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 2019-05-26

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.

GPL Only

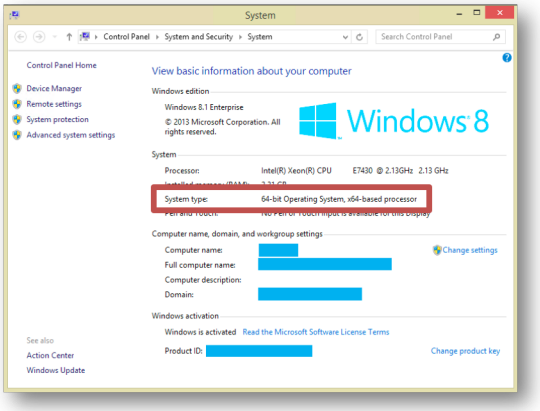
Note that the following instructions are **only** for installation of the GPL licensed version of PyQt. If you need to use PyQt in a non-GPL project you will need to purchase an alternative license from Riverbank Computing in order to release your software.

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.

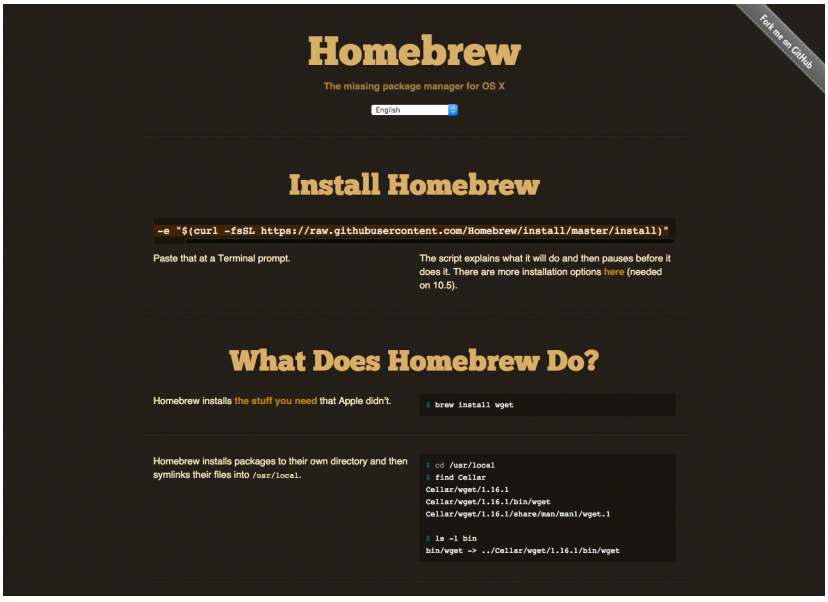
Simply download the above `.rar` file and unpack it with 7zip (or other unzip application). You can then copy the resulting folder to your Python site-packages folder — usually in `C:\Python27\lib\site-packages\`

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.

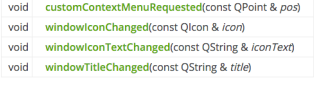
Load up a fresh copy of `MyApp_window.py` and save it under a new name for this section.

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)
```

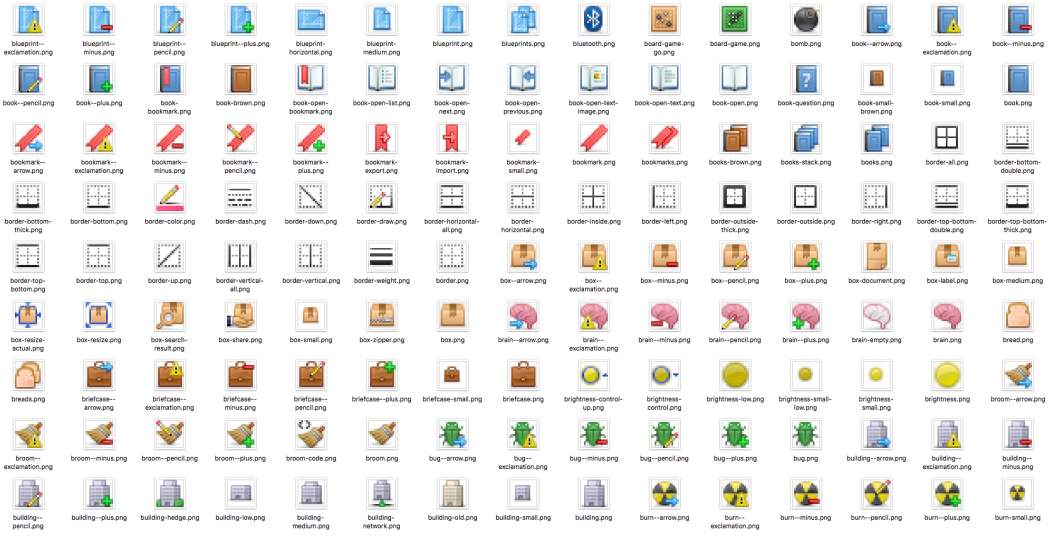
Run it!

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")
        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!

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)
```

