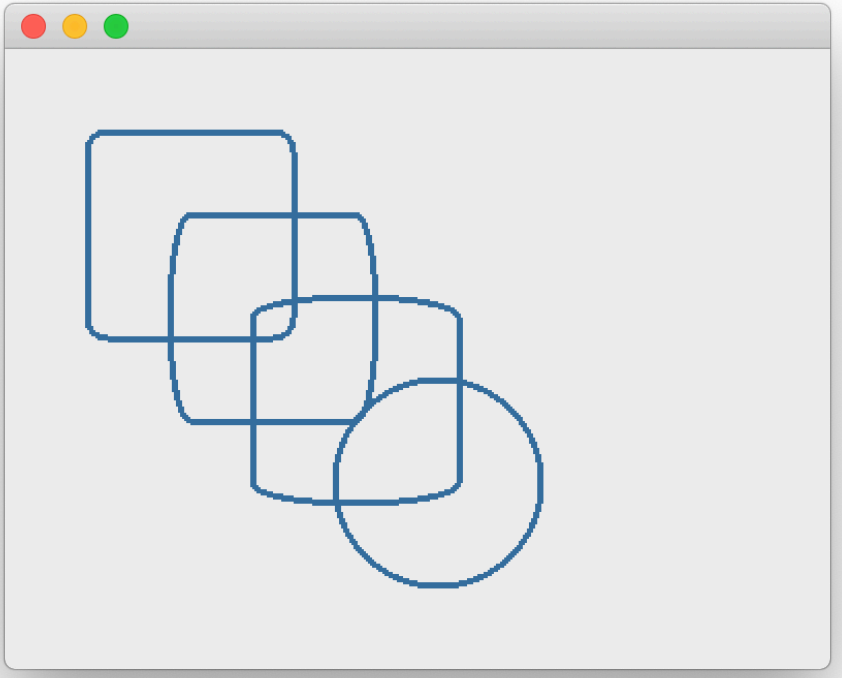


As for the pen, there is only ever one brush active on a given painter, but you can switch between them or change them while drawing. There are a number of brush style patterns available. You'll probably use `Qt.SolidPattern` more than any others though.

You *must* set a style to see any fill at all as the default is `Qt.NoBrush`.

The `drawRoundedRect` methods draw a rectangle, but with rounded edges, and so take two extra parameters for the x & y *radius* of the corners.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    pen.setColor(QtGui.QColor("#376F9F"))
    painter.setPen(pen)
    painter.drawRoundedRect(40, 40, 100, 100, 10, 10)
    painter.drawRoundedRect(80, 80, 100, 100, 10, 50)
    painter.drawRoundedRect(120, 120, 100, 100, 50, 10)
    painter.drawRoundedRect(160, 160, 100, 100, 50, 50)
    painter.end()
```



Rounded rectangles

There is an optional final parameter to toggle between the x & y ellipse radii of the corners being defined in absolute pixel terms `Qt.RelativeSize` (the default) or relative to the size of the rectangle (passed as a value 0...100). Pass `Qt.RelativeSize` to enable this.

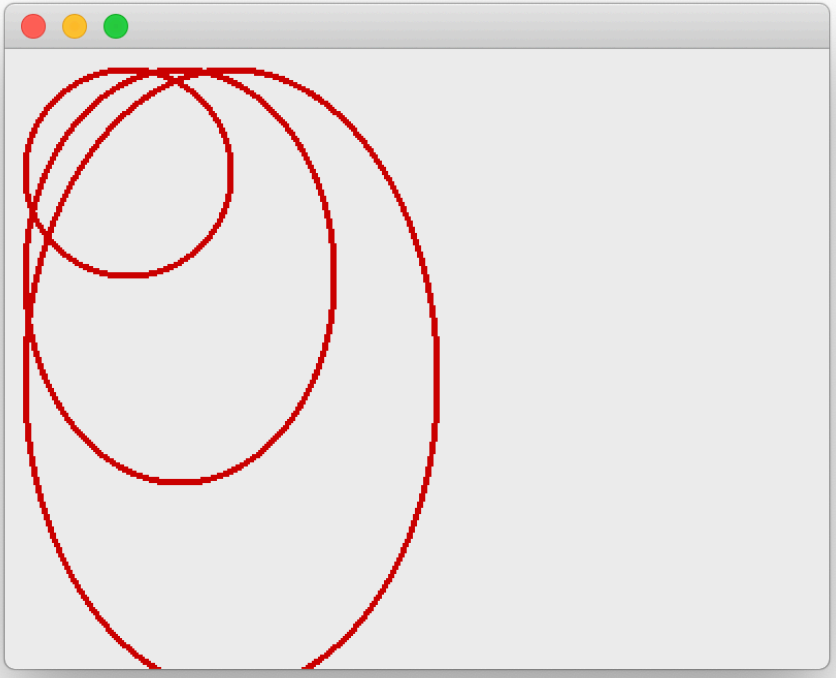
drawEllipse

The final primitive draw method we'll look at now is `drawEllipse` which can be used to draw an *ellipse* or a *circle*.

A circle is just an ellipse with an equal width and height.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())
    pen = QtGui.QPen()
    pen.setWidth(3)
    pen.setColor(QtGui.QColor(204,0,0))  # r, g, b
    painter.setPen(pen)

    painter.drawEllipse(10, 10, 100, 100)
    painter.drawEllipse(10, 10, 150, 200)
    painter.drawEllipse(10, 10, 200, 300)
    painter.end()
```

In this example `drawEllipse` is taking 4 parameters, with the first two being the x & y position of the *top left of the rectangle* in which the ellipse will be drawn, while the last two parameters are the width and height of that rectangle respectively.

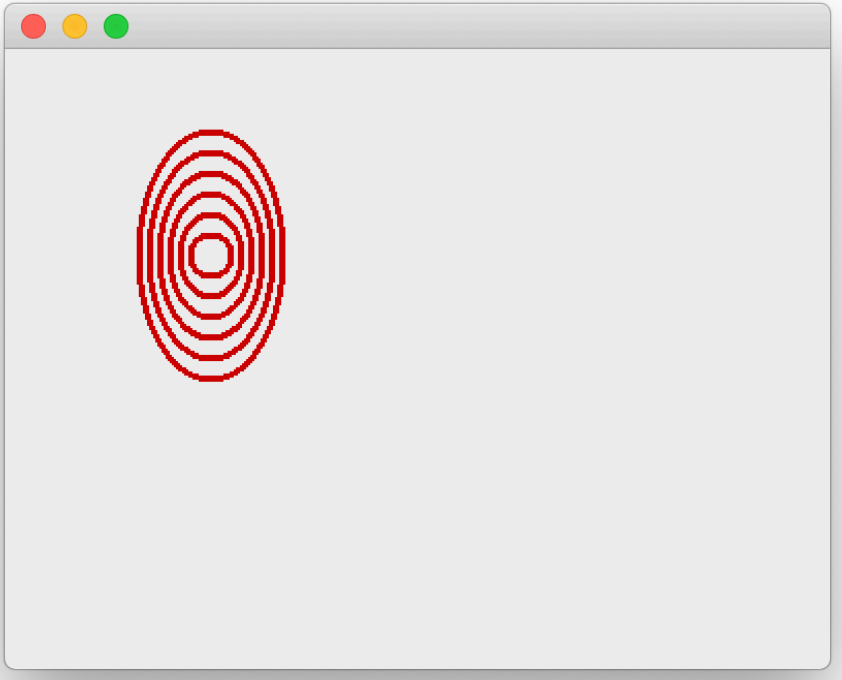


T>

You can achieve the same by passing in a `QRect`

There is another call signature which takes the *centre of the ellipse* as the first parameter, provided as `QPoint` or `QPointF` object, and then a x and y *radius*. The example below shows it in action.

```
painter.drawEllipse(QtCore.QPoint(100, 100), 10, 10)
painter.drawEllipse(QtCore.QPoint(100, 100), 15, 20)
painter.drawEllipse(QtCore.QPoint(100, 100), 20, 30)
painter.drawEllipse(QtCore.QPoint(100, 100), 25, 40)
painter.drawEllipse(QtCore.QPoint(100, 100), 30, 50)
painter.drawEllipse(QtCore.QPoint(100, 100), 35, 60)
```



Drawing an ellipse using Point & radius

You can fill ellipses using the same `QBrush` approach described for rectangles.

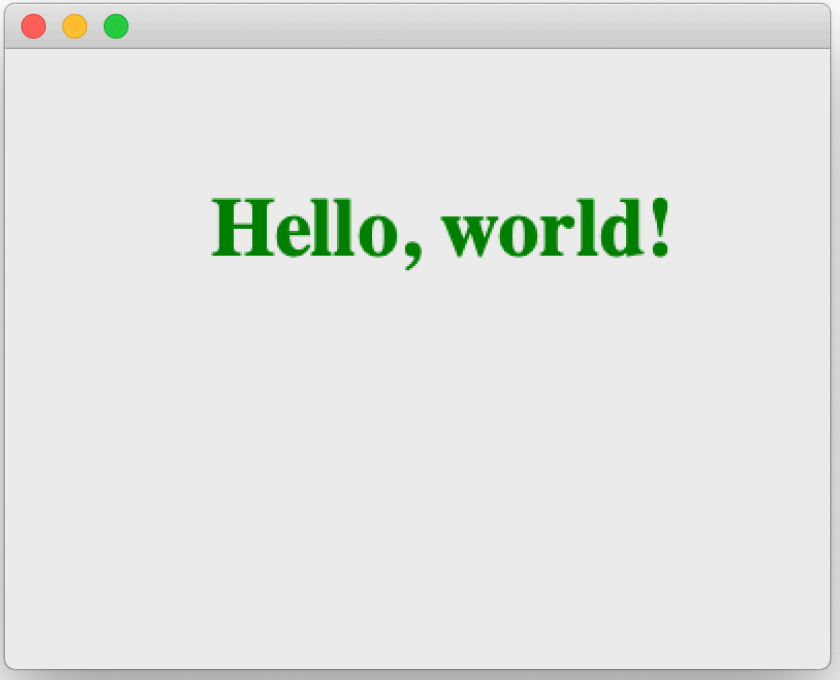
Text

Finally, we'll take a brief tour through the `QPainter` text drawing methods. To control the current font on a `QPainter` you use `setFont` passing in a `QFont` instance. With this you can control the family, weight and size (among other things) of the text you write. The colour of the text is still defined using the current pen however.

```
def draw_something(self):
    from random import randint
    painter = QtGui.QPainter(self.label.pixmap())

    pen = QtGui.QPen()
    pen.setWidth(1)
    pen.setColor(QtGui.QColor('green'))
    painter.setPen(pen)

    font = QtGui.QFont()
    font.setFamily('Times')
    font.setBold(True)
    font.setPointSize(40)
    painter.setFont(font)

    painter.drawText(100, 100, 'Hello, world!')
    painter.end()
```

You can also specify location with `QPoint` or `QPointF`.

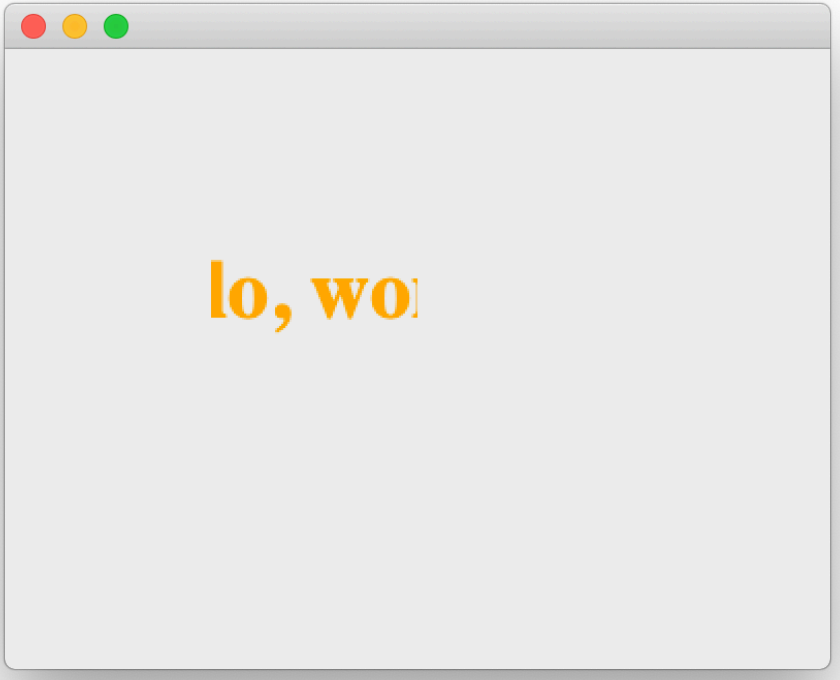
The width of the pen has no effect on the appearance of the text.



Bitmap text hello world example

There are also methods for drawing text within a specified area. Here the parameters define the x & y position and the width & height of the bounding box. Text outside this box is clipped (hidden). The 5th parameter *flags* can be used to control alignment of the text within the box among other things.

```
1  painter.drawText(100, 100, 100, 100, Qt.AlignHCenter, 'Hello, world!')
```



Bounding box clipped on drawText

You have complete control over the display of text by setting the active font on the painter via a `QFont` object. Check out the QFont documentation for more information.

A bit of fun with QPainter

The got a bit heavy, so let's take a breather and make something fun. So far we've been programmatically defining the draw operations to perform on the `QPixmap` surface. But we can just as easily draw in response to user input — for example allowing a user to scribble all over the canvas. Let's take what we've learned so far

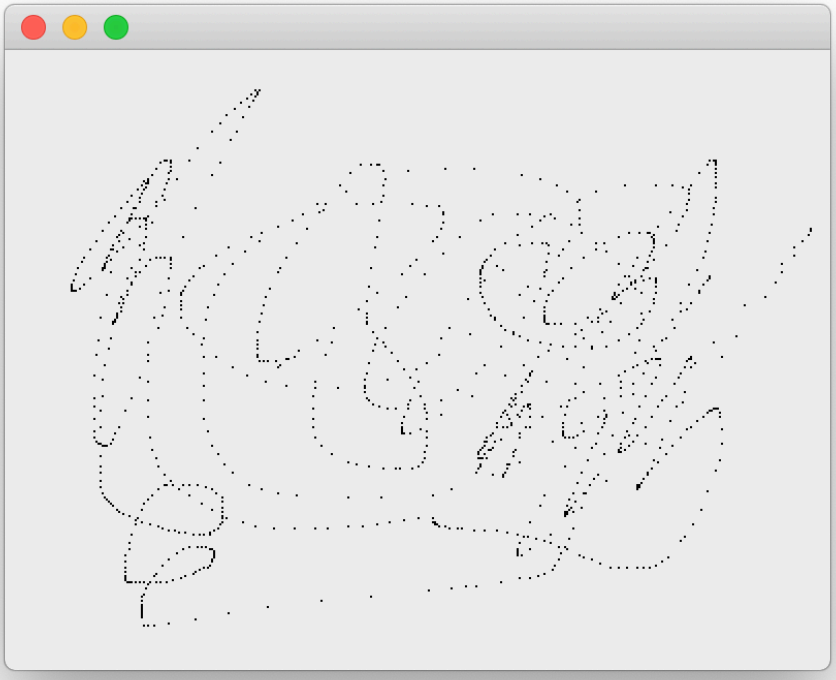
and use it to build a rudimentary Paint app.

We can start with the same simple application outline, adding a `mouseMoveEvent` handler to the `MainWindow` class in place of our draw method. Here we'll take the current position of the user's mouse and draw it to the canvas.

```
import sys
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel()
        canvas = QtGui.QPixmap(400, 300)
        self.label.setPixmap(canvas)
        self.setCentralWidget(self.label)

    def mouseMoveEvent(self, e):
        painter = QtGui.QPainter(self.label.pixmap())
        painter.drawPoint(e.x(), e.y())
        painter.end()
        self.update()


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
```

Why no click event? Widgets by default only receive mouse move events when a mouse button is pressed, unless *mouse tracking* is enabled. This can be configured using the `.setMouseTracking` method — setting this to `True` (it is `False` by default) will track the mouse continuously.

If you save this and run it you should be able to move your mouse over the screen and click to draw individual points. It should look something like this —



Drawing individual mouseMoveEvent points

The issue here is that when you move the mouse around quickly it actually jumps between locations on the screen, rather than moving smoothly from one place to the next. The `mouseMoveEvent`is fired for each location the mouse is in, but that's not enough to draw a continuous line, unless you move *very slowly*.

The solution to this is to draw *lines* instead of *points*. On each event we simply draw a line from where we were (previous `e.x()` and `e.y()`) to where we are now (current `e.x()` and `e.y()`). We can do this by tracking `last_x` and `last_y` ourselves.

We also need to *forget* the last position when releasing the mouse, or we'll start

drawing from that location again after moving the mouse across the page — i.e. we won't be able to break the line.

```
import sys
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel()
        canvas = QtGui.QPixmap(400, 300)
        self.label.setPixmap(canvas)
        self.setCentralWidget(self.label)

        self.last_x, self.last_y = None, None

    def mouseMoveEvent(self, e):
        if self.last_x is None: # First event.
            self.last_x = e.x()
            self.last_y = e.y()
            return # Ignore the first time.

        painter = QtGui.QPainter(self.label.pixmap())
        painter.drawLine(self.last_x, self.last_y, e.x(), e.y())
        painter.end()
        self.update()

        # Update the origin for next time.
        self.last_x = e.x()
        self.last_y = e.y()

    def mouseReleaseEvent(self, e):
        self.last_x = None
        self.last_y = None
```

```
app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
````

If you run this you should be able to scribble on the screen as you would expec\
t.

![Drawing with the mouse, using a continuous line](images/bitmap-draw.png)

It's still a bit dull, so let's add a simple palette to allow us to change the \
pen colour.

This requires a bit of re-architecting to ensure the mouse position is detected\
 accurately. So far we've using the `mouseMoveEvent` on the `QMainWindow` . Wh\
en we only have a single widget in the window this is fine – as long as you don\
't resize the window, the coordinates of the container and the single nested wi\
dget line up. However, if we add other widgets to the layout this won't hold – \
the coordinates of the `QLabel` will be offset from the window, and we'll be dr\
awing in the wrong location.

This is easily fixed by moving the mouse handling onto the `QLabel` itself– it'\
s event coordinates are always relative to itself. This we wrap up as an indivi\
dual `Canvas` object, which handles the creation of the pixmap surface, sets up\
 the x & y locations and the holds the current pen colour (set to black by defa\
ult).

T> This self-contained `Canvas` is a drop-in drawable surface you could use in \
your own apps.

```python
import sys
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt

class Canvas(QtWidgets.QLabel):
```

```
75
76      def __init__(self):
77          super().__init__()
78          pixmap = QtGui.QPixmap(600, 300)
79          self.setPixmap(pixmap)
80
81          self.last_x, self.last_y = None, None
82          self.pen_color = QtGui.QColor('#000000')
83
84      def set_pen_color(self, c):
85          self.pen_color = QtGui.QColor(c)
86
87      def mouseMoveEvent(self, e):
88          if self.last_x is None: # First event.
89              self.last_x = e.x()
90              self.last_y = e.y()
91              return # Ignore the first time.
92
93          painter = QtGui.QPainter(self.pixmap())
94          p = painter.pen()
95          p.setWidth(4)
96          p.setColor(self.pen_color)
97          painter.setPen(p)
98          painter.drawLine(self.last_x, self.last_y, e.x(), e.y())
99          painter.end()
100         self.update()
101
102         # Update the origin for next time.
103         self.last_x = e.x()
104         self.last_y = e.y()
105
106     def mouseReleaseEvent(self, e):
107         self.last_x = None
108         self.last_y = None
```

For the colour selection we're going to build a custom widget, based off `QPushButton`. This widget accepts a `color` parameter which can be a `QColour` instance, or a colour name ('red', 'black') or hex value. This colour is set on the background of the widget

to make it identifiable. We can use the standard `QPushButton.pressed` signal to hook it up to any actions.

```
COLORS = [
# 17 undertones https://lospec.com/palette-list/17undertones
'#000000', '#141923', '#414168', '#3a7fa7', '#35e3e3', '#8fd970', '#5ebb49',
'#458352', '#dcd37b', '#fffee5', '#ffd035', '#cc9245', '#a15c3e', '#a42f3b',
'#f45b7a', '#c24998', '#81588d', '#bcb0c2', '#ffffff',
]


class QPaletteButton(QtWidgets.QPushButton):

    def __init__(self, color):
        super().__init__()
        self.setFixedSize(QtCore.QSize(24,24))
        self.color = color
        self.setStyleSheet("background-color: %s;" % color)
```

With those two new parts defined, we simply need to iterate over our list of colours, create a `QPaletteButton` passing in the colour, connect its pressed signal to the `set_pen_color` handler on the canvas (indirectly through a `lambda` to pass the additional colour data) and add it to the palette layout.

```
class MainWindow(QtWidgets.QMainWindow):

    def __init__(self):
        super().__init__()

        self.canvas = Canvas()

        w = QtWidgets.QWidget()
        l = QtWidgets.QVBoxLayout()
        w.setLayout(l)
        l.addWidget(self.canvas)

        palette = QtWidgets.QHBoxLayout()
```

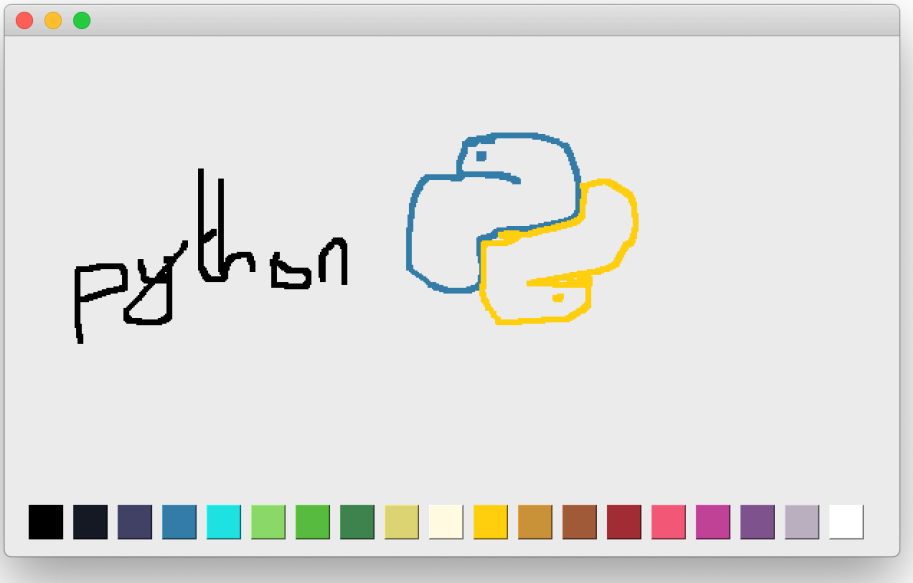
```
        self.add_palette_buttons(palette)
        l.addLayout(palette)

        self.setCentralWidget(w)

    def add_palette_buttons(self, layout):
        for c in COLORS:
            b = QPaletteButton(c)
            b.pressed.connect(lambda c=c: self.canvas.set_pen_color(c))
            layout.addWidget(b)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
```

This should give you a fully-functioning multicolour paint application, where you can draw lines on the canvas and select colours from the palette.



