Create GUI Applications *with* Python & Qt5

The **hands-on** guide to making apps with Python

PyQt5 Edition

Martin Fitzpatrick

Create GUI Applications with Python & Qt5

The hands-on guide to making apps with Python

Martin Fitzpatrick

Version 4.0, 2020-07-01

SAMPLE

This is a sample of **Create GUI Applications with Python & Qt5** and includes a selection of pages from the book. The full table of contents is provided on the next page, so you can see what is in the full version.

You can buy the complete book from

https://www.learnpyqt.com/pyqt5-book/
(PyQt5 Edition)

https://www.learnpyqt.com/pyside2-book/
(PySide2 Edition)

If you have any questions please feel free to get in in touch with me at
martin@learnpyqt.com

Martin Fitzpatrick

Table of Contents

Listing 1. basic/creating_a_window_1.py

```
from PyQt5.QtWidgets import QApplication, QWidget

# 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)

# Create a Qt widget, which will be our window.
window = QWidget()
window.show()  # IMPORTANT!!!!! Windows are hidden by default.

# Start the event loop.
app.exec_()


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

First, launch your application. You can run it from the command line like any other Python script, for example—

```
python MyApp.py
```

Or, for Python 3—

```
python3 MyApp.py
```

From now on, you'll see the following box as a hint to run your application and test it out, along with an indication of what you'll see.

🚀 **Run it!** You will now see your window. Qt automatically creates a window with the normal window decorations and you can drag it around and resize it like any window.

What you'll see will depend on what platform you're running this example on. The image below shows the window as displayed on Windows, macOS and Linux (Ubuntu).

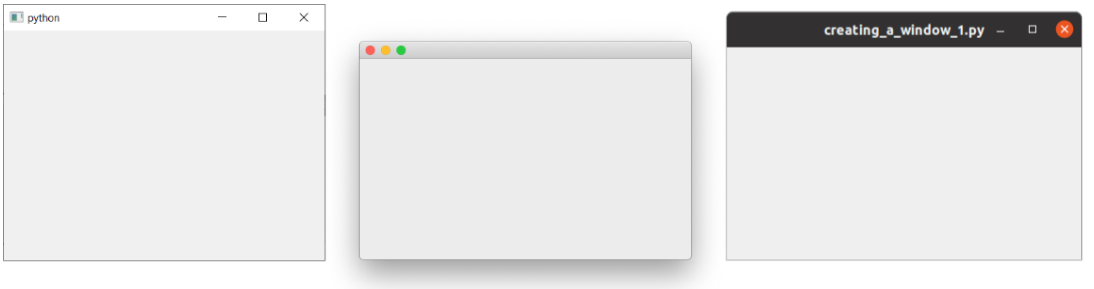


Figure 3. Our window, as seen on Windows, macOS and Linux.

Stepping through the code

Let's step through the code line by line, so we understand exactly what is happening.

First, we import the PyQt5 classes that we need for the application. Here we're importing `QApplication`, the application handler and `QWidget`, a basic *empty* GUI widget, both from the `QtWidgets` module.