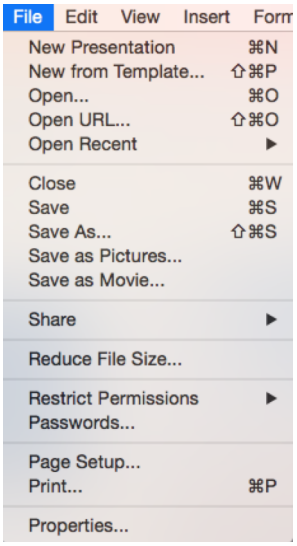
```
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        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)
```

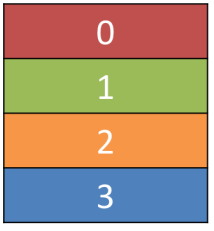
Run it!

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.

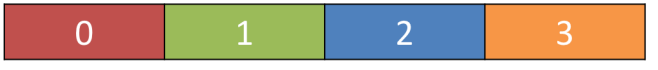
If you add a few more coloured widgets to the layout you'll notice that they line themselves up vertical in the order they are added.

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)
```

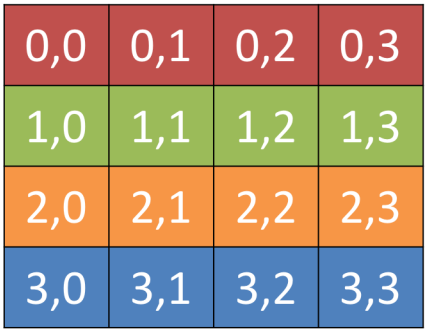
The following code shows the combination of nested widgets and layout margins and spacing. Experiment with the numbers til you get a feel for them.

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

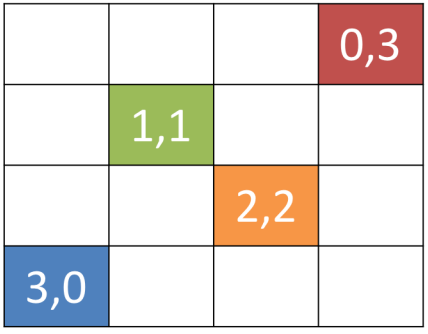
As useful as they are, if you try and using `QVBoxLayout` and `QHBoxLayout` for laying out multiple elements, e.g. for a form, you'll find it very difficult to ensure differently sized widgets line up. The solution to this is `QGridLayout`.



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`:

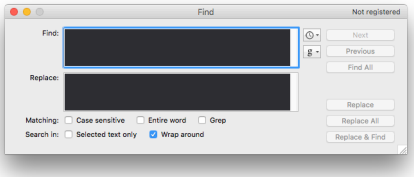
QTabWidget

```
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)
```

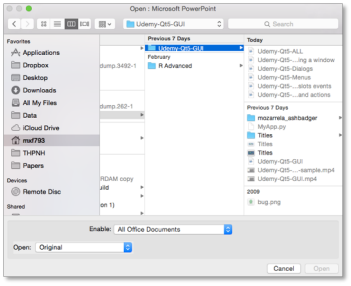
As you can see, it's a little more straightforward — and a bit more attractive! You can set the position of the tabs using the cardinal directions, toggle whether tabs are moveable with `.setMoveable` and turn a 'document mode' on and off which (on OS X) shows a slimmer tab interface. We encounter more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

The complete book

Thankyou for downloading this sample of *Create Simple GUI Applications*

If you like what you see you can purchase the complete book, together with an optional video course, at: https://www.learnpyqt.com/purchase

Use the code **SAMPLE50X4JK** to get a 50% discount on any book or course.

The full book covers many more aspects of developing with PyQt, from getting started with the Qt Creator to multithreading advanced applications. All purchases come with free updates as the book is developed and expands.

For latest tutorials, tips and code samples see https://www.learnpyqt.com/courses/