Unfortunately, it doesn't make you good.

Unfortunately, it doesn't make you a good artist.

Spray

For a final bit of fun you can switch out the `mouseMoveEvent` with the following to draw with a "spray can" effect instead of a line. This is simulated using `random.gauss` to generate a series of *normally distributed* dots around the current mouse position which we plot with `drawPoint`.

```
    def mouseMoveEvent(self, e):
        painter = QtGui.QPainter(self.pixmap())
        p = painter.pen()
        p.setWidth(1)
        p.setColor(self.pen_color)
        painter.setPen(p)

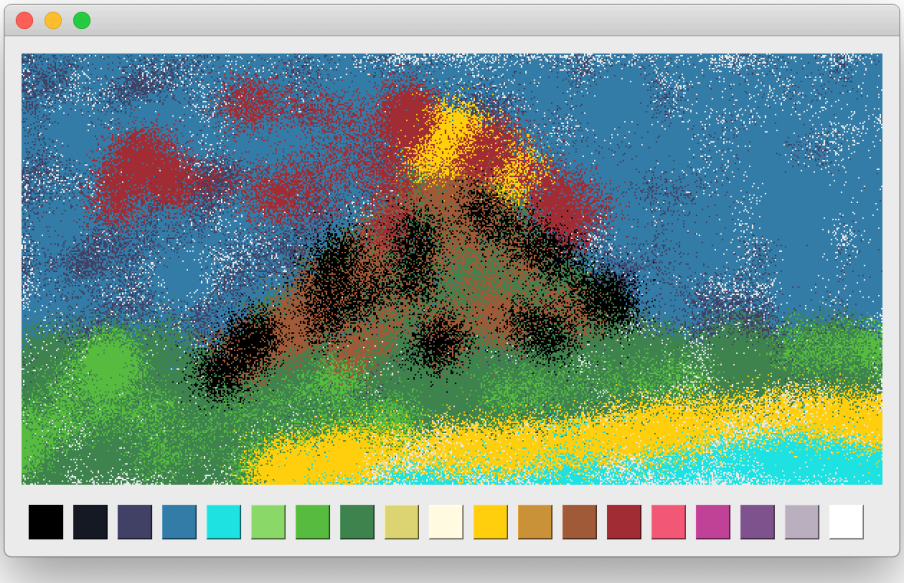
        for n in range(SPRAY_PARTICLES):
            xo = random.gauss(0, SPRAY_DIAMETER)
            yo = random.gauss(0, SPRAY_DIAMETER)
            painter.drawPoint(e.x() + xo, e.y() + yo)

        self.update()
```

Define the `SPRAY_PARTICLES` and `SPRAY_DIAMETER` variables at the top of your file and import the `random` standard library module. The image below shows the spray behaviour when using the following settings:

```
import random

SPRAY_PARTICLES = 100
SPRAY_DIAMMETER = 10
```



Just call me Picasso

Just call me Picasso.

N> For the spray can we don't need to track the previous position, as we always spray around the current point.

If you want a challenge, you could try adding an additional button to toggle between draw and spray mode, or an input to define the brush/spray diameter.

For a fully-functional drawing program written with PyQt5 check out my 15 Minute App "Piecasso" available here: https://github.com/mfitzp/15-minute-apps/tree/master/paint

This introduction should have given you a good idea of what you can do with `QPainter`. As described, this system is the basis of all widget drawing. If you want to look further, check out the widget `.paint()` method, which receives a `QPainter` instance, to allow the widget to draw on itself. The same methods you've learnt here can be used in `.paint()` to draw some basic custom widgets.

Creating Custom Widgets

In the previous chapter we introduced `QPainter` and looked at some basic bitmap drawing operations which you can used to draw dots, lines, rectangles and circles on a `QPainter` *surface* such as a `QPixmap`.

This process of *drawing on a surface* with `QPainter` is in fact the basis by which all widgets in Qt are drawn.Now you know how to use `QPainter` you know how to draw your own custom widgets!

In this chapter we'll take what we've learnt so far and use it to construct a completely new *custom* widget. For a working example we'll be building the following widget — a customisable PowerBar meter with a dial control.

PowerBar-meter

This widget is actually a mix of a *compound widget* and *custom widget* in that we are using the built-in Qt `QDial` component for the dial, while drawing the power bar ourselves. We then assemble these two parts together into a parent widget which can be dropped into place seamlessly in any application, without needing to know how it's put together. The resulting widget provides the common `QAbstractSlider` interface with some additions for configuring the bar display.

After following this example you will be able to build your very own custom widgets — whether they are compounds of built-ins or completely novel self-drawn wonders.

Getting started

As we've previously seen compound widgets are simply widgets with a layout applied, which itself contains >1 other widget. The resulting "widget" can then be used as any other, with the internals hidden/exposed as you like.

The outline for our *PowerBar* widget is given below — we'll build our custom widget up gradually from this outline stub.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import Qt


class _Bar(QtWidgets.QWidget):
    pass

class PowerBar(QtWidgets.QWidget):
    """
    Custom Qt Widget to show a power bar and dial.
    Demonstrating compound and custom-drawn widget.
    """

    def __init__(self, steps=5, *args, **kwargs):
        super(PowerBar, self).__init__(*args, **kwargs)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar()
        layout.addWidget(self._bar)

        self._dial = QtWidgets.QDial()
        layout.addWidget(self._dial)

        self.setLayout(layout)
```

This simply defines our custom power bar is defined in the `_Bar` object — here just unaltered subclass of `QWidget`. The `PowerBar` widget (which is the complete widget) combines this, using a `QVBoxLayout` with the built in `QDial` to display them together.

Save this to a file named `power_bar.py`

We also need a little demo application to display the widget.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from power_bar import PowerBar


app = QtWidgets.QApplication([])
volume = PowerBar()
volume.show()
app.exec_()
```

N> We don't need to create a `QMainWindow` since any widget without a parent is a window in it's own right. Our custom `PowerBar` widget will appear as any normal window.

This is all you need, just save it in the same folder as the previous file, under something like `demo.py`. You can run this file at any time to see your widget in action. Run it now and you should see something like this:

PowerBar-dial

If you stretch the window down you'll see the dial has more space above it than below — this is being taken up by our (currently invisible) `_Bar` widget.

paintEvent

The `paintEvent` handler is the core of all widget drawing in PyQt.

Every complete and partial re-draw of a widget is triggered through a `paintEvent` which the widget handles to draw itself. A `paintEvent` can be triggered by —

- repaint() or update() was called
- the widget was obscured and has now been uncovered
- the widget has been resized

— but it can also occur for many other reasons. What is important is that when a `paintEvent` is triggered your widget is able to redraw it.

If a widget is simple enough (like ours is) you can often get away with simply redrawing the entire thing any time *anything* happens. But for more complicated widgets this can get very inefficient. For these cases the `paintEvent` includes the specific region that needs to be updated. We'll make use of this in later, more complicated examples.

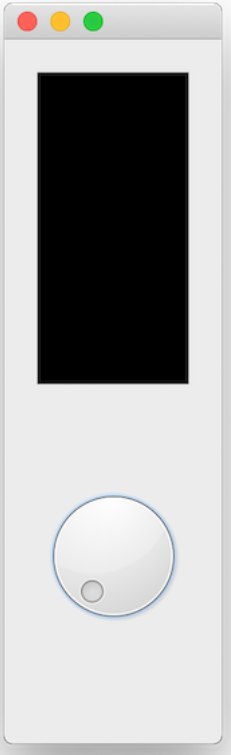
For now we'll do something very simple, and just fill the entire widget with a single colour. This will allow us to see the area we're working with to start drawing the bar.

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)
        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor('black'))
        brush.setStyle(Qt.SolidPattern)
        rect = QtCore.QRect(0, 0, painter.device().width(), painter.device().he\
ight())
        painter.fillRect(rect, brush)
```

Positioning

Now we can see the `_Bar` widget we can tweak its positioning and size. If you drag around the shape of the window you'll see the two widgets changing shape to fit

the space available. This is what we want, but the `QDial` is also expanding vertically more than it should, and leaving empty space we could use for the bar.



PowerBar-stretch

We can use `setSizePolicy` on our `_Bar` widget to make sure it expands as far as possible. By using the `QSizePolicy.MinimumExpanding` the provided `sizeHint` will be used as a minimum, and the widget will expand as much as possible.

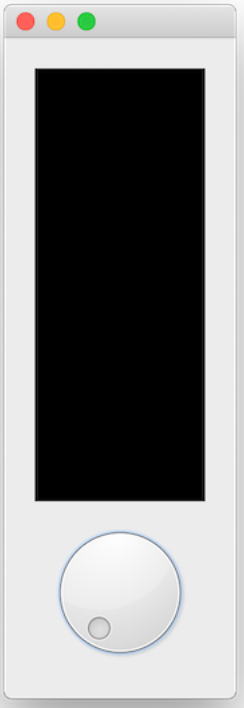
```
class _Bar(QtWidgets.QWidget):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setSizePolicy(
            QtWidgets.QSizePolicy.MinimumExpanding,
            QtWidgets.QSizePolicy.MinimumExpanding
        )

    def sizeHint(self):
        return QtCore.QSize(40,120)
```

It's still not *perfect* as the `QDial` widget resizes itself a bit awkwardly, but our bar is now expanding to fill all the available space.



PowerBar-policy

With the positioning sorted we can now move on to define our paint methods to draw our PowerBar meter in the top part (currently black) of the widget.

Updating the display

We now have our canvas completely filled in black, next we'll use `QPainter` draw commands to actually draw something on the widget.

Before we start on the bar, we've got a bit of testing to do to make sure we can update the display with the values of our dial. Update the `paintEvent`with the following code.

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor('black'))
        brush.setStyle(Qt.SolidPattern)
        rect = QtCore.QRect(0, 0, painter.device().width(), painter.device().he\
ight())
        painter.fillRect(rect, brush)

        # Get current state.
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        pen = painter.pen()
        pen.setColor(QtGui.QColor('red'))
        painter.setPen(pen)

        font = painter.font()
        font.setFamily('Times')
        font.setPointSize(18)
        painter.setFont(font)

        painter.drawText(25, 25, "{}-->{}<--{}".format(vmin, value, vmax))
        painter.end()
```

This draws the black background as before, then uses `.parent()` to access our parent `PowerBar` widget and through that the `QDial` via `_dial`. From there we get the current value, as well as the allowed range minimum and maximum values. Finally we draw those using the painter, just like we did in the previous part.

We're leaving handling of the current value, min and max values to the `QDial` here, but we could also store that value ourselves and use signals to/from the dial to keep things in sync.

Run this, wiggle the dial around and .....nothing happens. Although we've defined the `paintEvent` handler we're not triggering a repaint when the dial changes.

You can force a refresh by resizing the window, as soon as you do this you should see the text appear. Neat, but terrible UX — "just resize your app to see your settings!"

To fix this we need to hook up our `_Bar`widget to repaint itself in response to changing values on the dial. We can do this using the `QDial.valueChanged`signal, hooking it up to a custom slot method which calls `.refresh()` — triggering a full-repaint.

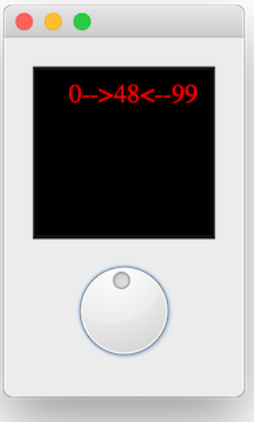
Add the following method to the `_Bar` widget.

```
    def _trigger_refresh(self):
        self.update()
```

...and add the following to the `__init__` block for the parent `PowerBar` widget.

```
self._dial.valueChanged.connect(self._bar._trigger_refresh)
```

If you re-run the code now, you will see the display updating automatically as you turn the dial (click and drag with your mouse). The current value is displayed as text.



PowerBar-text

Drawing the bar

Now we have the display updating and displaying the current value of the dial, we can move onto drawing the actual bar display. This is a little complicated, with a bit of maths to calculate bar positions, but we'll step through it to make it clear what's going on.

The sketch below shows what we are aiming for — a series of *N* boxes, inset from the edges of the widget, with spaces between them.

power-goal

Calculating what to draw

The number of boxes to draw is determined by the current value — and how far along it is between the minimum and maximum value configured for the `QDial`. We already have that information in the example above.

```
dial = self.parent()._dial
vmin, vmax = dial.minimum(), dial.maximum()
value = dial.value()
```

If `value` is half way between `vmin` and `vmax` then we want to draw half of the boxes (if we have 4 boxes total, draw 2). If `value` is at `vmax` we want to draw them all.

To do this we first convert our `value` into a number between 0 and 1, where `0 = vmin` and `1 = vmax`. We first subtract `vmin` from `value` to adjust the range of possible values to start from zero — i.e. from `vmin...vmax` to `0…(vmax-vmin)`. Dividing this value by `vmax-vmin` (the new maximum) then gives us a number between 0 and 1.

The trick then is to multiply this value (called `pc` below) by the number of steps and that gives us a number between 0 and 5 — the number of boxes to draw.

```
pc = (value - vmin) / (vmax - vmin)
n_steps_to_draw = int(pc * 5)
```

We're wrapping the result in `int` to convert it to a whole number (rounding down) to remove any partial boxes.

Update the `drawText` method in your paint event to write out this number instead.

```
pc = (value - vmin) / (vmax - vmin)
n_steps_to_draw = int(pc * 5)
painter.drawText(25, 25, "{}".format(n_steps_to_draw))
```

As you turn the dial you will now see a number between 0 and 5.

Drawing boxes

Next we want to convert this number 0...5 to a number of bars drawn on the canvas. Start by removing the `drawText` and font and pen settings, as we no longer need those.

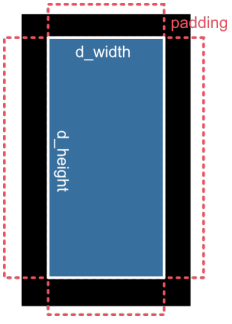
To draw accurately we need to know the size of our canvas — i.e the size of the widget. We will also add a bit of padding around the edges to give space around the edges of the blocks against the black background.

All measurements in the `QPainter` are in pixels.

```
        padding = 5

        # Define our canvas.
        d_height = painter.device().height() - (padding * 2)
        d_width = painter.device().width() - (padding * 2)
```

We take the height and width and subtract `2 * padding` from each — it's 2x because we're padding both the left and right (and top and bottom) edges. This gives us our resulting *active canvas* area in `d_height` and `d_width`.



power-padding

We need to break up our `d_height` into 5 equal parts, one for each block — we can calculate that height simply by `d_height / 5`. Additionally, since we want spaces

between the blocks we need to calculate how much of this step size is taken up by space (top and bottom, so halved) and how much is actual block.

```
step_size = d_height / 5
bar_height = step_size * 0.6
bar_spacer = step_size * 0.4 / 2
```

These values are all we need to draw our blocks on our canvas. To do this we count up to the number of steps-1 starting from 0 using `range` and then draw a `fillRect` over a region for each block.

```
brush.setColor(QtGui.QColor('red'))

for n in range(5):
    rect = QtCore.QRect(
        padding,
        padding + d_height - ((n+1) * step_size) + bar_spacer,
        d_width,
        bar_height
    )
painter.fillRect(rect, brush)
```

N> The fill is set to a red brush to begin with but we will customise this later.

The box to draw with `fillRect` is defined as a `QRect` object to which we pass, in turn, the left x, top y, width and height.

The *width* is the full canvas width minus the padding, which we previously calculated and stored in `d_width`. The *left x* is similarly just the `padding` value (5px) from the left hand side of the widget.

The *height* of the bar `bar_height`we calculated as 0.6 times the `step_size`.

This leaves parameter 2 `d_height - ((1 + n) * step_size) + bar_spacer` which gives the *top y* position of the rectangle to draw. This is the only calculation that changes as we draw the blocks.

A key fact to remember here is that *y* coordinates in `QPainter` start at the top and increase down the canvas. This means that plotting at `d_height` will be plotting at

the very bottom of the canvas. When we draw a rectangle from a point it is drawn to the *right* and *down* from the starting position.

To draw a block at the very bottom we must start drawing at `d_height-step_size` i.e. one block up to leave space to draw downwards.

In our bar meter we're drawing blocks, in turn, starting at the bottom and working upwards. So our very first block must be placed at `d_height-step_size` and the second at `d_height-(step_size*2)`. Our loop iterates from 0 upwards, so we can achieve this with the following formula —

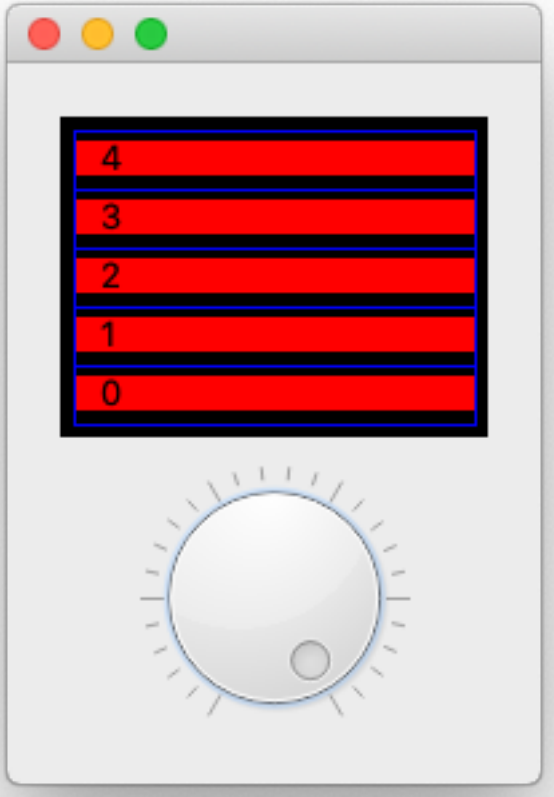
```
d_height - ((1 + n) * step_size
```

The final adjustment is to account for our blocks only taking up part of each `step_size` (currently 0.6). We add a little padding to move the block away from the edge of the box and into the middle, and finally add the padding for the bottom edge. That gives us the final formula —

```
padding + d_height - ((n+1) * step_size) + bar_spacer,
```

This produces the following layout.

In the picture below the current value of `n` has been printed over the box, and a blue box has been drawn around the complete `step_size` so you can see the padding and spacers in effect.