```
from PyQt5.QtWidgets import QApplication, QWidget
```

The main modules for Qt are `QtWidgets`, `QtGui` and `QtCore`.

You could do `from <module> import *` but this kind of global import is generally frowned upon in Python, so we'll avoid it here.

Next we create an instance of `QApplication`, passing in `sys.arg`, which is

What's the event loop?

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 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.

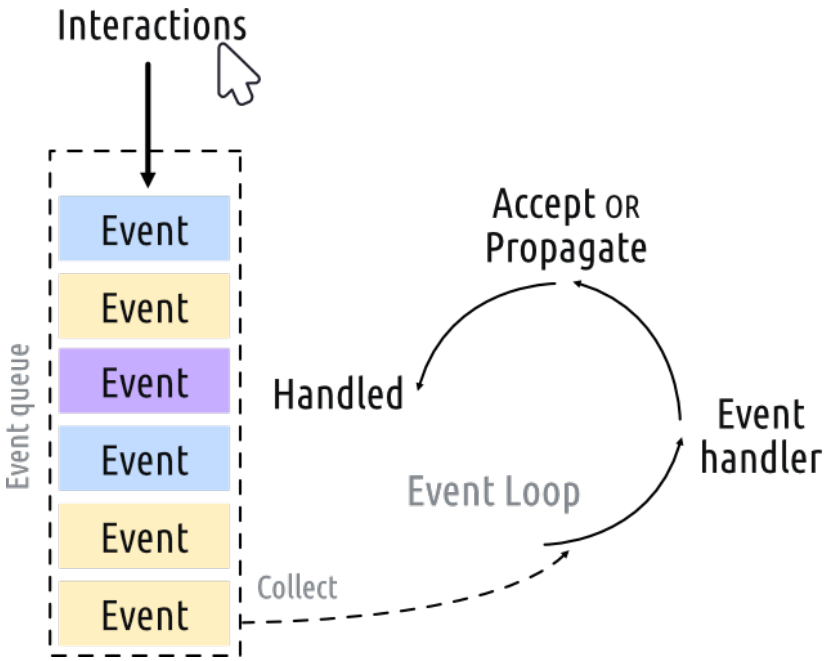


Figure 4. The event loop in Qt.

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.

The `QApplication` class

- `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 at any time

The underscore is there because `exec` was a reserved word in Python 2.7. PyQt5 handles this by appending an underscore to the name used in the C++ library. You'll also see `.print_()` methods on widgets for example.

QMainWindow

As we discovered in the last part, in Qt *any* widgets can be windows. For example, if you replace `QtWidget` with `QPushButton`. In the example below, you would get a window with a single push-able button in it.

Listing 2. basic/creating_a_window_2.py

```
from PyQt5.QtWidgets import QApplication, QPushButton
window = QPushButton("Push Me")
window.show()
```

This is neat, but not really very *useful*—it's rare that you need a UI that consists of only a single control! But, as we'll discover later, the ability to nest widgets within other widgets using *layouts* means you can construct complex UIs inside an empty `QWidget`.

But, Qt already has a solution for you—the `QMainWindow`. This is a pre-made widget which provides a lot of standard window features you'll make use of in your apps, including toolbars, menus, a statusbar, dockable widgets and more. We'll look at these advanced features later, but for now, we'll add a simple empty `QMainWindow` to our application.

Listing 3. basic/creating_a_window_3.py

```
from PyQt5.QtWidgets import QApplication, QMainWindow

import sys


app = QApplication(sys.argv)

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

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

🚀 **Run it!** You will now see your main window. It looks exactly the same as before!

So our `QMainWindow` isn't very interesting at the moment. We can fix that by adding some content. 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 behavior to be self contained. We can add our own subclass of `QMainWindow` — call it `MainWindow` to keep things simple.

Listing 4. basic/creating_a_window_4.py

```
import sys

from PyQt5.QtCore import QSize, Qt
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton  ①


# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()  ②

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")

        # Set the central widget of the Window.
        self.setCentralWidget(button)  ③


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec_()
```

① Common Qt widgets are always imported from the `QtWidgets` namespace.

② We must always call the `__init__` method of the `super()` class.

③ Use `.setCentralWidget` to place a widget in the `QMainWindow`.

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

In our `__init__` block we first use `.setWindowTitle()` to change the title of our main window. Then we add our first widget — a `QPushButton` — to the middle of the window. This is one of the basic widgets available in Qt. When creating the button you can pass in the text that you want the button to display.

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.

> 🚀 **Run it!** You will now see your window again, but this time with the `QPushButton` widget in the middle. Pressing the button will do nothing, we'll sort that next.