PowerBar-spacer

Putting this all together gives the following code, which when run will produce a working power-bar widget with blocks in red. You can drag the wheel back and forth and the bars will move up and down in response.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import Qt

class _Bar(QtWidgets.QWidget):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setSizePolicy(
            QtWidgets.QSizePolicy.MinimumExpanding,
            QtWidgets.QSizePolicy.MinimumExpanding
        )

    def sizeHint(self):
        return QtCore.QSize(40,120)

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor('black'))
        brush.setStyle(Qt.SolidPattern)
        rect = QtCore.QRect(0, 0, painter.device().width(), painter.device().he\
ight())
        painter.fillRect(rect, brush)

        # Get current state.
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        padding = 5

        # Define our canvas.
        d_height = painter.device().height() - (padding * 2)
        d_width = painter.device().width() - (padding * 2)

        # Draw the bars.
        step_size = d_height / 5
```

```
40          bar_height = step_size * 0.6
41          bar_spacer = step_size * 0.4 / 2
42
43          pc = (value - vmin) / (vmax - vmin)
44          n_steps_to_draw = int(pc * 5)
45          brush.setColor(QtGui.QColor('red'))
46          for n in range(n_steps_to_draw):
47              rect = QtCore.QRect(
48                  padding,
49                  padding + d_height - ((n+1) * step_size) + bar_spacer,
50                  d_width,
51                  bar_height
52              )
53              painter.fillRect(rect, brush)
54
55          painter.end()
56
57      def _trigger_refresh(self):
58          self.update()
59
60
61  class PowerBar(QtWidgets.QWidget):
62      """
63      Custom Qt Widget to show a power bar and dial.
64      Demonstrating compound and custom-drawn widget.
65      """
66
67      def __init__(self, steps=5, *args, **kwargs):
68          super(PowerBar, self).__init__(*args, **kwargs)
69
70          layout = QtWidgets.QVBoxLayout()
71          self._bar = _Bar()
72          layout.addWidget(self._bar)
73
74          self._dial = QtWidgets.QDial()
75          self._dial.valueChanged.connect(
76             self._bar._trigger_refresh
77          )
78
```

```
79          layout.addWidget(self._dial)
80          self.setLayout(layout)
```

PowerBar-basic

That already does the job, but we can go further to provide more customisation, add some UX improvements and improve the API for working with our widget.

Customising the Bar

We now have a working power bar, controllable with a dial. But it's nice when creating widgets to provide options to configure the behaviour of your widget to make it more flexible. In this part we'll add methods to set customisable numbers of segments, colours, padding and spacing.

The elements we're going to provide customisation of are as follows —

Option	Description
number of bars	How many bars are displayed on the widget
colours	Individual colours for each of the bars
background colour	The colour of the draw canvas (default black)
padding	Space around the widget edge, between bars and edge of canvas.
bar height / bar percent	Proportion (0...1) of the bar which is solid (the rest will be spacing between adjacent bars)

We can store each of these as attributes on the `_bar` object, and use them from the `paintEvent` method to change its behaviour.

The `_Bar.__init__` is updated to accept an initial argument for either the number of bars (as an integer) or the colours of the bars (as a list of `QColor`, hex values or names). If a number is provided, all bars will be coloured red. If the a list of colours is provided the number of bars will be determined from the length of the colour list. Default values for`self._bar_solid_percent`, `self._background_color`, `self._padding` are also set.

```
class _Bar(QtWidgets.QWidget):
    clickedValue = QtCore.pyqtSignal(int)

    def __init__(self, steps, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setSizePolicy(
            QtWidgets.QSizePolicy.MinimumExpanding,
            QtWidgets.QSizePolicy.MinimumExpanding
        )

        if isinstance(steps, list):
            # list of colours.
            self.n_steps = len(steps)
            self.steps = steps

        elif isinstance(steps, int):
            # int number of bars, defaults to red.
            self.n_steps = steps
            self.steps = ['red'] * steps

        else:
            raise TypeError('steps must be a list or int')

        self._bar_solid_percent = 0.8
        self._background_color = QtGui.QColor('black')
        self._padding = 4.0  # n-pixel gap around edge.
```

Likewise we update the `PowerBar.__init__` to accept the steps parameter, and pass it through.

```
class PowerBar(QtWidgets.QWidget):
    def __init__(self, steps=5, *args, **kwargs):
        super().__init__(*args, **kwargs)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar(steps)

        #...continued as before.
```

We now have the parameters in place to update the `paintEvent` method. The modified code is shown below.

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(self._background_color)
        brush.setStyle(Qt.SolidPattern)
        rect = QtCore.QRect(0, 0, painter.device().width(), painter.device().he\
ight())
        painter.fillRect(rect, brush)

        # Get current state.
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        value = parent.value()

        # Define our canvas.
        d_height = painter.device().height() - (self._padding * 2)
        d_width = painter.device().width() - (self._padding * 2)

        # Draw the bars.
        step_size = d_height / self.n_steps
        bar_height = step_size * self._bar_solid_percent
        bar_spacer = step_size * (1 - self._bar_solid_percent) / 2

        # Calculate the y-stop position, from the value in range.
        pc = (value - vmin) / (vmax - vmin)
```

```
27          n_steps_to_draw = int(pc * self.n_steps)
28
29          for n in range(n_steps_to_draw):
30              brush.setColor(QtGui.QColor(self.steps[n]))
31              rect = QtCore.QRect(
32                  self._padding,
33                  self._padding + d_height - ((1 + n) * step_size) + bar_spacer,
34                  d_width,
35                  bar_height
36              )
37              painter.fillRect(rect, brush)
38
39          painter.end()
```

You can now experiment with passing in different values for the init to `PowerBar`, e.g. increasing the number of bars, or providing a colour list. Some examples are shown below — a good source of hex palettes is the Bokeh source.

```
PowerBar(10)
PowerBar(3)
PowerBar(["#5e4fa2", "#3288bd", "#66c2a5", "#abdda4", "#e6f598", "#ffffbf", "#f\
ee08b", "#fdae61", "#f46d43", "#d53e4f", "#9e0142"])
PowerBar(["#a63603", "#e6550d", "#fd8d3c", "#fdae6b", "#fdd0a2", "#feedde"])
```

power-examples

You could fiddle with the padding settings through the variables e.g. `self._bar_solid_percent` but it'd be nicer to provide proper methods to set these.

N> We're following the Qt standard of camelCase method names for these external methods for consistency with the others inherited from `QDial`.

```
    def setColor(self, color):
        self._bar.steps = [color] * self._bar.n_steps
        self._bar.update()

    def setColors(self, colors):
        self._bar.n_steps = len(colors)
        self._bar.steps = colors
        self._bar.update()

    def setBarPadding(self, i):
        self._bar._padding = int(i)
        self._bar.update()

    def setBarSolidPercent(self, f):
        self._bar._bar_solid_percent = float(f)
        self._bar.update()

    def setBackgroundColor(self, color):
        self._bar._background_color = QtGui.QColor(color)
        self._bar.update()
```

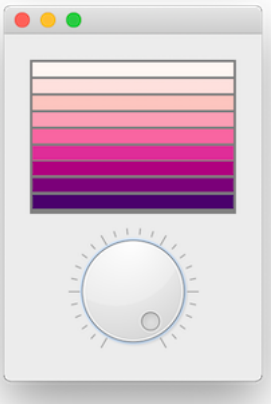
In each case we set the private variable on the `_bar` object and then call `_bar.update()` to trigger a redraw of the widget. The method support changing the colour to a single colour, or updating a list of them — setting a list of colours can also be used to change the number of bars.

N> There is no method to set the bar count, since expanding a list of colours would be faffy. But feel free to try adding this yourself!

Here's an example using 25px padding, a fully solid bar and a grey background.

```
bar = PowerBar(["#49006a", "#7a0177", "#ae017e", "#dd3497", "#f768a1", "#fa9fb5\
", "#fcc5c0", "#fde0dd", "#fff7f3"])
bar.setBarPadding(2)
bar.setBarSolidPercent(0.9)
bar.setBackgroundColor('gray')
```

With these settings you get the following result.



power-attributes

Adding the QAbstractSlider Interface

We've added methods to configure the behaviour of the power bar. But we currently provide no way to configure the standard `QDial` methods — for example, setting the min, max or step size — from our widget. We could work through and add wrapper methods for all of these, but it would get very tedious very quickly.

```
# Example of a single wrapper, we'd need 30+ of these.
def setNotchesVisible(self, b):
    return self._dial.setNotchesVisible(b)
```

Instead we can add a little handler onto our outer widget to automatically look for methods (or attributes) on the `QDial` instance, if they don't exist on our class directly. This way we can implement our own methods, yet still get all the `QAbstractSlider` goodness for free.

The wrapper is shown below, implemented as a custom `__getattr__` method.

```
def __getattr__(self, name):
    if name in self.__dict__:
        return self[name]

      try:
        return getattr(self._dial, name)
      except AttributeError:
        raise AttributeError(
          "'{}' object has no attribute '{}'".format(self.__class__.__name__, n\
ame)
        )
```

When accessing a property (or method) — e.g. when when call `PowerBar.setNotchesVisible(tr` Python internally uses `__getattr__` to get the property from the current object. This handler does this through the object dictionary `self.__dict__`. We've overridden this method to provide our custom handling logic.

Now, when we call `PowerBar.setNotchesVisible(true)`, this handler first looks on our current object (a `PowerBar` instance) to see if `.setNotchesVisible` exists and if it does uses it. If *not* it then calls `getattr()` on `self._dial` instead returning what it finds there. This gives us access to all the methods of `QDial` from our custom `PowerBar`widget.

If `QDial` doesn't have the attribute either, and raises an `AttributeError` we catch it and raise it again from our custom widget, where it belongs.

This works for any properties or methods, including signals. So the standard `QDial` signals such as `.valueChanged` are available too.

Updating from the Meter display

Currently you can update the current value of the PowerBar meter by twiddling with the dial. But it would be nice if you could also update the value by clicking a position on the power bar, or by dragging you mouse up and down. To do this we can update our `_Bar` widget to handle mouse events.

```
class _Bar(QtWidgets.QWidget):

    clickedValue = QtCore.pyqtSignal(int)

    # ... existing code ...

    def _calculate_clicked_value(self, e):
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        d_height = self.size().height() + (self._padding * 2)
        step_size = d_height / self.n_steps
        click_y = e.y() - self._padding - step_size / 2

        pc = (d_height - click_y) / d_height
        value = vmin + pc * (vmax - vmin)
        self.clickedValue.emit(value)

    def mouseMoveEvent(self, e):
        self._calculate_clicked_value(e)

    def mousePressEvent(self, e):
        self._calculate_clicked_value(e)
```

In the `__init__` block for the `PowerBar` widget we can connect to the `_Bar.clickedValue` signal and send the values to `self._dial.setValue` to set the current value on the dial.

```
# Take feedback from click events on the meter.
self._bar.clickedValue.connect(self._dial.setValue)
```

If you run the widget now, you'll be able to click around in the bar area and the value will update, and the dial rotate in sync.

The final code

Below is the complete final code for our PowerBar meter widget, called `PowerBar`. You can save this over the previous file (e.g. named `power_bar.py`) and then use it

in any of your own projects, or customise it further to your own requirements.

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import Qt


class _Bar(QtWidgets.QWidget):

    clickedValue = QtCore.pyqtSignal(int)

    def __init__(self, steps, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.setSizePolicy(
            QtWidgets.QSizePolicy.MinimumExpanding,
            QtWidgets.QSizePolicy.MinimumExpanding
        )

        if isinstance(steps, list):
            # list of colours.
            self.n_steps = len(steps)
            self.steps = steps

        elif isinstance(steps, int):
            # int number of bars, defaults to red.
            self.n_steps = steps
            self.steps = ['red'] * steps

        else:
            raise TypeError('steps must be a list or int')

        self._bar_solid_percent = 0.8
        self._background_color = QtGui.QColor('black')
        self._padding = 4.0  # n-pixel gap around edge.

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

```

```
        brush = QtGui.QBrush()
        brush.setColor(self._background_color)
        brush.setStyle(Qt.SolidPattern)
        rect = QtCore.QRect(0, 0, painter.device().width(), painter.device().he\
ight())
        painter.fillRect(rect, brush)

        # Get current state.
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        value = parent.value()

        # Define our canvas.
        d_height = painter.device().height() - (self._padding * 2)
        d_width = painter.device().width() - (self._padding * 2)

        # Draw the bars.
        step_size = d_height / self.n_steps
        bar_height = step_size * self._bar_solid_percent
        bar_spacer = step_size * (1 - self._bar_solid_percent) / 2

        # Calculate the y-stop position, from the value in range.
        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * self.n_steps)

        for n in range(n_steps_to_draw):
            brush.setColor(QtGui.QColor(self.steps[n]))
            rect = QtCore.QRect(
                self._padding,
                self._padding + d_height - ((1 + n) * step_size) + bar_spacer,
                d_width,
                bar_height
            )
            painter.fillRect(rect, brush)

        painter.end()

    def sizeHint(self):
        return QtCore.QSize(40, 120)
```

```
76
77      def _trigger_refresh(self):
78          self.update()
79
80      def _calculate_clicked_value(self, e):
81          parent = self.parent()
82          vmin, vmax = parent.minimum(), parent.maximum()
83          d_height = self.size().height() + (self._padding * 2)
84          step_size = d_height / self.n_steps
85          click_y = e.y() - self._padding - step_size / 2
86
87          pc = (d_height - click_y) / d_height
88          value = vmin + pc * (vmax - vmin)
89          self.clickedValue.emit(value)
90
91      def mouseMoveEvent(self, e):
92          self._calculate_clicked_value(e)
93
94      def mousePressEvent(self, e):
95          self._calculate_clicked_value(e)
96
97
98  class PowerBar(QtWidgets.QWidget):
99      """
100     Custom Qt Widget to show a power bar and dial.
101     Demonstrating compound and custom-drawn widget.
102
103     Left-clicking the button shows the color-chooser, while
104     right-clicking resets the color to None (no-color).
105     """
106
107     colorChanged = QtCore.pyqtSignal()
108
109     def __init__(self, steps=5, *args, **kwargs):
110         super().__init__(*args, **kwargs)
111
112         layout = QtWidgets.QVBoxLayout()
113         self._bar = _Bar(steps)
114         layout.addWidget(self._bar)
```

```
115
116         # Create the QDial widget and set up defaults.
117         # - we provide accessors on this class to override.
118         self._dial = QtWidgets.QDial()
119         self._dial.setNotchesVisible(True)
120         self._dial.setWrapping(False)
121         self._dial.valueChanged.connect(self._bar._trigger_refresh)
122
123         # Take feedback from click events on the meter.
124         self._bar.clickedValue.connect(self._dial.setValue)
125
126         layout.addWidget(self._dial)
127         self.setLayout(layout)
128
129     def __getattr__(self, name):
130         if name in self.__dict__:
131             return self[name]
132
133         return getattr(self._dial, name)
134
135     def setColor(self, color):
136         self._bar.steps = [color] * self._bar.n_steps
137         self._bar.update()
138
139     def setColors(self, colors):
140         self._bar.n_steps = len(colors)
141         self._bar.steps = colors
142         self._bar.update()
143
144     def setBarPadding(self, i):
145         self._bar._padding = int(i)
146         self._bar.update()
147
148     def setBarSolidPercent(self, f):
149         self._bar._bar_solid_percent = float(f)
150         self._bar.update()
151
152     def setBackgroundColor(self, color):
153         self._bar._background_color = QtGui.QColor(color)
```

```
154         self._bar.update()
```

You should be able to use many of these ideas in creating your own custom widgets. For more examples, take a look at the Learn PyQt widget library — these widgets are all open source and freely available to use in your own projects.

The Model View Architecture

As you start to build more complex applications with PyQt5 you'll likely come across issues keeping widgets in sync with your data.

Data stored in widgets (e.g. a simple `QListWidget`) is not readily available to manipulate from Python — changes require you to get an item, get the data, and then set it back. The default solution to this is to keep an external data representation in Python, and then either duplicate updates to the both the data and the widget, or simple rewrite the whole widget from the data. This can get ugly quickly, and results in a lot of boilerplate just for fiddling the data.

Thankfully Qt has a solution for this — ModelViews. ModelViews are a powerful alternative to the standard display widgets, which use a regular model interface to interact with data sources — from simple data structures to external databases. This isolates your data, allowing it to be kept in any structure you like, while the view takes care of presentation and updates.

This chapter introduces the key aspects of Qt's ModelView architecture and uses it to build simple desktop Todo application in PyQt5.

Model View Controller

Model–View–Controller (MVC) is an architectural pattern used for developing user interfaces which divides an application into three interconnected parts. This separates the internal representation of data from how information is presented to and accepted from the user.

The MVC design pattern decouples three major components —

- **Model** holds the data structure which the app is working with.
- **View** is any representation of information as shown to the user, whether graphical or tables. Multiple views of the same data model are allowed.
- **Controller** accepts input from the user, transforming it into commands to for the model or view.

It Qt land the distinction between the View & Controller gets a little murky. Qt accepts input events from the user (via the OS) and delegates these to the widgets (Controller) to handle. However, widgets also handle presentation of the current state to the user, putting them squarely in the View. Rather than agonize over where to draw the line, in Qt-speak the View and Controller are instead merged together creating a Model/ViewController architecture — called "Model View" for simplicity sake.

Importantly, the distinction between the *data* and *how it is presented* is preserved.

The Model View

The Model acts as the interface between the data store and the ModelView. The Model holds the data (or a reference to it) and presents this data through a standardised API which Views then consume and present to the user. Multiple Views can share the same data, presenting it in completely different ways.

You can use any "data store" for your model, including for example a standard Python list or dictionary, or a database (via e.g. SQLAlchemy) — it's entirely up to you.

The two parts are essentially responsible for —

1. The **model** stores the data, or a reference to it and returns individual or ranges of records, and associated metadata or *display* instructions.
2. The **view** requests data from the model and displays what is returned on the widget.

There is an in-depth view of the Qt architecture here: http://doc.qt.io/qt-5/model-view-programming.html

A simple Model View — a Todo List

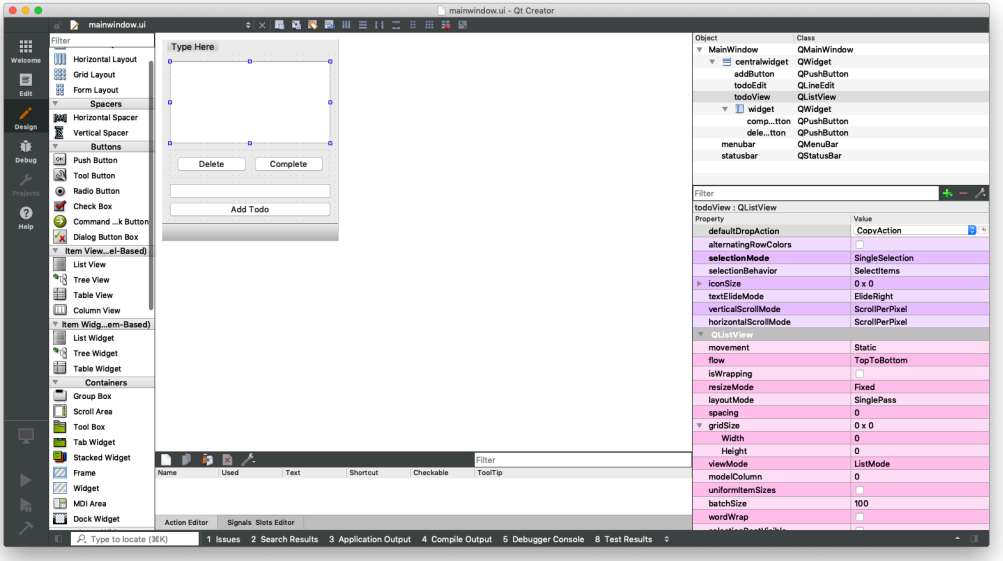
To demonstrate how to use the ModelViews in practise, we'll put together a very simple implementation of a desktop Todo List. This will consist of a `QListView` for

the list of items, a `QLineEdit` to enter new items, and a set of buttons to add, delete, or mark items as done.

The code and associated files for this example are in the downloadable source file.

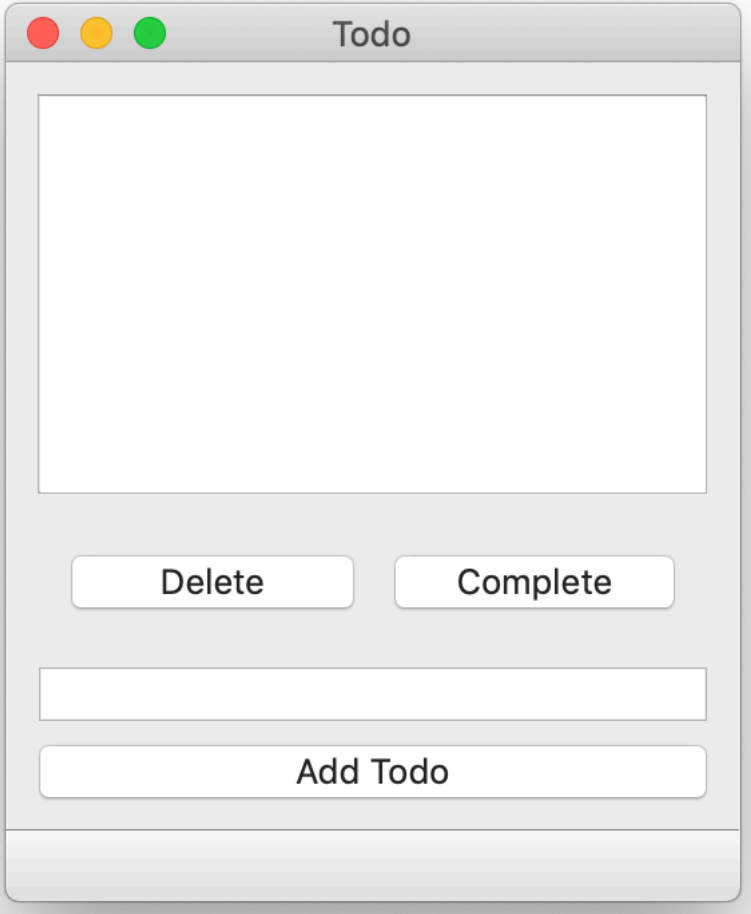
The UI

The simple UI was laid out using Qt Creator and saved as `mainwindow.ui`. The `.ui` file is included in the downloads for this book.



Designing the UI in Qt Creator

The running app is shown below.



The MainWindow

The widgets available in the interface were given the IDs shown in the table below.

objectName	Type	Description
`todoView`	`QListView`	The list of current todos
`todoEdit`	`QLineEdit`	The text input for creating a new todo item
`addButton`	`QPushButton`	Create the new todo, adding it to the todos list
`deleteButton`	`QPushButton`	Delete the current selected todo, removing it from the todos list
`completeButton`	`QPushButton`	Mark the current selected todo as done

We'll use these identifiers to hook up the application logic later.

The Model

We define our custom model by subclassing from a base implementation, allowing us to focus on the parts unique to our model. Qt provides a number of different model bases, including those with support for multidimensional data (think spreadsheet).

But for this example we only need a simple list for our data and are displaying the result to a `QListView`. The matching base model for this is `QAbstractListModel`. The outline definition for our model is shown below.

```
class TodoModel(QtCore.QAbstractListModel):
    def __init__(self, *args, todos=None, **kwargs):
        super(TodoModel, self).__init__(*args, **kwargs)
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.DisplayRole:
            # See below for the data structure.
            status, text = self.todos[index.row()]
            # Return the todo text only.
            return text

    def rowCount(self, index):
        return len(self.todos)
```

The `.todos` variable is our data store and the two methods `rowcount()` and `data()` are standard Model methods we must implement for a list model. We'll go through these in turn below.

.todos list

The data store for our model is `.todos`, a simple Python list in which we'll store a `tuple` of values in the format `[(bool, str), (bool, str), (bool, str)]` where `bool` is the *done* state of a given entry, and `str` is the text of the todo.

We initialise `self.todo` to an empty list on startup, unless a list is passed in view the `todos` keyword argument.

`self.todos = todos or []` will set `self.todos` to the value of the provided `todos` variable if it is *truthy* (i.e. anything other than an empty list, the bool 'False `or` None `the default value), otherwise it will be set to the empty list` []'.

To create an instance of this model we can simply do —

```
model = TodoModel()   # create an empty todo list
```

Or to pass in an existing list —

```
todos = [(False, 'an item'), (False, 'another item')]
model = TodoModel(todos)
````

.rowcount()

The `.rowcount()` method is called by the view to get the number of rows in the\
 current data. This is required for the view to know what the maximum index it \
can request from the data store is (`row count-1`). Since we're using a Python \
list as our data store, the return value for this is simply the `len()` of the \
list.

```
#### .data()

This is the core of your model, which handles requests for data from the view a\
nd returns the appropriate result. It receives two parameters `index` and `role\
.`

`index` is the position/coordinates of the data which the view is requesting sp\
ecified by two methods `.row()` and `.column()` which give the position in a pa\
rticular dimension.

T> For our `QListView` the column is always 0 and can be ignored, but you would\
 need to use this for 2D data in a spreadsheet view.

 `role` is a flag indicating the *type* of data the view is requesting. This is\
 because the `.data()` method actually has more responsibility than just the co\
re data. It also handles requests for style information, tooltips, status bars,\
 etc. — basically anything that could be informed by the data itself.

The naming of `Qt.DisplayRole` is a bit weird, but this indicates that the *vie\
w* is asking us "please give me data for display". There are other *roles* whic\
h the `data` can receive for styling requests or requesting data in "edit-ready\
" format.

| Role                | Value | Description                                                \
              |
| ------------------- | ----- | ----------------------------------------------------\
------------- |
| `Qt.DisplayRole`    | `0`   | The key data to be rendered in the form of text\
. ([QString](https://doc.qt.io/qt-5/qstring.html)) |
| `Qt.DecorationRole` | `1`   | The data to be rendered as a decoration in the \
form of an icon. ([QColor](https://doc.qt.io/qt-5/qcolor.html), [QIcon](https:/\
/doc.qt.io/qt-5/qicon.html) or [QPixmap](https://doc.qt.io/qt-5/qpixmap.html)) \
|
| `Qt.EditRole`       | `2`   | The data in a form suitable for editing in an e\
ditor. ([QString](https://doc.qt.io/qt-5/qstring.html)) |
| `Qt.ToolTipRole`    | `3`   | The data displayed in the item's tooltip. ([QSt\
ring](https://doc.qt.io/qt-5/qstring.html)) |
| `Qt.StatusTipRole`  | `4`   | The data displayed in the status bar. ([QString\
](https://doc.qt.io/qt-5/qstring.html)) |
```

```
| `Qt.WhatsThisRole`   | `5`    | The data displayed for the item in "What's This\
?" mode. ([QString](https://doc.qt.io/qt-5/qstring.html)) |
| `Qt.SizeHintRole`    | `13`   | The size hint for the item that will be supplie\
d to views. ([QSize](https://doc.qt.io/qt-5/qsize.html)) |

For a full list of available *roles* that you can receive see [the Qt ItemDataR\
ole documentation](https://doc.qt.io/qt-5/qt.html#ItemDataRole-enum). Our todo \
list will only be using `Qt.DisplayRole` and `Qt.DecorationRole`.

### Basic implementation

Below is the basic stub application needed to load the UI and display it. We'll\
 add our model code and application logic to this base.

```python
import sys
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt


qt_creator_file = "mainwindow.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qt_creator_file)


class TodoModel(QtCore.QAbstractListModel):
    def __init__(self, *args, todos=None, **kwargs):
        super(TodoModel, self).__init__(*args, **kwargs)
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.DisplayRole:
            status, text = self.todos[index.row()]
            return text

    def rowCount(self, index):
        return len(self.todos)


class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):
```
```

```
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.model = TodoModel()
        self.todoView.setModel(self.model)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
```

We define our `TodoModel` as before, and initialise the `MainWindow` object. In the `__init__` for the MainWindow we create an instance of our todo model and set this model on the `todo_view`. Save this file as `todo.py` and run it with —

```
python3 todo.py
```

While there isn't much to see yet, the `QListView` and our model are actually working — if you add some default data you'll see it appear in the list.

```
self.model = TodoModel(todos=[(False, 'my first todo')])
```

QListView showing hard-coded todo item

You can keep adding items manually like this and they will show up in order in the `QListView`. Next we'll make it possible to add items from within the application.

First create a new method on the `MainWindow` named `add_todo`. This is our callback

which will take care of adding the current text from the input as a new todo. Connect this method to the `addButton.pressed` signal at the end of the `__init__` block.

```
class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.model = TodoModel()
        self.todoView.setModel(self.model)
        # Connect the button.
                            self.addButton.pressed.connect(self.add)

    def add(self):
        """
        Add an item to our todo list, getting the text from the QLineEdit .todo\
Edit
        and then clearing it.
        """
        text = self.todoEdit.text()
        if text: # Don't add empty strings.
            # Access the list via the model.
            self.model.todos.append((False, text))
            # Trigger refresh.
            self.model.layoutChanged.emit()
            # Empty the input
            self.todoEdit.setText("")
```

In the `add_todo` block notice the line `self.model.layoutChanged.emit()`. Here we're emitting a model signal `.layoutChanged` to let the view know that the *shape* of the data has been altered. This triggers a refresh of the entirety of the view. If you omit this line, the todo will still be added but the `QListView` won't update.

If the just data is altered, but the number of rows/columns are unaffected you can use the `.dataChanged()` signal to let Qt know about this. This also a top-left and bottom-right location in the data, to avoid redrawing the entire view.

Hooking up the other actions

We can now connect the rest of the button's signals and add helper functions for performing the *delete* and *complete* operations. We add the button signals to the `__init__` block as before.

```
        self.addButton.pressed.connect(self.add)
        self.deleteButton.pressed.connect(self.delete)
        self.completeButton.pressed.connect(self.complete)
```

Then define a new `delete` method as follows —

```
    def delete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            # Indexes is a list of a single item in single-select mode.
            index = indexes[0]
            # Remove the item and refresh.
            del self.model.todos[index.row()]
            self.model.layoutChanged.emit()
            # Clear the selection (as it is no longer valid).
            self.todoView.clearSelection()
```

We use `self.todoView.selectedIndexes` to get the indexes (actually a list of a single item, as we're in single-selection mode) and then use the `.row()` as an index into our list of todos on our model. We delete the indexed item using Python's `del` operator, and then trigger a `layoutChanged` signal because the shape of the data has been modified.

Finally, we clear the active selection since the item it relates to may now out of bounds (if you had selected the last item).

You could try make this smarter, and select the last item in the list instead.

The `complete` method looks like this —

```
    def complete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            index = indexes[0]
            row = index.row()
            status, text = self.model.todos[row]
            self.model.todos[row] = (True, text)
            # .dataChanged takes top-left and bottom right, which are equal
            # for a single selection.
            self.model.dataChanged.emit(index, index)
            # Clear the selection (as it is no longer valid).
            self.todoView.clearSelection()
```

This uses the same indexing as for delete, but this time we fetch the item from the model `.todos` list and then replace the status with `True`.

We have to do this fetch-and-replace, as our data is stored as Python tuples which cannot be modified.

The key difference here vs. standard Qt widgets is that we make changes directly to our data, and simply need to notify Qt that some change has occurred — updating the widget state is handled automatically.

Using Qt.DecorationRole

If you run the application now you should find that adding and deleting both work, but while completing items is working, there is no indication of it in the view. We need to update our model to provide the view with an indicator to display when an item is complete. The updated model is shown below.

```
tick = QtGui.QImage('tick.png')


class TodoModel(QtCore.QAbstractListModel):
    def __init__(self, *args, todos=None, **kwargs):
        super(TodoModel, self).__init__(*args, **kwargs)
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.DisplayRole:
            _, text = self.todos[index.row()]
            return text

        if role == Qt.DecorationRole:
            status, _ = self.todos[index.row()]
            if status:
                return tick

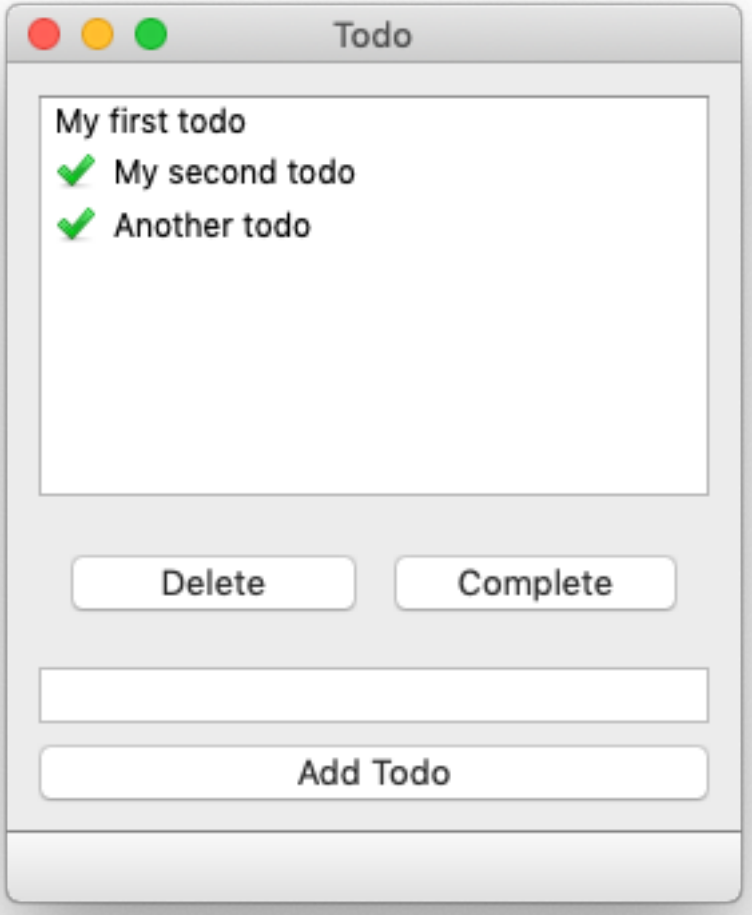
    def rowCount(self, index):
        return len(self.todos)
```

We're using a tick icon `tick.png` to indicate completed items, which we load into a `QImage` object named `tick`. In the model we've implemented a handler for the `Qt.DecorationRole` which returns the tick icon for rows who's `status` is `True` (for complete).

The icon I'm using is taken from the Fugue set by p.yusukekamiyamane

Instead of an icon you can also return a color, e.g. `QtGui.QColor('green')` which will be drawn as solid square.

Running the app you should now be able to mark items as complete.



Todos complete

A persistent data store

Our todo app works nicely, but it has one fatal flaw — it forgets your todos as soon as you close the application While thinking you have nothing to do when you do may help to contribute to short-term feelings of Zen, long term it's probably a bad idea.

The solution is to implement some sort of persistent data store. The simplest approach is a simple file store, where we load items from a JSON or Pickle file at startup, and write back on changes.

To do this we define two new methods on our `MainWindow` class — `load` and `save`. These load data from a JSON file name `data.json` (if it exists, ignoring the error if it doesn't) to `self.model.todos` and write the current `self.model.todos` out to the same file, respectively.

```
    def load(self):
        try:
            with open('data.json', 'r') as f:
                self.model.todos = json.load(f)
        except Exception:
            pass

    def save(self):
        with open('data.json', 'w') as f:
            data = json.dump(self.model.todos, f)
```

To persist the changes to the data we need to add the `.save()` handler to the end of any method that modifies the data, and the `.load()` handler to the `__init__` block after the model has been created.

The final code looks like this —

```
import sys
import json
from PyQt5 import QtCore, QtGui, QtWidgets, uic
from PyQt5.QtCore import Qt


qt_creator_file = "mainwindow.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qt_creator_file)
tick = QtGui.QImage('tick.png')


class TodoModel(QtCore.QAbstractListModel):
    def __init__(self, *args, todos=None, **kwargs):
        super(TodoModel, self).__init__(*args, **kwargs)
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.DisplayRole:
            _, text = self.todos[index.row()]
            return text

        if role == Qt.DecorationRole:
            status, _ = self.todos[index.row()]
            if status:
                return tick

    def rowCount(self, index):
        return len(self.todos)


class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.model = TodoModel()
        self.load()
        self.todoView.setModel(self.model)
        self.addButton.pressed.connect(self.add)
```

```
40          self.deleteButton.pressed.connect(self.delete)
41          self.completeButton.pressed.connect(self.complete)
42
43      def add(self):
44          """
45          Add an item to our todo list, getting the text from the QLineEdit .todo\
46  Edit
47          and then clearing it.
48          """
49          text = self.todoEdit.text()
50          if text: # Don't add empty strings.
51              # Access the list via the model.
52              self.model.todos.append((False, text))
53              # Trigger refresh.
54              self.model.layoutChanged.emit()
55              # Empty the input
56              self.todoEdit.setText("")
57              self.save()
58
59      def delete(self):
60          indexes = self.todoView.selectedIndexes()
61          if indexes:
62              # Indexes is a list of a single item in single-select mode.
63              index = indexes[0]
64              # Remove the item and refresh.
65              del self.model.todos[index.row()]
66              self.model.layoutChanged.emit()
67              # Clear the selection (as it is no longer valid).
68              self.todoView.clearSelection()
69              self.save()
70
71      def complete(self):
72          indexes = self.todoView.selectedIndexes()
73          if indexes:
74              index = indexes[0]
75              row = index.row()
76              status, text = self.model.todos[row]
77              self.model.todos[row] = (True, text)
78              # .dataChanged takes top-left and bottom right, which are equal
```

```
            # for a single selection.
            self.model.dataChanged.emit(index, index)
            # Clear the selection (as it is no longer valid).
            self.todoView.clearSelection()
            self.save()

    def load(self):
        try:
            with open('data.db', 'r') as f:
                self.model.todos = json.load(f)
        except Exception:
            pass

    def save(self):
        with open('data.db', 'w') as f:
            data = json.dump(self.model.todos, f)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec_()
```

If the data in your application has the potential to get large or more complex, you may prefer to use an actual database to store it. In this case the model will wrap the interface to the database and query it directly for data to display.

For another interesting example of a QListView see this example media player application: https://www.learnpyqt.com/apps/failamp-multimedia-player/ This uses the Qt built-in QMediaPlaylist as the datastore, with the contents displayed to a QListView.

Multithreading

As you start to build more complex applications, you may come across problems where long-running tasks "lock up" your interface.

The event loop started by calling `.exec_()` on your `QApplication` object and runs within the same thread as your Python code. The thread which runs this event loop — commonly referred to as the *GUI thread* — also handles all window communication with the host operating system.

By default, any execution triggered by the event loop will also run synchronously within this thread. In practise this means that any time your PyQt application spends *doing something* in your code, window communication and GUI interaction are frozen.

If what you're doing is simple, and returns control to the GUI loop quickly, this freeze will be imperceptible to the user. However, if you need to perform longer-running tasks, for example opening/writing a large file, downloading some data, or rendering some complex image, there are going to be problems. To your user the application will appear to be unresponsive (because it is). Because your app is no longer communicating with the OS, on MacOS X if you click on your app you will see the spinning wheel of death. And, *nobody* wants that.

The solution is simple: get your work out of the *GUI* thread and into another . PyQt (via Qt) provides a straightforward interface to do exactly that.

Preparation

To demonstrate multi-threaded execution we need an application to work with. Below is a minimal stub application for PyQt which will allow us to demonstrate multithreading, and see the outcome in action. Simply copy and paste this into a new file, and save it with an appropriate filename like `multithread.py`. The remainder of the code will be added to this file (there is also a complete working example at the bottom if you're impatient).

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

import time


class MainWindow(QMainWindow):
    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def oh_no(self):
        time.sleep(5)

    def recurring_timer(self):
        self.counter += 1
```

```
40          self.l.setText("Counter: %d" % self.counter)
41
42
43  app = QApplication([])
44  window = MainWindow()
45  app.exec_()
```

Run the file as normal:

```
1  python3 multithread.py
```

You should see a demonstration window with a number counting upwards. This a generated by a simple recurring time, firing once per second. Think of this as our *event loop indicator*, a simple way to let us known that out application is ticking over normally. There is also a button with the word *“DANGER!”*.

Push it.

Push the button

You'll notice that each time you push the button the counter stops ticking and your application freezes entirely. On Windows you may see the window turn pale, indicating it is not responding, while on a Mac you may see the spinning wheel of death.

The dumb approach

What appears as a *frozen interface* is in fact caused by the Qt event loop being blocked from processing (and responding to) window events. Your clicks on the window as still registered by the host OS and sent to your application, but because it's sat in your big ol' lump of code (`time.sleep`), it can't accept or react to them. Your application does not respond to the OS and it interprets this as a frozen application.

The simplest way get around this is to accept events from within your code. This

allows Qt to continue to respond to the host OS and your application will stay responsive. You can do this easily by using the static `.processEvents()` function on the `QApplication` class. Simply add a line like the following, somewhere in your long-running code block:

```
QApplication.processEvents()
```

If we can take our long-running `time.sleep` code and break it down into multiple steps, we can insert `.processEvents` in between. The code for this would be:

```
def oh_no(self):
    for n in range(5):
        QApplication.processEvents()
        time.sleep(1)
```

Now when you push the button your code is entered as before. However, now `QApplication.processEvents()` intermittently passes control back to Qt, and allows it to respond to OS events as normal. Qt will now accept events *and handle them* before returning to run the remainder of your code.

This works, but it's horrible for a few reasons —

1. When you pass control back to Qt, your code is no longer running. This means that whatever long-running thing you're trying to do will take *longer*. That is definitely not what you want.
2. You can only do this if your long-running task is broken into multiple short-running steps. If you increase the the duration to `time.sleep(30)` and re-run it you'll see the UI freeze again. The individual step is now so long that control is not passed back frequently enough.
3. Processing events outside the main event loop (`app.exec_()`) causes your application to branch off into handling code (e.g. for triggered slots, or events) while within your loop. If your code depends on/responds to external state this can cause undefined behaviour. The code below demonstrates this in action:

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

import time


class MainWindow(QMainWindow):
    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        c = QPushButton("?")
        c.pressed.connect(self.change_message)

        layout.addWidget(self.l)
        layout.addWidget(b)

        layout.addWidget(c)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

    def change_message(self):
        self.message = "OH NO"

    def oh_no(self):
        self.message = "Pressed"
```

```
40
41          for n in range(100):
42              time.sleep(0.1)
43              self.l.setText(self.message)
44              QApplication.processEvents()
45
46
47  app = QApplication([])
48  window = MainWindow()
49  app.exec_()
```

If you run this code you'll see the counter as before. Pressing "DANGER!" will change the displayed text to "Pressed", as defined at the entry point to the `oh_no` function. However, if you press the "?" button while `oh_no` is still running you'll see that the message changes. State is being changed from outside your loop.

This is a toy example. However, if you have multiple long-running processes within your application, with each calling `QApplication.processEvents()` to keep things ticking, your application behaviour can be unpredictable.

Threads and Processes

If you take a step back and think about what you want to happen in your application, it can probably be summed up with "stuff to happen at the same time as other stuff happens".

There are two main approaches to running independent tasks within a PyQt application: *threads* and *processes*.

Threads share the same memory space, so are quick to start up and consume minimal resources. The shared memory makes it trivial to pass data between threads, however reading/writing memory from different threads can lead to race conditions or segfaults. In Python there is the added issue that multiple threads are bound by the same Global Interpreter Lock (GIL) — meaning non-GIL-releasing Python code can only execute in one thread at a time. However, this is not a major issue with PyQt where most of the time is spent outside of Python.

Processes use separate memory space (and an entirely separate Python interpreter). This side-steps any potential problems with the GIL, but at the cost of

slower start-up times, larger memory overhead and complexity in sending/receiving data.

For simplicity's sake it usually makes sense to use threads, unless you have a good reason to use processes (see caveats later). Subprocesses in Qt are better suited to running and communicating with external programs.

`QRunnable` and `QThreadPool`

Qt provides a very simple interface for running jobs in other threads, which is exposed nicely in PyQt. This is built around two classes: `QRunnable` and `QThreadPool`. The former is the container for the work you want to perform, while the latter is the method by which you pass that work to alternate threads.

The neat thing about using `QThreadPool` is that it handles queuing and execution of workers for you. Aside from queuing up jobs and retrieving the results, there is not very much to do at all.

To define a custom `QRunnable` you can subclass the base `QRunnable` class, then place the code you wish you execute within the `run()` method. The following is an implementation of our long running `time.sleep` job as a `QRunnable`. Add the following code to `multithread.py`, above the `MainWindow` class definition.

```
class Worker(QRunnable):
    '''
    Worker thread
    '''

    @pyqtSlot()
    def run(self):
        '''
        Your code goes in this function
        '''
        print("Thread start")
        time.sleep(5)
        print("Thread complete")
```

Executing our function in another thread is simply a matter of creating an instance of the `Worker` and then pass it to our `QThreadPool` instance and it will be executed automatically.

Next add the following within the `__init__` block, to set up our thread pool.

```
self.threadpool = QThreadPool()
print("Multithreading with maximum %d threads" % self.threadpool.maxThreadCount\
())
```

Finally, add the following lines to our `oh_no` function.

```
def oh_no(self):
    worker = Worker()
    self.threadpool.start(worker)
```

Now, clicking on the button will create a worker to handle the (long-running) process and spin that off into another thread via thread pool. If there are not enough threads available to process incoming workers, they'll be queued and executed in order at a later time.

Try it out and you'll see that your application now handles you bashing the button with no problems.

Check what happens if you hit the button multiple times. You should see your threads executed immediately *up to* the number reported by `.maxThreadCount`. If you hit the button again after there are already this number of active workers, the subsequent workers will be queued until a thread becomes available.

Extended Runners

If you want to pass custom data into the execution function you can set up your runner to take *arguments* or *keywords* and then store the data on the runner itself. The data will be accessible while running via `self` of your `QRunnable` object.

In fact, you can even take advantage of the fact that in Python functions are objects and pass in the function to execute rather than subclassing each time. In the

following construction we only require a single `Worker` class to handle all of our execution jobs.

```
class Worker(QRunnable):
    '''
    Worker thread

    :param fn: The function to be executed
    :param args: Arguments to make available to the run code
    :param kwargs: Keywords arguments to make available to the run
    :code
    :
    '''

    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()
        self.fn = fn
        self.args = args
        self.kwargs = kwargs

    @pyqtSlot()
    def run(self):
        '''
        Execute the runner function with passed self.args,
        self.kwargs.
        '''
        self.fn(args, kwargs)
```

You can now pass in any Python function and have it executed in a separate thread.

```
def execute_this_fn(self):
    print("Hello!")

def oh_no(self):
    # Pass the function to execute
    worker = Worker(self.execute_this_fn) # Any other args, kwargs are passed t\
o the run function

    # Execute
    threadpool.start(worker)
```

Thread IO

Sometimes it's helpful to be able to pass back *state* and *data* from running workers. This could include the outcome of calculations, raised exceptions or ongoing progress (think progress bars). Qt provides the *signals and slots* framework which allows you to do just that and is thread-safe, allowing safe communication directly from running threads to your GUI frontend. *Signals* allow you to `.emit` values, which are then picked up elsewhere in your code by *slot* functions which have been linked with `.connect`.

Below is a simple `WorkerSignals` class defined to contain a number of example signals.

Custom signals can only be defined on objects derived from `QObject`. Since `QRunnable` is not derived from `QObject` we can't define the signals on it directly. A custom `QObject` to hold the signals is the simplest solution.

```
import traceback, sys


class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

    Supported signals are:

    finished
        No data

    error
        `tuple` (exctype, value, traceback.format_exc() )

    result
        `object` data returned from processing, anything

    '''
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)
```

In this example we've defined 5 custom signals:

1. *finished* signal, with no data to indicate when the task is complete.
2. *error* signal which receives a `tuple` of `Exception` type, `Exception` value and formatted traceback.
3. *result* signal receiving any `object` type from the executed function.

You may not find a need for all of these signals, but they are included to give an indication of what is possible. In the following code we're going to implement a long-running task that makes use of these signals to provide useful information to the user.

```
class Worker(QRunnable):
    '''
    Worker thread

    Inherits from QRunnable to handler worker thread setup, signals and wrap-up.

    :param callback: The function callback to run on this worker
    :thread. Supplied args and
                    kwargs will be passed through to the runner.
    :type callback: function
    :param args: Arguments to pass to the callback function
    :param kwargs: Keywords to pass to the callback function
    :
    '''

    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()
        # Store constructor arguments (re-used for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        '''
        Initialise the runner function with passed args, kwargs.
        '''

        # Retrieve args/kwargs here; and fire processing using them
        try:
            result = self.fn(
                *self.args, **self.kwargs,
                status=self.signals.status,
                progress=self.signals.progress,
            )
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
```

```
            self.signals.error.emit((exctype, value, traceback.format_exc()))
        else:
            self.signals.result.emit(result)  # Return the result of the proces\
sing
        finally:
            self.signals.finished.emit()  # Done
```

You can connect your own handler functions to these signals to receive notification of completion (or the result) of threads.

```
def execute_this_fn(self):
    for n in range(0, 5):
        time.sleep(1)
    return "Done."

def print_output(self, s):
    print(s)

def thread_complete(self):
    print("THREAD COMPLETE!")

def oh_no(self):
    # Pass the function to execute
    worker = Worker(self.execute_this_fn) # Any other args, kwargs are passed t\
o the run function
    worker.signals.result.connect(self.print_output)
    worker.signals.finished.connect(self.thread_complete)

    # Execute
    self.threadpool.start(worker)
```

`QRunnable` Examples

The features of `QRunnables` described can be used to build runners suited for a variety of tasks. Below are some examples for how to construct runners and applications to use them in a number of different scenarios.

The Updater

You often want to receive progress information information from long-running threads. This can be done easily defining a signal on the `WorkerSignals` object, through which you pass a number indicating % completion. The example below uses this to update a running progress bar.

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

import time


class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

    progress
        int progress complete,from 0-100
    '''
    progress = pyqtSignal(int)


class Worker(QRunnable):
    '''
    Worker thread

    Inherits from QRunnable to handler worker thread setup, signals
    and wrap-up.
    '''

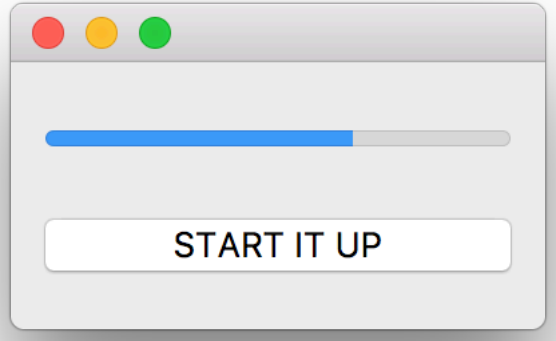
    def __init__(self):
        super(Worker, self).__init__()

        self.signals = WorkerSignals()

    @pyqtSlot()
```

```
32      def run(self):
33          total_n = 1000
34          for n in range(total_n):
35              progress_pc = int(100*float(n)/total_n)  # Progress 0-100% as int
36              self.signals.progress.emit(progress_pc)
37              time.sleep(0.01)
38
39
40  class MainWindow(QMainWindow):
41
42
43      def __init__(self, *args, **kwargs):
44          super(MainWindow, self).__init__(*args, **kwargs)
45
46
47          layout = QVBoxLayout()
48
49          self.bar = QProgressBar()
50
51          button = QPushButton("START IT UP")
52          button.pressed.connect(self.execute)
53
54          layout.addWidget(self.bar)
55          layout.addWidget(button)
56
57          w = QWidget()
58          w.setLayout(layout)
59
60          self.setCentralWidget(w)
61
62          self.show()
63
64          self.threadpool = QThreadPool()
65          print("Multithreading with maximum %d threads" % self.threadpool.maxThr\
66  eadCount())
67
68      def execute(self):
69          worker = Worker()
70          worker.signals.progress.connect(self.update_progress)
```

```
71
72          # Execute
73          self.threadpool.start(worker)
74
75      def update_progress(self, progress):
76          self.bar.setValue(progress)
77
78
79
80  app = QApplication([])
81  window = MainWindow()
82  app.exec_()
```



Progress bar showing current progress for a long-running worker.

If you want to support variable runners by passing in a function it will not have access to the progress callback. You can get around this by passing it into your function directly. See the *Generic Logger* description below.

The Logger

Threading is a good option where you need to run IO operations and receive the data from them. This can mean interacting with APIs or websites, or receiving serial data from hardware.

In this example we create multiple runners, each sending back their data live, tagged with their own identifier. This allows the returning data to be associated with the runner it has come from, and forwarded to the correct output.

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

import requests

class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

    data
        tuple of (identifier, data)
    '''
    data = pyqtSignal(tuple)


class Worker(QRunnable):
    '''
    Worker thread

    Inherits from QRunnable to handler worker thread setup, signals
    and wrap-up.

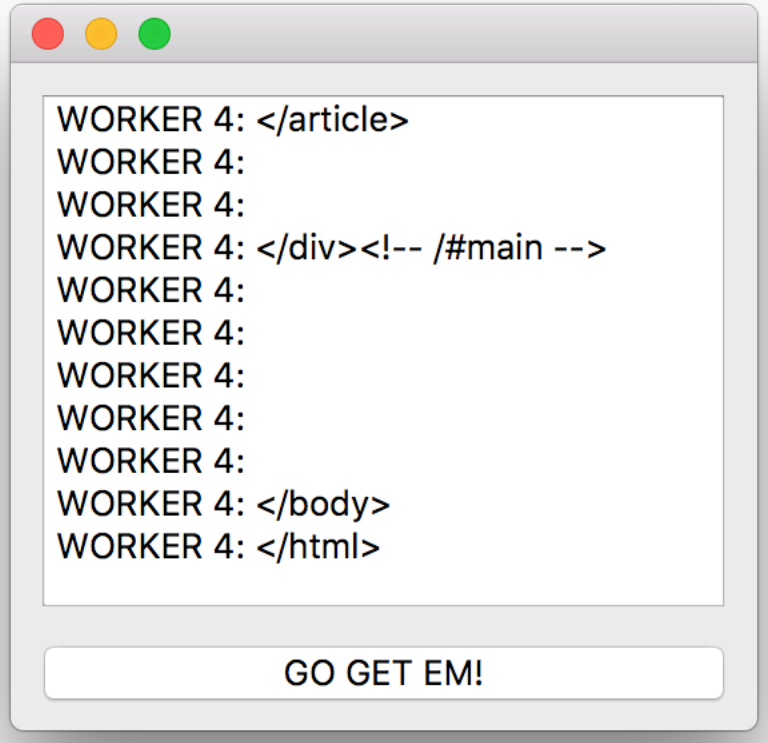
    :param id: The id for this worker
    :param url: The url to retrieve
    '''

    def __init__(self, id, url):
```

```
29          super(Worker, self).__init__()
30          self.id = id
31          self.url = url
32
33          self.signals = WorkerSignals()
34
35      @pyqtSlot()
36      def run(self):
37          r = requests.get(self.url)
38
39          for line in r.text.splitlines():
40              self.signals.data.emit((self.id, line))
41
42
43  class MainWindow(QMainWindow):
44
45
46      def __init__(self, *args, **kwargs):
47          super(MainWindow, self).__init__(*args, **kwargs)
48
49          self.urls = [
50              'http://www.example.com',
51              'https://www.mfitzp.com',
52              'https://www.google.com',
53              'https://www.udemy.com/create-simple-gui-applications-with-python-a\
54  nd-qt/',
55              'https://books.mfitzp.com/create-simple-gui-applications/'
56          ]
57
58          layout = QVBoxLayout()
59
60          self.text = QPlainTextEdit()
61          self.text.setReadOnly(True)
62
63          button = QPushButton("GO GET EM!")
64          button.pressed.connect(self.execute)
65
66          layout.addWidget(self.text)
67          layout.addWidget(button)
```

```
68
69          w = QWidget()
70          w.setLayout(layout)
71
72          self.setCentralWidget(w)
73
74          self.show()
75
76          self.threadpool = QThreadPool()
77          print("Multithreading with maximum %d threads" % self.threadpool.maxThr\
78  eadCount())
79
80
81      def execute(self):
82          for n, url in enumerate(self.urls):
83              worker = Worker(n, url)
84              worker.signals.data.connect(self.display_output)
85
86              # Execute
87              self.threadpool.start(worker)
88
89
90      def display_output(self, data):
91          id, s = data
92          self.text.appendPlainText("WORKER %d: %s" % (id, s))
93
94
95
96  app = QApplication([])
97  window = MainWindow()
98  app.exec_()
```

If you run this example and press the button you'll see the HTML output from a number of websites, prepended by the worker ID that retrieve them. Note that output from different workers is interleaved.



Logging output from multiple workers to the main window.

The `tuple` is of course optional, you could send back bare strings if you have only one runner, or don't need to associated outputs with a source. It is also possible to send a `bytestring`, or any other type of data, by setting up the signals appropriately.

The Generic

If you have multiple similar jobs to run, or the runners have no specific requirements, a generic runner may be all you need. Pass in the function to run and receive output, errors and progress.

A complete working example is given below, showcasing the custom `QRunnable` worker together with the worker & progress signals. You should be able to easily adapt this code to any application you develop.

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

import time
import traceback, sys


class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

    Supported signals are:

    finished
        No data

    error
        `tuple` (exctype, value, traceback.format_exc() )

    result
        `object` data returned from processing, anything

    progress
        `int` indicating % progress

    '''
    finished = pyqtSignal()
```

```
29      error = pyqtSignal(tuple)
30      result = pyqtSignal(object)
31      progress = pyqtSignal(int)
32
33
34  class Worker(QRunnable):
35      '''
36      Worker thread
37
38      Inherits from QRunnable to handler worker thread setup, signals
39  and wrap-up.
40
41      :param callback: The function callback to run on this worker
42      :thread. Supplied args and
43                          kwargs will be passed through to the runner.
44      :type callback: function
45      :param args: Arguments to pass to the callback function
46      :param kwargs: Keywords to pass to the callback function
47      :
48      '''
49
50      def __init__(self, fn, *args, **kwargs):
51          super(Worker, self).__init__()
52          # Store constructor arguments (re-used for processing)
53          self.fn = fn
54          self.args = args
55          self.kwargs = kwargs
56          self.signals = WorkerSignals()
57
58          # Add the callback to our kwargs
59          kwargs['progress_callback'] = self.signals.progress
60
61      @pyqtSlot()
62      def run(self):
63          '''
64          Initialise the runner function with passed args, kwargs.
65          '''
66
67          # Retrieve args/kwargs here; and fire processing using them
```

```
68          try:
69              result = self.fn(*self.args, **self.kwargs)
70          except:
71              traceback.print_exc()
72              exctype, value = sys.exc_info()[:2]
73              self.signals.error.emit((exctype, value, traceback.format_exc()))
74          else:
75              self.signals.result.emit(result)  # Return the result of the proces\
76  sing
77          finally:
78              self.signals.finished.emit()  # Done
79
80
81
82  class MainWindow(QMainWindow):
83
84
85      def __init__(self, *args, **kwargs):
86          super(MainWindow, self).__init__(*args, **kwargs)
87
88          self.counter = 0
89
90          layout = QVBoxLayout()
91
92          self.l = QLabel("Start")
93          b = QPushButton("DANGER!")
94          b.pressed.connect(self.oh_no)
95
96          layout.addWidget(self.l)
97          layout.addWidget(b)
98
99          w = QWidget()
100         w.setLayout(layout)
101
102         self.setCentralWidget(w)
103
104         self.show()
105
106         self.threadpool = QThreadPool()
```

```
        print("Multithreading with maximum %d threads" % self.threadpool.maxThr\
eadCount())

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def progress_fn(self, n):
        print("%d%% done" % n)

    def execute_this_fn(self, progress_callback):
        for n in range(0, 5):
            time.sleep(1)
            progress_callback.emit(n*100/4)

        return "Done."

    def print_output(self, s):
        print(s)

    def thread_complete(self):
        print("THREAD COMPLETE!")

    def oh_no(self):
        # Pass the function to execute
        worker = Worker(self.execute_this_fn) # Any other args, kwargs are pass\
ed to the run function
        worker.signals.result.connect(self.print_output)
        worker.signals.finished.connect(self.thread_complete)
        worker.signals.progress.connect(self.progress_fn)

        # Execute
        self.threadpool.start(worker)


    def recurring_timer(self):
        self.counter +=1
        self.l.setText("Counter: %d" % self.counter)
```

```
146
147
148 app = QApplication([])
149 window = MainWindow()
150 app.exec_()
```

Using Python multithreading in PyQt

You may have spotted the slight flaw in this master plan — we are still making use of the event loop (and the *GUI thread*) to process the output of our workers.

This isn't a problem when we're simply tracking progress, completion or returning metadata. However, if you have workers which return large amounts of data — e.g. loading large files, performing complex analysis and need (large) results, or querying databases — passing this data back through the GUI thread may cause performance problems and is best avoided.

Similarly, if your application makes use of a large number of threads and Python result handlers, you may come up against the limitations of the GIL. As mentioned previously, when using threads execution of Python is limited to a single thread at one time. The Python code that handles signals from your threads can be blocked by your workers and *vice versa*. Since blocking your slot functions blocks the event loop, this can directly impact GUI responsiveness.

In these cases it is often better to use a pure-Python thread pool (e.g. concurrent futures) implementation to keep your *processing* and thread-event handling further isolated from your GUI.

Example PyQt5 Applications

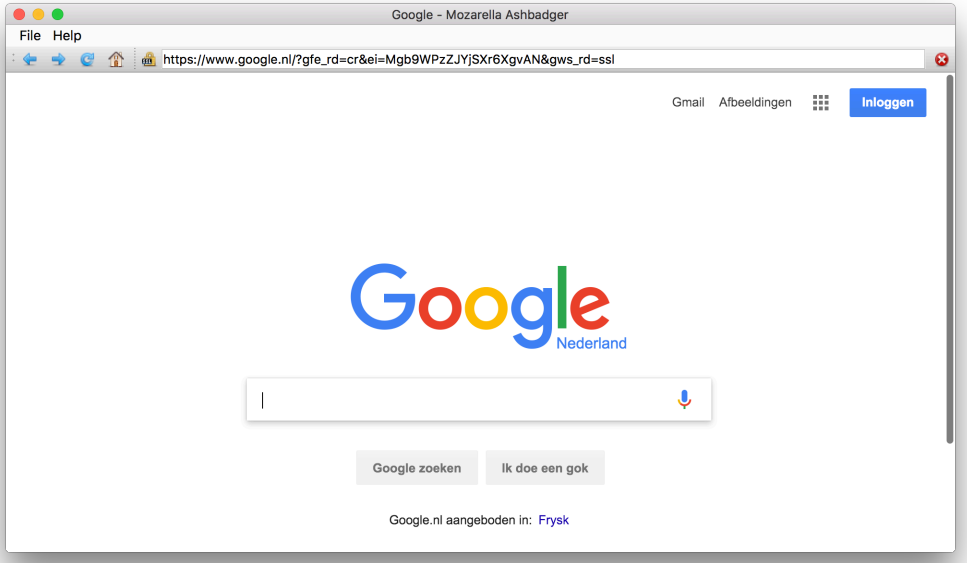
By now you should have a firm grasp of how to go about building simple applications with PyQt. To show how you can put what you've learnt into practise, I've included a few example applications in this chapter.

These applications are functional, simple and in some ways *incomplete*. Use them for inspiration, to pull apart and as an opportunity to improve. Read on for a walkthrough of each app's most interesting parts.

The full source for both apps is available for download, along with 13 other applications in my 15 Minute Apps repository on Github. Have fun!

Mozzarella Ashbadger

Mozzarella Ashbadger is the latest revolution in web browsing! Go back and forward! Print! Save files! Get help! (you'll need it). Any similarity to other browsers is entirely coincidental.



Mozzarella Ashbadger.

This application makes use of features covered in Extended Signals.

The source code for Mozzarella Ashbadger is provided in two forms, on with tabbed browsing and one without. Adding tabs complicates the signal handling a little bit, so the tab-less version is covered first.

Source code

The full source for the tab-less browser is included in the downloads for this book. The browser code has the name `browser.py`.

```
1 python3 browser.py
```

Run it! Explore the *Mozzarella Ashbadger* interface and features before moving onto the code.

It is recommend you download and take a quick look at the source code before continuing. The walkthrough below highlights the key (and interesting) parts of the code, but there is more to see.

The browser widget

The core of our browser is the `QWebEngineView` which we import from `PyQt5.QtWebEngineWidget` This provides a complete browser window, which handles the rendering of the downloaded pages.

Below is the bare-minimum of code required to use web browser widget in PyQt.

```
1  from PyQt5.QtCore import *
2  from PyQt5.QtWidgets import *
3  from PyQt5.QtGui import *
4  from PyQt5.QtWebEngineWidgets import *
5
6  import sys
7
8  class MainWindow(QMainWindow):
9
10     def __init__(self, *args, **kwargs):
11         super(MainWindow,self).__init__(*args, **kwargs)
12
13         self.browser = QWebEngineView()
14         self.browser.setUrl(QUrl("http://google.com"))
15
16         self.setCentralWidget(self.browser)
17
18         self.show()
19
20 app = QApplication(sys.argv)
21 window = MainWindow()
22
23 app.exec_()
```

If you click around a bit you'll discover that the browser behaves as expected — links work correctly, and you can interact with the pages. However, you'll also

notice things you take for granted are missing — like an URL bar, controls or any sort of interface whatsoever. This makes it a little tricky to use.

Navigation

To convert this bare-bones browser into something usable we add some controls, as a series of `QActions` on a `QToolbar`. We add these definitions to the `__init__`-block of the `QMainWindow`.

```
62          navtb = QToolBar("Navigation")
63          navtb.setIconSize( QSize(16,16) )
64          self.addToolBar(navtb)
65
66          back_btn = QAction( QIcon(os.path.join('icons','arrow-180.png')), "Back\
67  ", self)
68          back_btn.setStatusTip("Back to previous page")
69          back_btn.triggered.connect( self.browser.back )
70          navtb.addAction(back_btn)
```

The `QWebEngineView` includes slots for forward, back and reload navigation, which we can connect to directly to our action's `.triggered` signals.

We use the same `QAction` structure for the remaining controls.

```
73          next_btn = QAction( QIcon(os.path.join('icons','arrow-000.png')), "Forw\
74  ard", self)
75          next_btn.setStatusTip("Forward to next page")
76          next_btn.triggered.connect( self.browser.forward )
77          navtb.addAction(next_btn)
78
79          reload_btn = QAction( QIcon(os.path.join('icons','arrow-circle-315.png'\
80  )), "Reload", self)
81          reload_btn.setStatusTip("Reload page")
82          reload_btn.triggered.connect( self.browser.reload )
83          navtb.addAction(reload_btn)
84
85          home_btn = QAction( QIcon(os.path.join('icons','home.png')), "Home", se\
```

```
lf)
        home_btn.setStatusTip("Go home")
        home_btn.triggered.connect( self.navigate_home )
        navtb.addAction(home_btn)
```

Notice that while forward, back and reload can use built-in slots, the navigate home button requires a custom slot function. The slot function is defined on our `QMainWindow` class, and simply sets the URL of the browser to the Google homepage. Note that the URL must be passed as a `QUrl` object.

```
    def navigate_home(self):
        self.browser.setUrl( QUrl("http://www.google.com") )
```

Try making the home navigation location configurable. You could create a Preferences `QDialog` with an input field.

Any decent web browser also needs an URL bar, and some way to stop the navigation — either when it's by mistake, or the page is taking too long.

```
        self.httpsicon = QLabel() # Yes, really!
        self.httpsicon.setPixmap( QPixmap( os.path.join('icons','lock-nossl.png\
') ) )
        navtb.addWidget(self.httpsicon)

        self.urlbar = QLineEdit()
        self.urlbar.returnPressed.connect( self.navigate_to_url )
        navtb.addWidget(self.urlbar)

        stop_btn = QAction( QIcon(os.path.join('icons','cross-circle.png')), "S\
top", self)
        stop_btn.setStatusTip("Stop loading current page")
        stop_btn.triggered.connect( self.browser.stop )
        navtb.addAction(stop_btn)
```

As before the 'stop' functionality is available on the `QWebEngineView`, and we can simply connect the `.triggered` signal from the stop button to the existing slot. However, other features of the URL bar we must handle independently.

First we add a `QLabel` to hold our SSL or non-SSL icon to indicate whether the page is secure. Next, we add the URL bar which is simply a `QLineEdit`. To trigger the loading of the URL in the bar when entered (return key pressed) we connect to the `.returnPressed` signal on the widget to drive a custom slot function to trigger navigation to the specified URL.

```
202     def navigate_to_url(self): # Does not receive the Url
203         q = QUrl( self.urlbar.text() )
204         if q.scheme() == "":
205             q.setScheme("http")
206 
207         self.browser.setUrl(q)
```

We also want the URL bar to update in response to page changes. To do this we can use the `.urlChanged` and `.loadFinished` signals from the `QWebEngineView`. We set up the connections from the signals in the `__init__` block as follows:

```
57         self.browser.urlChanged.connect(self.update_urlbar)
58         self.browser.loadFinished.connect(self.update_title)
```

Then we define the target slot functions which for these signals. The first, to update the URL bar accepts a `QUrl` object and determines whether this is a `http` or `https` URL, using this to set the SSL icon.

WARNING: This is a terrible way to test if a connection is 'secure'. To be correct we should perform a certificate validation.

The `QUrl` is converted to a string and the URL bar is updated with the value. Note that we also set the cursor position back to the beginning of the line to prevent the `QLineEdit` widget scrolling to the end.

```
211     def update_urlbar(self, q):
212 
213         if q.scheme() == 'https':
214             # Secure padlock icon
215             self.httpsicon.setPixmap( QPixmap( os.path.join('icons','lock-ssl.p\
216 ng') ) )
217 
218         else:
219             # Insecure padlock icon
220             self.httpsicon.setPixmap( QPixmap( os.path.join('icons','lock-nossl\
221 .png') ) )
222 
223         self.urlbar.setText( q.toString() )
224         self.urlbar.setCursorPosition(0)
```

It's also a nice touch to update the title of the application window with the title of the current page. We can get this via `browser.page().title()` which returns the contents of the `<title></title>` tag in the currently loaded web page.

```
149     def update_title(self):
150         title = self.browser.page().title()
151         self.setWindowTitle("%s - Mozarella Ashbadger" % title)
```

File operations

A standard File menu with `self.menuBar().addMenu("&File")` is created assigning the `F` key as an Alt-shortcut (as normal). Once we have the menu object, we can can assign `QAction` objects to create the entries. We create two basic entries here, for opening and saving HTML files (from a local disk). These both require custom slot functions.

```
110         file_menu = self.menuBar().addMenu("&File")
111 
112         open_file_action = QAction( QIcon( os.path.join('icons','disk--arrow.pn\
113 g') ), "Open file...", self)
114         open_file_action.setStatusTip("Open from file")
115         open_file_action.triggered.connect( self.open_file )
116         file_menu.addAction(open_file_action)
117 
118         save_file_action = QAction( QIcon( os.path.join('icons','disk--pencil.p\
119 ng') ), "Save Page As...", self)
120         save_file_action.setStatusTip("Save current page to file")
121         save_file_action.triggered.connect( self.save_file )
122         file_menu.addAction(save_file_action)
```

The slot function for opening a file uses the built-in `QFileDialog.getOpenFileName()` function to create a file-open dialog and get a name. We restrict the names by default to files matching `\*.htm` or `*.html`.

We read the file into a variable `html` using standard Python functions, then use `.setHtml()` to load the HTML into the browser.

```
164     def open_file(self):
165         filename, _ = QFileDialog.getOpenFileName(self, "Open file", "",
166                         "Hypertext Markup Language (*.htm *.html);;"
167                         "All files (*.*)")
168 
169         if filename:
170             with open(filename, 'r') as f:
171                 html = f.read()
172 
173             self.browser.setHtml( html )
174             self.urlbar.setText( filename )
```

Similarly to save the HTML from the current page, we use the built-in `QFileDialog.getSaveFile` to get a filename. However, this time we get the HTML from `self.browser.page().toHtml()` and write it to the selected filename. Again we use standard Python functions for the file handler.

```
178     def save_file(self):
179         filename, _ = QFileDialog.getSaveFileName(self, "Save Page As", "",
180                         "Hypertext Markup Language (*.htm *html);;"
181                         "All files (*.*)")
182 
183         if filename:
184             html = self.browser.page().toHtml()
185             with open(filename, 'w') as f:
186                 f.write(html)
```

Printing

We can add a print option to the File menu using the same approach we used earlier. Again this needs a custom slot function to perform the print action.

```
124         print_action = QAction( QIcon( os.path.join('icons','printer.png') ), "\
125 Print...", self)
126         print_action.setStatusTip("Print current page")
127         print_action.triggered.connect( self.print_page )
128         file_menu.addAction(print_action)
```

Qt provides a complete print framework. `QPrintPreviewDialog` to request the settings from the user. The dialog object has a `.paintRequested` signal, which we can connect to the print handler of the widget we wish to print. Thankfully, the `QWebEngineView` provides a compatible interface for us to connect to.

The `.paintRequested` signal will be triggered if the dialog is accepted, and the page will be printed.

```
190     def print_page(self):
191         dlg = QPrintPreviewDialog()
192         dlg.paintRequested.connect( self.browser.print_ )
193         dlg.exec_()
```

Help

Finally, to complete the standard interface we can add a Help menu. This is defined as before, two two custom slot functions to handle the display of an About dialog, and to load the 'browser page' with more information.

```
131         help_menu = self.menuBar().addMenu("&Help")
132 
133         about_action = QAction( QIcon( os.path.join('icons','question.png') ), \
134 "About Mozarella Ashbadger", self)
135         about_action.setStatusTip("Find out more about Mozarella Ashbadger") # \
136 Hungry!
137         about_action.triggered.connect( self.about )
138         help_menu.addAction(about_action)
139 
140         navigate_mozarella_action = QAction( QIcon( os.path.join('icons','lifeb\
141 uoy.png') ), "Mozarella Ashbadger Homepage", self)
142         navigate_mozarella_action.setStatusTip("Go to Mozarella Ashbadger Homep\
143 age")
144         navigate_mozarella_action.triggered.connect( self.navigate_mozarella )
145         help_menu.addAction(navigate_mozarella_action)
```

We define two methods to be used as slots for the Help menu signals. The first `navigate_mozzarella` opens up a page with more information on the browser (or in this case, this book). The second creates and executes a custom `QDialog` class `AboutDialog` which we will define next.

```
155     def navigate_mozarella(self):
156         self.browser.setUrl( QUrl("https://www.udemy.com/522076") )
157 
158     def about(self):
159         dlg = AboutDialog()
160         dlg.exec_()
```

The definition for the about dialog is given below. The structure follows that seen earlier in the book, with a `QDialogButtonBox` and associated signals to handle user input, and a series of `QLabels` to display the application information and a logo.

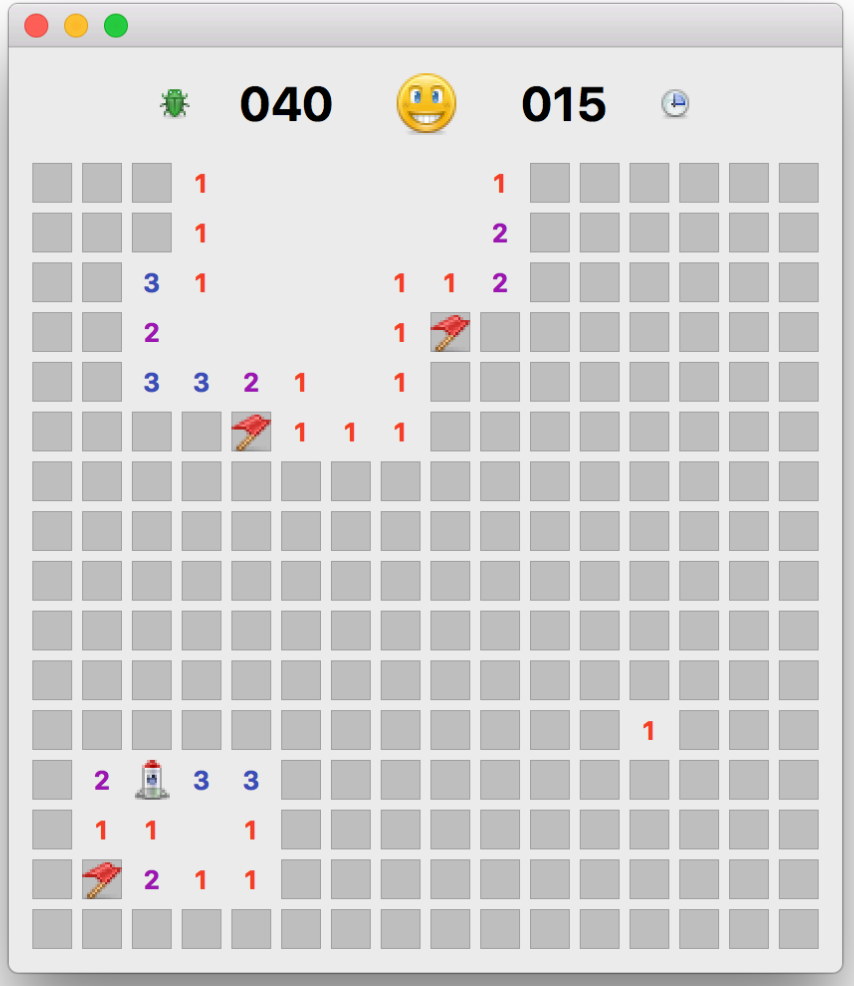
The only trick here is adding all the elements to the layout, then iterate over them to set the alignment to the center in a single loop. This saves duplication for the individual sections.

Moonsweeper

Explore the mysterious moon of Q'tee without getting too close to the alien natives!

Moonsweeper is a single-player puzzle video game. The objective of the game is to explore the area around your landed space rocket, without coming too close to the deadly B'ug aliens. Your trusty tri-counter will tell you the number of B'ugs in the vicinity.

This application makes use of features covered in Extended Signals.



Moonsweeper

This a simple single-player exploration game modelled on *Minesweeper* where you must reveal all the tiles without hitting hidden mines. This implementation uses custom `QWidget` objects for the tiles, which individually hold their state as mines,

status and the adjacent count of mines. In this version, the mines are replaced with alien bugs (B'ug) but they could just as easily be anything else.

In many *Minesweeper* variants the initial turn is considered a free go — if you hit a mine on the first click, it is moved somewhere else. Here we cheat a little bit by taking the first go for the player, ensuring that it is on a non-mine spot. This allows us not to worry about the bad first move which would require us to recalculate the adjacencies. We can explain this away as the "initial exploration around the rocket" and make it sound completely sensible.

If you want to implement this, you can catch the first click on a position and at that point generate mines/adjacencies, excluding your location, before handling the click. You will need to give your custom widgets access to the parent window object.

Source code

The full source for the *Moonsweeper* game is included in the downloads for this book. The game file is saved with the name `minesweeper.py`.

```
1 python3 minesweeper.py
```

It is recommend you download and take a quick look at the source code before continuing. The walkthrough below highlights the key (and interesting) parts of the code, but there is more to see.

Playing Field

The playing area for Moonsweeper is a NxN grid, containing a set number of mines. The dimensions and mine counts we'll used are taken from the default values for the Windows version of Minesweeper. The values used are shown in the table below:

.Table Dimensions and mine counts |=== |Level |Dimensions |Number of Mines |Easy |8 x 8 |10 |Medium |16 x 16 |40 |Hard |24 x 24 |99 |===

We store these values as a constant `LEVELS` defined at the top of the file. Since all the playing fields are square we only need to store the value once (8, 16 or 24).

```
LEVELS = [
    ("Easy", 8, 10),
    ("Medium", 16, 40),
    ("Hard", 24, 99)
]
```

The playing grid could be represented in a number of ways, including for example a 2D 'list of lists' representing the different states of the playing positions (mine, revealed, flagged).

However, in our implementation we'll be using an object-orientated approach, where individual positions on the map hold all relevant data about themselves. Taking this a step further, we can make these objects individually responsible for drawing themselves. In Qt we can do this simply by subclassing from `QWidget` and then implementing a custom paint function.

We'll cover the construction and behaviour of these custom widgets before moving onto its appearance. Since our tile objects are subclassing from `QWidget` we can lay them out like any other widget. We do this, by setting up a `QGridLayout`.

```
        self.grid = QGridLayout()
        self.grid.setSpacing(5)
        self.grid.setSizeConstraint(QLayout.SetFixedSize)
```

Next we need to set up the playing field, creating our position tile widgets and adding them our grid. The initial setup for the level is defined in custom method, which reads from `LEVELS` and assigns a number of variables to the window. The window title and mine counter are updated, and then the setup of the grid is begun.

```
234     def set_level(self, level):
235         self.level_name, self.b_size, self.n_mines = LEVELS[level]
236 
237         self.setWindowTitle("Moonsweeper - %s" % (self.level_name))
238         self.mines.setText("%03d" % self.n_mines)
239 
240         self.clear_map()
241         self.init_map()
242         self.reset_map()
```

The setup functions will be covered next.

We're using a custom `Pos` class here, which we'll look at in detail later. For now you just need to know that this holds all the relevant information for the relevant position in the map — including, for example, whether it's a mine, revealed, flagged and the number of mines in the immediate vicinity.

Each `Pos` object also has 3 custom signals *clicked*, *revealed* and *expandable* which we connect to custom slot methods. Finally, we call resize to adjust the size of the window to the new contents. Note that this is actually only necessary when the window *shrinks* — it will grow automatically.

```
257     def init_map(self):
258         # Add positions to the map
259         for x in range(0, self.b_size):
260             for y in range(0, self.b_size):
261                 w = Pos(x,y)
262                 self.grid.addWidget(w, y, x)
263                 # Connect signal to handle expansion.
264                 w.clicked.connect(self.trigger_start)
265                 w.revealed.connect(self.on_reveal)
266                 w.expandable.connect(self.expand_reveal)
267 
268         # Place resize on the event queue, giving control back to Qt before.
269         QTimer.singleShot(0, lambda: self.resize(1,1)) # <1>
```

1. The `singleShot` timer is required to ensure the resize runs after Qt is aware of the new contents. By using a timer we guarantee control will return to Qt *before* the resize occurs.

We also need to implement the inverse of the `init_map` function to remove tile objects from the map. Removing tiles will be necessary when moving from a higher to a lower level. It would be possible to be a little smarter here and adding/removing only those tiles that are necessary to get to the correct size. But, since we already have the function to add all up to the right size, we can cheat a bit.

Update this code to add/remove the neccessary tiles to size the new level dimensions.

Notice that we both remove the item from the grid with `self.grid.removeItem(c)` and clear the parent `c.widget().setParent(None)`. This second step is necessary, since adding the items assigning them the parent window as a parent. Just removing them leaves them floating in the window outside the layout.

```
246     def clear_map(self):
247         # Remove all positions from the map, up to maximum size.
248         for x in range(0, LEVELS[-1][1]): # <1>
249             for y in range(0, LEVELS[-1][1]):
250                 c = self.grid.itemAtPosition(y,x)
251                 if c: # <2>
252                     self.grid.removeItem(c)
253                     c.widget().setParent(None)
```

1. To ensure we clear all sizes of maps we take the dimension of the highest level.
2. If there isn't anything in the grid at this location, we can skip it.

Now we have our grid of positional tile objects in place, we can begin creating the initial conditions of the playing board. This process is rather complex, so it's broken down into a number of functions. We name them `_reset` (the leading underscore is a convention to indicate a private function, not intended for external use). The main function `reset_map` calls these functions in turn to set it up.

The process is as follows —

1. Remove all mines (and reset data) from the field.
2. Add new mines to the field.
3. Calculate the number of mines adjacent to each position.
4. Add a starting marker (the rocket) and trigger initial exploration.
5. Reset the timer.

```
273     def reset_map(self):
274         self._reset_position_data()
275         self._reset_add_mines()
276         self._reset_calculate_adjacency()
277         self._reset_add_starting_marker()
278         self.update_timer()
```

The separate steps from 1-5 are described in detail in turn below, with the code for each step.

The first step is to reset the data for each position on the map. We iterate through every position on the board, calling `.reset()` on the widget at each point. The code for the `.reset()` function is defined on our custom `Pos` class, we'll explore in detail later. For now it's enough to know it clears mines, flags and sets the position back to being unrevealed.

```
    def _reset_position_data(self):
        # Clear all mine positions
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = self.grid.itemAtPosition(y, x).widget()
                w.reset()
```

Now all the positions are blank, we can begin the process of adding mines to the map. The maximum number of mines `n_mines` is defined by the level settings, described earlier.

```
    def _reset_add_mines(self):
        # Add mine positions
        positions = []
        while len(positions) < self.n_mines:
            x, y = random.randint(0, self.b_size-1), random.randint(0, self.b_s\
ize-1)
            if (x ,y) not in positions:
                w = self.grid.itemAtPosition(y,x).widget()
                w.is_mine = True
                positions.append((x, y))

        # Calculate end-game condition
        self.end_game_n = (self.b_size * self.b_size) - (self.n_mines + 1)
        return positions
```

With mines in position, we can now calculate the 'adjacency' number for each position — simply the number of mines in the immediate vicinity, using a 3x3 grid around the given point. The custom function `get_surrounding` simply returns those positions around a given `x` and `y` location. We count the number of these that is a mine `is_mine == True` and store.

Pre-calculating the adjacent counts in this way helps simplify the reveal logic later.

```
    def _reset_calculate_adjacency(self):

        def get_adjacency_n(x, y):
            positions = self.get_surrounding(x, y)
            return sum(1 for w in positions if w.is_mine)

        # Add adjacencies to the positions
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = self.grid.itemAtPosition(y, x).widget()
                w.adjacent_n = get_adjacency_n(x, y)
```

A starting marker is used to ensure that the first move is *always_valid. This is implemented as a _brute force* search through the grid space, effectively trying random positions until we find a position which is not a mine. Since we don't know how many attempts this will take, we need to wrap it in a continuous loop.

Once that location is found, we mark it as the start location and then trigger the exploration of all surrounding positions. We break out of the loop, and reset the ready status.

```
    def _reset_add_starting_marker(self):
        # Place starting marker.

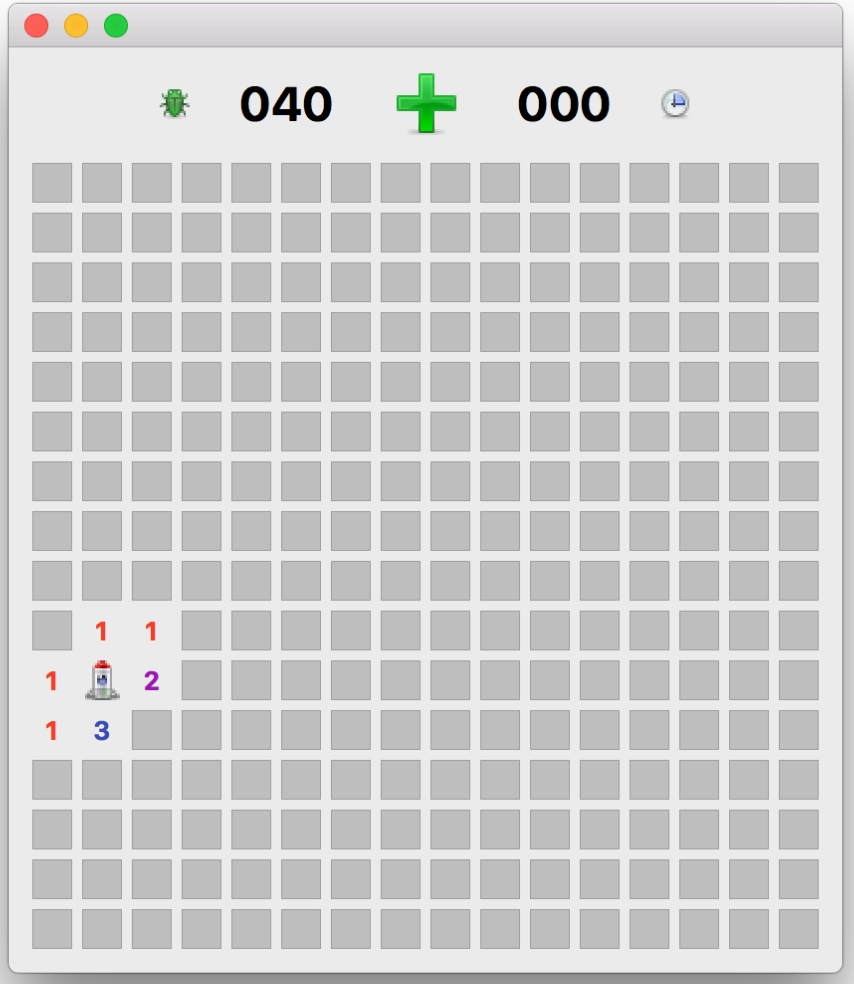
        # Set initial status (needed for .click to function)
        self.update_status(STATUS_READY)

        while True:
            x, y = random.randint(0, self.b_size - 1), random.randint(0, self.b\
_size - 1)
            w = self.grid.itemAtPosition(y, x).widget()
            # We don't want to start on a mine.
            if not w.is_mine:
                w.is_start = True
                w.is_revealed = True
                w.update()

                # Reveal all positions around this, if they are not mines eithe\
```

```
r.
                for w in self.get_surrounding(x, y):
                    if not w.is_mine:
                        w.click()
                break

        # Reset status to ready following initial clicks.
        self.update_status(STATUS_READY)
```



Initial exporation around the rocket.

Position Tiles

As previously described, we've structured the game so that individual tile positions hold their own state information. This means that `Pos` objects are ideally positioned to handle game logic which reacts to to interactions that relate to their own state — in other words, this is where the magic is.

Since the `Pos` class is relatively complex, it is broken down here in to main themes, which are discussed in turn. The initial setup `__init__` block is simple, accepting an `x` and `y` position and storing it on the object. `Pos` positions never change once created.

To complete setup the `.reset()` function is called which resets all object attributes back to default, zero values. This flags the mine as_not the start position, *_not a mine*, *not revealed* and *not flagged*. We also reset the adjacent count.

```
class Pos(QWidget):

    expandable = pyqtSignal(int,int)
    revealed = pyqtSignal(object)
    clicked = pyqtSignal()

    def __init__(self, x, y, *args, **kwargs):
        super(Pos, self).__init__(*args, **kwargs)

        self.setFixedSize(QSize(20, 20))
        self.x = x
        self.y = y
        self.reset()

    def reset(self):
        self.is_start = False
        self.is_mine = False
        self.adjacent_n = 0
        self.is_revealed = False
        self.is_flagged = False

        self.update()
```

Gameplay is centered around mouse interactions with the tiles in the playing field, so detecting and reacting to mouse clicks is central. In Qt we catch mouse clicks by detecting the `mouseReleaseEvent`. To do this for our custom `Pos` widget we define a handler on the class. This receives `QMouseEvent` with the information containing what happened. In this case we are only interested in whether the mouse release occurred from the left or the right mouse button.

For a left mouse click we check whether the tile is flagged or already revealed. If it is either, we ignore the click — making flagged tiles 'safe', unable to be click by accident. If the tile is not flagged we simply initiation the `.click()` method (see later).

For a right mouse click, on tiles which are *not* revealed, we call our `.toggle_flag()` method to toggle a flag on and off.

```
    def mouseReleaseEvent(self, e):

        if (e.button() == Qt.RightButton and not self.is_revealed):
            self.toggle_flag()

        elif (e.button() == Qt.LeftButton):
            # Block clicking on flagged mines.
            if not self.is_flagged and not self.is_revealed:
                self.click()
```

The methods called by the `mouseReleaseEvent` handler are defined below.

The `.toggle_flag` handler simply sets `.is_flagged` to the inverse of itself (`True` becomes `False`, `False` becomes `True`) having the effect of toggling it on and off. Note that we have to call `.update()` to force a redraw having changed the state. We also emit our custom `.clicked` signal, which is used to start the timer — because placing a flag should also count as starting, not just revealing a square.

The `.click()` method handles a left mouse click, and in turn triggers the reveal of the square. If the number of adjacent mines to this `Pos` is zero, we trigger the `.expandable` signal to begin the process of auto-expanding the region explored (see later). Then, we again emit `.clicked` to signal the start of the game.

Finally, the `.reveal()` method checks whether the tile is already revealed, and if not sets `.is_revealed` to `True`. Again we call `.update()` to trigger a repaint of the widget.

The optional emit of the `.revealed` signal is used only for the endgame full-map reveal. Because each reveal triggers a further lookup to find what tiles are also revealable, revealing the entire map would create a large number of redundant callbacks. By suppressing the signal here we avoid that.

```
    def toggle_flag(self):
        self.is_flagged = not self.is_flagged
        self.update()

        self.clicked.emit()

    def click(self):
        self.reveal()
        if self.adjacent_n == 0:
            self.expandable.emit(self.x, self.y)

        self.clicked.emit()

    def reveal(self, emit=True):
        if not self.is_revealed:
            self.is_revealed = True
            self.update()

            if emit:
                self.revealed.emit(self)
```

Finally, we define a custom `paintEvent` method for our `Pos` widget to handle the display of the current position state. As described in chapter to perform custom paint over a widget canvas we take a `QPainter` and the `event.rect()` which provides the boundaries in which we are to draw — in this case the outer border of the `Pos` widget.

Revealed tiles are drawn differently depending on whether the tile is a *start position*, *bomb* or *empty space*. The first two are represented by icons of a rocket and bomb respectively. These are drawn into the tile `QRect` using `.drawPixmap`. Note we need to convert the `QImage` constants to pixmaps, by passing through `QPixmap` by passing.

You may think "why not just store these as `QPixmap` objects since that's what we're using? We can't do this and store them in constants because you can't create `QPixmap` objects before a `QApplication` instance is up and running.

For empty positions (not rockets, not bombs) we optionally show the adjacency number if it is larger than zero. To draw text onto our `QPainter` we use `.drawText()` passing in the `QRect`, alignment flags and the number to draw as a string. We've defined a standard color for each number (stored in `NUM_COLORS`) for usability.

For tiles that are *not* revealed we draw a tile, by filling a rectangle with light gray and draw a 1 pixel border of darker grey. If `.is_flagged` is set, we also draw a flag icon over the top of the tile using `drawPixmap` and the tile `QRect`.

```
    def paintEvent(self, event):
        p = QPainter(self)
        p.setRenderHint(QPainter.Antialiasing)

        r = event.rect()

        if self.is_revealed:
            if self.is_start:
                p.drawPixmap(r, QPixmap(IMG_START))

            elif self.is_mine:
                p.drawPixmap(r, QPixmap(IMG_BOMB))

            elif self.adjacent_n > 0:
                pen = QPen(NUM_COLORS[self.adjacent_n])
                p.setPen(pen)
                f = p.font()
                f.setBold(True)
                p.setFont(f)
                p.drawText(r, Qt.AlignHCenter | Qt.AlignVCenter, str(self.adjac\
ent_n))

        else:
            p.fillRect(r, QBrush(Qt.lightGray))
```

```
            pen = QPen(Qt.gray)
            pen.setWidth(1)
            p.setPen(pen)
            p.drawRect(r)

            if self.is_flagged:
                p.drawPixmap(r, QPixmap(IMG_FLAG))
```

Mechanics

We commonly need to get all tiles surrounding a given point, so we have a custom function for that purpose. It simple iterates across a 3x3 grid around the point, with a check to ensure we do not go out of bounds on the grid edges (`0 ≥ x ≤ self.b_size`). The returned list contains a `Pos` widget from each surrounding location.

```
    # tag::surrounding[]
    def get_surrounding(self, x, y):
        positions = []

        for xi in range(max(0, x - 1), min(x + 2, self.b_size)):
            for yi in range(max(0, y - 1), min(y + 2, self.b_size)):
                if not (xi == x and yi == y):
                    positions.append( self.grid.itemAtPosition(yi, xi).widget()\
 )

        return positions
```

The `expand_reveal` method is triggered in response to a click on a tile with zero adjacent mines. In this case we want to expand the area around the click to any spaces which also have zero adjacent mines, and also reveal any squares around the border of that expanded area (which aren't mines).

We start with a list `to_expand` containing the positions to check on the next iteration, a list `to_reveal` containing the tile widgets to reveal, and a flag `any_added` to determine when to exit the loop. The loop stops the first time no new widgets are added to `to_reveal`.

Inside the loop we reset `any_added` to `False`, and empty the `to_expand` list, keeping a temporary store in `l` for iterating over.

For each `x` and `y` location we get the 8 surrounding widgets. If any of these widgets is not a mine, and is not already in the `to_reveal` list we add it. This ensures that the edges of the expanded area are all revealed. If the position has no adjacent mines, we append the coordinates onto `to_expand` to be checked on the next iteration.

By adding any non-mine tiles to `to_reveal`, and only expanding tiles that are not already in `to_reveal`, we ensure that we won't visit a tile more than once.

```
    def expand_reveal(self, x, y):
        """
        Iterate outwards from the initial point, adding new locations to the
        queue. This allows us to expand all in a single go, rather than
        relying on multiple callbacks.
        """
        to_expand = [(x,y)]
        to_reveal = []
        any_added = True

        while any_added:
            any_added = False
            to_expand, l = [], to_expand

            for x, y in l:
                positions = self.get_surrounding(x, y)
                for w in positions:
                    if not w.is_mine and w not in to_reveal:
                        to_reveal.append(w)
                        if w.adjacent_n == 0:
                            to_expand.append((w.x,w.y))
                            any_added = True

        # Iterate an reveal all the positions we have found.
        for w in to_reveal:
            w.reveal()
```

Endgames

Endgame states are detected during the reveal process following a click on a title. There are two possible outcomes —

1. Tile is a mine, game over.
2. Tile is not a mine, decrement the `self.end_game_n`.

This continues until `self.end_game_n` reaches zero, which triggers the win game process by calling either `game_over` or `game_won`. Success/failure is triggered by revealing the map and setting the relevant status, in both cases.

```
    def on_reveal(self, w):
        if w.is_mine:
            self.game_over()

        else:
            self.end_game_n -= 1 # decrement remaining empty spaces

            if self.end_game_n == 0:
                self.game_won()

    def game_over(self):
        self.reveal_map()
        self.update_status(STATUS_FAILED)

    def game_won(self):
        self.reveal_map()
        self.update_status(STATUS_SUCCESS)
```

Further ideas

If you want to have a go at expanding *Moonsweeper*, here are a few ideas —

1. Allow the player to take their own first turn. Try postponing the calculation of mine positions til after the user first clicks, and then generate positions until you get a miss.

2. Add power-ups, e.g. a scanner to reveal a certain area of the board automatically.
3. Let the hidden B'ugs move around between each turn. Keep a list of free-unrevealed positions, and allow the B'ugs to move into them. You'll need to recalculate the adjacencies after each click.

Packaging PyQt Applications

There is not much fun in creating your own applications if you can't share it with other people — whether that means publishing it commercially, sharing it online or just giving it to someone you know. Sharing your apps allows other people to benefit from your hard work!

Packaging Python applications for distribution has typically been a little tricky, particularly when targeting multiple platforms (Windows, MacOS and Linux). This is in part because of the need to bundle the source, data files, the Python runtime and all associated libraries in a way that will work reliably on the target system.

The good news is that there is a package build system designed specifically for PyQt — fbs. This simplifies and automates much of the build process to ensure reliable and reproducible cross-platform packages.

fbs only works with PyQt5 or PySide2 and Python version > 3.5. However, it is built on top of PyInstaller which works with earlier versions of both. If you are still on Python 2.7 you may wish to consult the PyInstaller manual directly.

fbs: fman Build System

fbs is a cross-platform PyQt5 packaging system which supports building desktop applications for Windows, Mac and Linux (Ubuntu, Fedora and Arch). Built on top of *PyInstaller* it wraps some of the rough edges and defines a standard project structure which allows the build process to be entirely automated. The included resource API is particularly useful, simplifying the handling of external data files, images or third-party libraries — a common pain point when bundling apps.

fbs is licensed under the GPL. This means you can use the **fbs** system for free in open-source packages distributed with the GPL. For commercial (or non-GPL) packages you must buy a commercial license. See the fbs licensing page for up-to-date information.

If you're impatient, you can grab the Moonsweeper installers directly for Windows, MacOS or Linux (Ubuntu).

Requirements

fbs works out of the box with both PyQt `PyQt5` and Qt for Python `PySide2`. The only other requirement is `PyInstaller` which handles the packaging itself. You can install these in a virtual environment (or your applications virtual environment) to keep your environment clean.

fbs only supports Python versions 3.5 and 3.6

```
python3 -m venv fbsenv
```

Once created, activate the virtual environment by running from the command line —

```
# On Mac/Linux:
source fbsenv/bin/activate

# On Windows:
call fbsenv\scripts\activate.bat
```

Finally, install the required libraries. For PyQt5 you would use —

```
pip3 install fbs PyQt5 PyInstaller==3.4
```

Or for Qt for Python (PySide2) —

```
pip3 install fbs PySide2 PyInstaller==3.4
```

fbs installs a command line tool `fbs` into your path which provides access to all **fbs ** management commands. To see the complete list of commands available run `fbs`.

```
martin@Martins-Laptop testapp $ fbs
usage: fbs [-h]
           {startproject,run,freeze,installer,sign_installer,repo,upload,releas\
e,test,clean,buildvm,runvm,gengpgkey,register,login,init_licensing}
           ...

fbs

positional arguments:
  {startproject,run,freeze,installer,sign_installer,repo,upload,release,test,cl\
ean,buildvm,runvm,gengpgkey,register,login,init_licensing}
    startproject        Start a new project in the current directory
    run                 Run your app from source
    freeze              Compile your code to a standalone executable
    installer           Create an installer for your app
    sign_installer      Sign installer, so the user's OS trusts it
    repo                Generate files for automatic updates
    upload              Upload installer and repository to fbs.sh
    release             Bump version and run clean,freeze,...,upload
    test                Execute your automated tests
    clean               Remove previous build outputs
    buildvm             Build a Linux VM. Eg.: buildvm ubuntu
    runvm               Run a Linux VM. Eg.: runvm ubuntu
    gengpgkey           Generate a GPG key for Linux code signing
    register            Create an account for uploading your files
    login               Save your account details to secret.json
    init_licensing      Generate public/private keys for licensing

optional arguments:
  -h, --help            show this help message and exit
```

Starting an app

If you're starting a PyQt5 application from scratch, you can use the `fbs startproject` management command to create a complete, working and packageable application stub in the current folder. This has the benefit of allowing you to test (and continue to test) the packageability of your application as you develop it, rather than leaving it to the end.

```
fbs startproject
```

The command walks you through a few questions, allowing you to fill in details of your application. These values will be written into your app source and configuration. The bare-bones app will be created under the `src/` folder in the current directory.

```
martin@Martins-Laptop ~ $ fbs startproject
App name [MyApp] : HelloWorld
Author [Martin] : Martin Fitzpatrick
Mac bundle identifier (eg. com.martin.helloworld, optional):
```

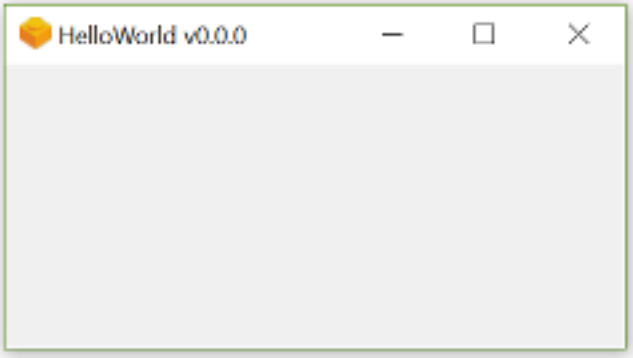
If you already have your own working PyQt5 app you will need to either a) use the generated app as a guideline for converting yours to the same structure, or b) create a new app using `startproject` and migrate the code over.

Running your new project

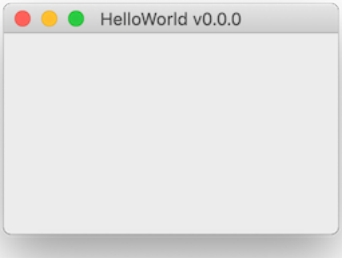
You can run this new application using the following **fbs** command in the same folder you ran `startproject` from.

```
fbs run
```

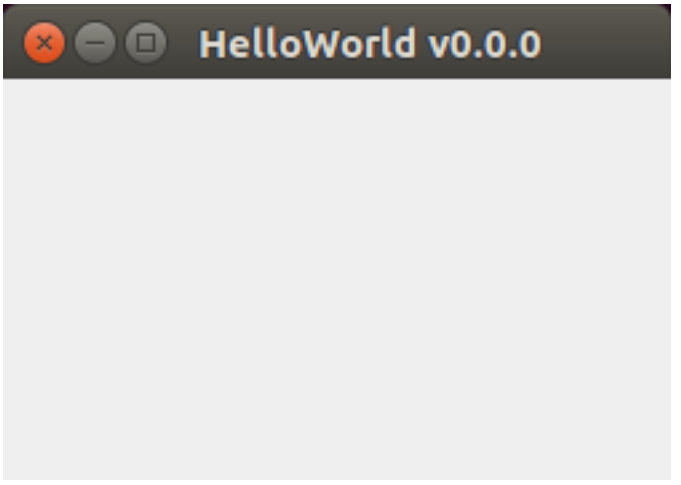
If everything is working this should show you a small empty window with your apps' title — exciting eh?



HelloWorld on Windows



HelloWorld on Mac



HelloWorld on Ubuntu

The application structure

The `startproject` command generates the required folder structure for a **fbs** PyQt5 application. This includes a `src/build` which contains the build settings for your package, `main/icons` which contains the application icons, and `src/python` for the source.

```
.
└── src
    ├── build
    │   └── settings
    │       ├── base.json
    │       ├── linux.json
    │       └── mac.json
    └── main
        ├── icons
        │   ├── Icon.ico
        │   ├── README.md
        │   ├── base
        │   │   ├── 16.png
        │   │   ├── 24.png
        │   │   ├── 32.png
```

```
16          │   │   ├── 48.png
17          │   │   └── 64.png
18          │   ├── linux
19          │   │   ├── 1024.png
20          │   │   ├── 128.png
21          │   │   ├── 256.png
22          │   │   └── 512.png
23          │   └── mac
24          │       ├── 1024.png
25          │       ├── 128.png
26          │       ├── 256.png
27          │       └── 512.png
28          └── python
29              └── main.py
```

Your bare-bones PyQt5 application is generated in `src/main/python/main.py` and is a complete working example you can use to base your own code on.

```
1  from fbs_runtime.application_context import ApplicationContext
2  from PyQt5.QtWidgets import QMainWindow
3
4  import sys
5
6  class AppContext(ApplicationContext):           # 1. Subclass ApplicationContext
7      def run(self):                              # 2. Implement run()
8          window = QMainWindow()
9          version = self.build_settings['version']
10         window.setWindowTitle("HelloWorld v" + version)
11         window.resize(250, 150)
12         window.show()
13         return self.app.exec_()                 # 3. End run() with this line
14
15 if __name__ == '__main__':
16     appctxt = AppContext()                      # 4. Instantiate the subclass
17     exit_code = appctxt.run()                   # 5. Invoke run()
18     sys.exit(exit_code)
```

If you've built PyQt5 applications before you'll notice that building an application with **fbs** introduces a new concept — the `ApplicationContext`.

The `ApplicationContext`

When building PyQt5 applications there are typically a number of components or resources that are used throughout your app. These are commonly stored in the `QMainWindow` or as global vars which can get a bit messy as your application grows. The `ApplicationContext` provides a central location for initialising and storing these components, as well as providing access to some core **fbs** features.

The `ApplicationContext` object also creates and holds a reference to a global `QApplication` object — available under `ApplicationContext.app`. Every Qt application must have one (and only one) `QApplication` to hold the event loop and core settings. Without **fbs** you would usually define this at the base of your script, and call `.exec()` to start the event loop.

Without **fbs** this would look something like this —

```
if __name__ == '__main__':
                app = QApplication()
                w = MyCustomWindow()
          app.exec_()
```

The equivalent with **fbs** would be —

```
if __name__ == '__main__':
                ctx = ApplicationContext()
                w = MyCustomWindow()
                ctx.app.exec_()
```

If you want to create your own custom `QApplication` initialisation you can overwrite the `.app` property on your `ApplicationContext` subclass using `cached_property` (see below).

This basic example is clear to follow. However, once you start adding custom styles and translations to your application the initialisation can grow quite a bit. To keep things nicely structured **fbs** recommends creating a `.run` method on your `ApplicationContext`.

This method should handle the setup of your application, such as creating and showing a window, finally starting up the event loop on the `.app` object. This final step is performed by calling `self.app.exec_()` at the end of the method.

```
class AppContext(ApplicationContext):
    def run(self):
        ...
        return self.app.exec_()
```

As your initialisation gets more complicated you can break out subsections into separate methods for clarity, for example —

```
class AppContext(ApplicationContext):
    def run(self):
                self.setup_fonts()
                self.setup_styles()
                self.setup_translations()
        return self.app.exec_()

        def setup_fonts(self):
        # ...do something ...

        def setup_styles(self):
        # ...do something ...

        def setup_translations(self):
        # ...do something ...
```

On execution the `.run()` method will be called and your event loop started. Execution continues in this event loop until the application is exited, at which point your `.run()` method will return (with the appropriate exit code).

Building a real application

The bare-bones application doesn't do very much, so below we'll look at something more complete — the *Moonsweeper* application from the previous chapter. The modified source code is available to download here.

Only the changes required to convert *Moonsweeper* over to **fbs** are covered here. If you want to see how *Moonsweeper* itself works, see the previous chapter. The custom application icons were created using icon art by Freepik.

The project follows the same basic structure as for the stub application we created above.

```
.
├── README.md
├── requirements.txt
├── screenshot-minesweeper1.jpg
├── screenshot-minesweeper2.jpg
└── src
    ├── build
    │   └── settings
    │       ├── base.json
    │       ├── linux.json
    │       └── mac.json
    └── main
        ├── Installer.nsi
        ├── icons
        │   ├── Icon.ico
        │   ├── README.md
        │   ├── base
        │   │   ├── 16.png
        │   │   ├── 24.png
        │   │   ├── 32.png
        │   │   ├── 48.png
        │   │   └── 64.png
        │   ├── linux
```

```
24          │   │   ├── 1024.png
25          │   │   ├── 128.png
26          │   │   ├── 256.png
27          │   │   └── 512.png
28          │   └── mac
29          │       ├── 1024.png
30          │       ├── 128.png
31          │       ├── 256.png
32          │       └── 512.png
33          ├── python
34          │   ├── __init__.py
35          │   └── main.py
36          └── resources
37              ├── base
38              │   └── images
39              │       ├── bomb.png
40              │       ├── bug.png
41              │       ├── clock-select.png
42              │       ├── cross.png
43              │       ├── flag.png
44              │       ├── plus.png
45              │       ├── rocket.png
46              │       ├── smiley-lol.png
47              │       └── smiley.png
48              └── mac
49                  └── Contents
50                      └── Info.plist
```

The `src/build/settings/base.json` stores the basic details about the application, including the entry point to run the app with `fbs run` or once packaged.

```
{
    "app_name": "Moonsweeper",
    "author": "Martin Fitzpatrick",
    "main_module": "src/main/python/main.py",
    "version": "0.0.0"
}
```

The script *entry point* is at the base of `src/main/python/main.py`. This creates the `AppContext` object and calls the `.run()` method to start up the app.

```
if __name__ == '__main__':
    appctxt = AppContext()
    exit_code = appctxt.run()
    sys.exit(exit_code)
```

The `ApplicationContext` defines a `.run()` method to handle initialisation. In this case that consists of creating and showing the main window, then starting up the event loop.

```
from fbs_runtime.application_context import ApplicationContext, \
    cached_property


class AppContext(ApplicationContext):
    def run(self):
        self.main_window.show()
        return self.app.exec_()

    @cached_property
    def main_window(self):
        return MainWindow(self)  # Pass context to the window.

        # ... snip ...
```

The `cached_property` decorator

The `.run()` method accesses `self.main_window`. You'll notice that this method is wrapped in an **fbs** `@cached_property` decorator. This decorator turns the method into a property (like the Python `@property` decorator) and caches the return value.

The first time the property is accessed the method is executed and the return value cached. On subsequent calls, the cached value is returned directly without executing anything. This also has the side-effect of postponing creation of these objects until they are needed.

You can use `@cached_property` to define each application component (a window, a toolbar, a database connection or other resources). However, you don't *have* to use the `@cached_property` — you could alternatively declare all properties in your `ApplicationContext.__init__` block as shown below.

```
from fbs_runtime.application_context import ApplicationContext

class AppContext(ApplicationContext):

        def __init__(self, *args, **kwargs):
                super(AppContent, self).__init__(*args, **kwargs)

                self.window = Window()

    def run(self):
        self.window.show()
        return self.app.exec_()
```

Accessing resources with `.get_resource`

Applications usually require additional data files beyond the source code — for example files icons, images, styles (Qt's `.qss` files) or documentation. You may also want to bundle platform-specific libraries or binaries. To simplify this **fbs** defines a folder structure and access method which work seamlessly across development and distributed versions.

The top level folder `resources/` should contain a folder `base` plus any combination of the other folders shown below. The `base` folder contains files common to all

platforms, while the platform-specific folders can be used for any files specific to a given OS.

```
base/     # for files required on all OSs
windows/   # for files only required on Windows
mac/      # "   "       "    "         "  Mac
linux/    # "   "       "    "         "  Linux
arch/     # "   "       "    "         "  Arch Linux
fedora/   # "   "       "    "         "  Debian Linux
ubuntu/   # "   "       "    "         "  Ubuntu Linux
```

Getting files into the right place to load from a distributed app across all platforms is usually one of the faffiest bits of distributing PyQt applications. It's really handy that **fbs** handles this for you.

To simplify the loading of resources from your `resources/` folder in your applications **fbs** provides the `ApplicationContext.get_resource()` method. This method takes the name of a file which can be found somewhere in the `resources/` folder and returns the absolute path to that file. You can use this returned absolute path to open the file as normal.

```
from fbs_runtime.application_context import ApplicationContext, cached_property


class AppContext(ApplicationContext):

        # ... snip ...

    @cached_property
    def img_bomb(self):
        return QImage(self.get_resource('images/bug.png'))

    @cached_property
    def img_flag(self):
        return QImage(self.get_resource('images/flag.png'))
```

```
15
16      @cached_property
17      def img_start(self):
18          return QImage(self.get_resource('images/rocket.png'))
19
20      @cached_property
21      def img_clock(self):
22          return QImage(self.get_resource('images/clock-select.png'))
23
24      @cached_property
25      def status_icons(self):
26          return {
27              STATUS_READY: QIcon(self.get_resource("images/plus.png")),
28              STATUS_PLAYING: QIcon(self.get_resource("images/smiley.png")),
29              STATUS_FAILED: QIcon(self.get_resource("images/cross.png")),
30              STATUS_SUCCESS: QIcon(self.get_resource("images/smiley-lol.png"))
31          }
32
33          # ... snip ...
```

In our *Moonsweeper* application above, we have a *bomb* image file available at `src/main/resources/base/images/bug.jpg`. By calling `ctx.get_resource('images/bug.png')` we get the absolute path to that image file on the filesystem, allowing us to open the file within our app.

If the file does not exist `FileNotFoundError` will be raised instead.

The handy thing about this method is that it transparently handles the platform folders under `src/main/resources` giving OS-specific files precedence. For example, if the same file was also present under `src/main/resources/mac/images/bug.jpg` and we called `ctx.get_resource('images/bug.jpg')` we would get the Mac version of the file.

Additionally `get_resource` works both when running from source and when running a frozen or installed version of your application. If your `resources/` load correctly locally you can be confident they will load correctly in your distributed applications.

Using the `ApplicationContext` from app

As shown above, our `ApplicationContext` object has cached properties to load and return the resources. To allow us to access these from our `QMainWindow` we can pass the context in and store a reference to it in our window `__init__`.

```
class MainWindow(QMainWindow):
    def __init__(self, ctx):
        super(MainWindow, self).__init__()

                self.ctx = ctx  # Store a reference to the context for resources, et

# ... snip ...
```

Now that we have access to the context via `self.ctx` we can use it this in any place we want to reference these external resources.

```
        l = QLabel()
        l.setPixmap(QPixmap.fromImage(self.ctx.img_bomb))
        l.setAlignment(Qt.AlignRight | Qt.AlignVCenter)
        hb.addWidget(l)

# ... snip ...

        l = QLabel()
        l.setPixmap(QPixmap.fromImage(self.ctx.img_clock))
        l.setAlignment(Qt.AlignLeft | Qt.AlignVCenter)
        hb.addWidget(l)
```

The first time we access `self.ctx.img_bomb` the file will be loaded, the `QImage` created and returned. On subsequent calls, we'll get the image from the cache.

```
    def init_map(self):
        # Add positions to the map
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = Pos(x, y, self.ctx.img_flag, self.ctx.img_start, self.ctx.i\
mg_bomb)
                self.grid.addWidget(w, y, x)
                # Connect signal to handle expansion.
                w.clicked.connect(self.trigger_start)
                w.expandable.connect(self.expand_reveal)
                w.ohno.connect(self.game_over)

# ... snip ...

        self.button.setIcon(self.ctx.status_icons[STATUS_PLAYING])

# ... snip ...

    def update_status(self, status):
        self.status = status
        self.button.setIcon(self.ctx.status_icons[self.status])
```

Those are all the changes needed to get the *Moonsweeper* app packageable with **fbs**. If you open up the source folder you should be able to start it up as before.

```
fbs run
```

If that's working, you're ready to move onto freezing and building in the installer.

Freezing the app

Freezing is the process of turning a Python application into a standalone executable that can run on another user's computer. Use the following command to turn the app's source code into a standalone executable:

```
fbs freeze
```

The resulting executable depends on the platform you *freeze* on — the executable will only work on the OS you built it on (e.g. an executable built on Windows will run on another Windows computer, but not on a Mac).

- Windows will create an `.exe` executable in the folder `target/<AppName>`
- MacOS X will create an `.app` application bundle in `target/<AppName>.app`
- Linux will create an executable in the folder `target/<AppName>`

On Windows you may need to install the Windows 10 SDK, although **fbs** will prompt you if this is the case.

Creating an installer

While you can share the executable files with users, desktop applications are normally distributed with *installers* which handle the process of putting the executable (and any other files) in the correct place. See the following sections for platform-specific notes before creating

You must *freeze* your app first *then* create the installer.

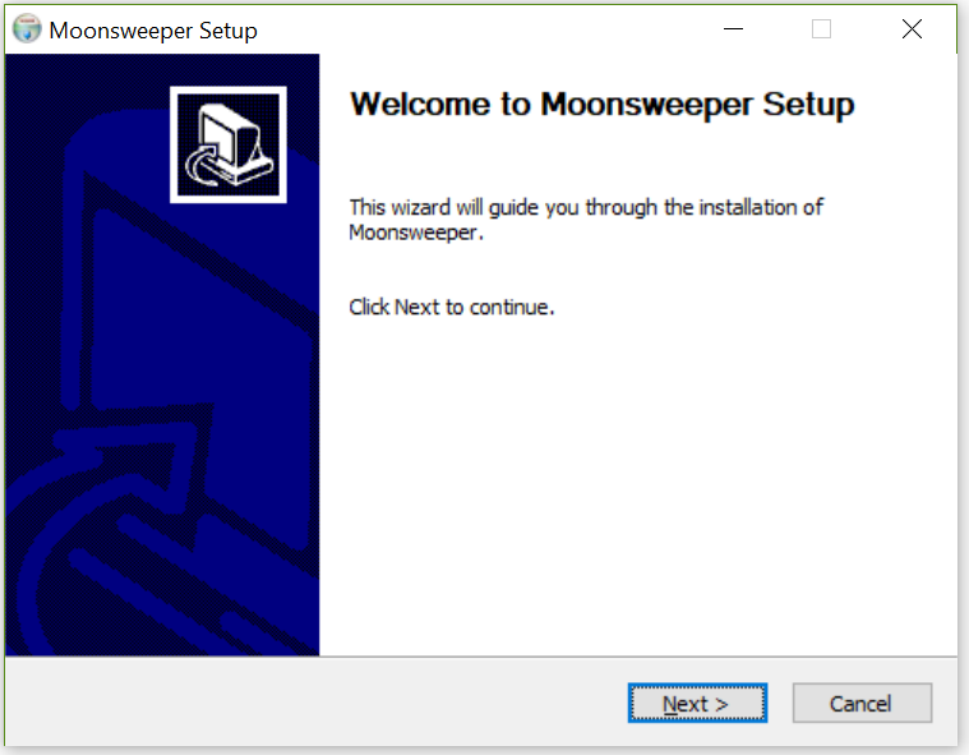
Windows installer

The Windows installer allows your users to pick the installation directory for the executable and adds your app to the user's Start Menu. The app is also added to installed programs, allowing it to be uninstalled by your users.

Before you create installers on Windows you will need to install NSIS and ensure its installation directory is in your `PATH`. You can then build an installer using —

```
fbs installer
```

The Windows installer will be created at `target/<AppName>Setup.exe`.



The Windows NSIS installer

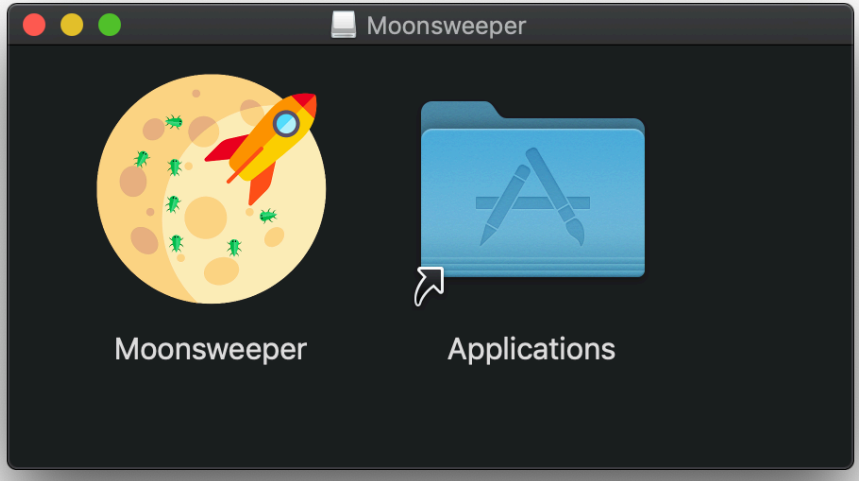
Download the *MoonsweeperSetup* .exe

Mac installer

There are no additional steps to create a MacOS installer. Just run the **fbs** command —

```
fbs installer
```

On Mac the command will generate a disk image at `target/<AppName>.dmg`. This disk image will contain the app bundle and a shortcut to the Applications folder. When your users open it they can drag the app to the Applications folder to install it.



The .dmg installer on Mac

Download the *Moonsweeper* .dmg bundle

Linux installer

To build installers on Linux you need to install the Ruby tool Effing package management! — use the installation guide to get it set up. Once that is in place you can use the standard command to create the Linux package file.

```
fbs installer
```

The resulting package will be created under the `target/` folder. Depending on your platform the package file will be named `<AppName>.deb`, `<AppName>.pkg.tar.xz` or `<AppName>.rpm`. Your users can install this file with their package manager.

Download the *Moonsweeper* .deb file

Find out more about fbs

More information about how the **fbs** packaging system works can be found in the manual which also introduces more advanced features should as distributing releases of Linux apps, reporting errors to the Sentry error logging platform and adding license keys to your software.

What's next?

If you've made it here you should be well on your way to creating your own apps! But there is still a lot to discover.

If you received a copy of this book from someone else (totally fine!) you might be interested in the accompanying website, containing regular new tutorials, video courses and demo applications. Like this book all samples are MIT licensed and free to mix into your own app.

Finally, this book is licensed CC-BY-NC-SA. This means you should feel free to give a copy of it to someone you know (or don't know). Help share the knowledge and fun of creating desktop apps with Python.

Thanks for reading, and if you have any feedback or suggestions let me know!

The video course

Thankyou for purchasing *Create Simple GUI Applications!*

If you enjoyed this book, you might also enjoy the accompanying video course, which you can purchase at https://www.learnpyqt.com/purchase

If you bought this book remember to claim your account on https://www.learnpyqt.com to get the upgrade price.

For latest tutorials, tips and code samples see https://www.learnpyqt.com/courses/

Resources

This section is a short list of resources that you may find useful in writing your Qt applications. They include both sources of good documentation and help, and also useful resources for making your applications look and function well.

Tutorials

For up to date tutorials, tips and code samples, check out the associated tutorial site for this book at https://www.learnpyqt.com

Documentation

Resource	Description
Qt5.5 Documentation	
Qt 4.8 Documentation	
PyQt4 Library documentation	
PyQt5 Library documentation	
PySide Library documentation	

Icon sets

The following icon sets are free to use, with the appropriate license, to give you applications a more professional and consistent look. The Fugue set are the icons suggested and used in the examples in this book but the others are also worth a look.

Resource	Description	License
Fugue by p.yusukekamiyamane	3,570 16x16 icons in PNG format	CC BY 3.0
Diagona by p.yusukekamiyamane	400 16x16 and 10x10 icons in PNG format	CC BY 3.0

Resource	Description	License
Tango Icons by The Tango Desktop Project	Icons using the Tango project colour theme.	Public domain

Source code

The complete source code all examples in this book is available to download from here.

Copyright