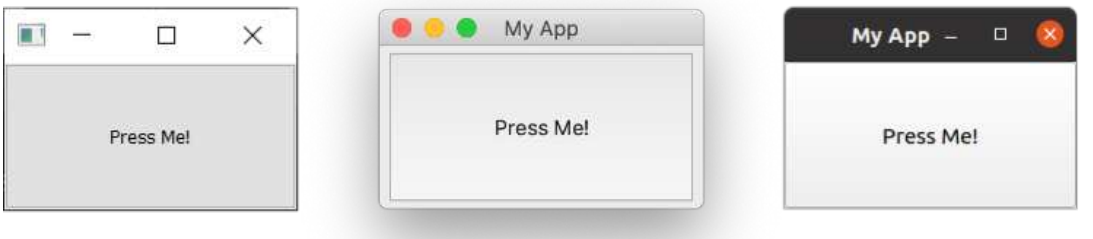


Figure 5. Our `QMainWindow` with a single `QPushButton` on Windows, macOS and Linux.

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!

Sizing windows and widgets

The window is currently freely resizable—if you grab any corner with your mouse you can drag and resize it to any size you want. While it's good to let your users resize your applications, sometimes you may want to place restrictions on minimum or maximum sizes, or lock a window to a fixed size.

In Qt sizes are defined using a `QSize` object. This accepts *width* and *height* parameters in that order. For example, the following will create a *fixed size*

window of 400x300 pixels.

Listing 5. basic/creating_a_window_end.py

```
import sys

from PyQt5.QtCore import QSize, Qt
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton


# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")

        self.setFixedSize(QSize(400, 300))  ①

        # Set the central widget of the Window.
        self.setCentralWidget(button)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec_()
```

① Setting the size on the window.

🚀 **Run it!** You will see a fixed size window—try and resize it, it won't work.

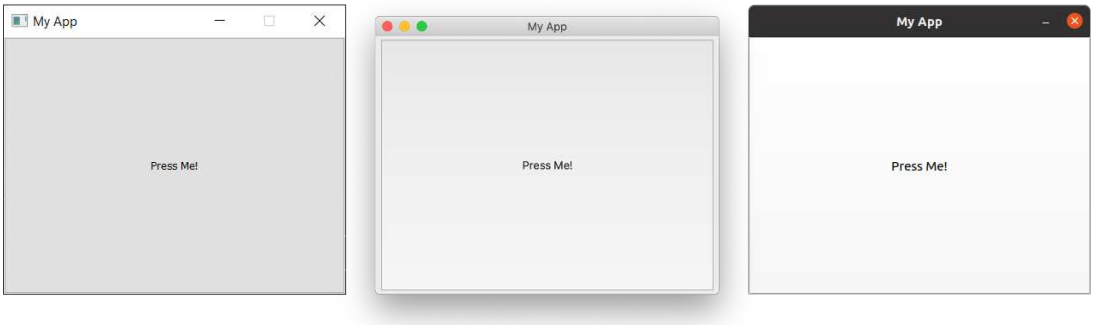


Figure 6. Our fixed-size window, notice that the maximize control is disabled on Windows & Linux. On macOS you can maximize the app to fill the screen, but the central widget will not resize.

As well as `.setFixedSize()` you can also call `.setMinimumSize()` and `.setMaximumSize()` to set the minimum and maximum sizes respectively. Experiment with this yourself!

You can use these size methods on *any* widget.

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.py` as we'll need it again later.

Qt Flags

Note that you use an **OR** pipe (`|`) to combine the two flags by convention. The flags are non-overlapping *bitmasks*. e.g. `Qt.AlignLeft` has the binary value `0b0001`, while `Qt.AlignBottom` is `0b0100`. By ORing together we get the value `0b0101` representing 'bottom left'.

We'll take a more detailed look at the `Qt` namespace and Qt flags later in Enums & the Qt Namespace.

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

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

Weirdly, you can also use `QLabel` to display an image using the `.setPixmap()` method. This accepts an *pixmap* (a pixel array), 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:

```
widget.setPixmap(QPixmap('otje.jpg'))
```

Figure 9. Otje. What a lovely face.

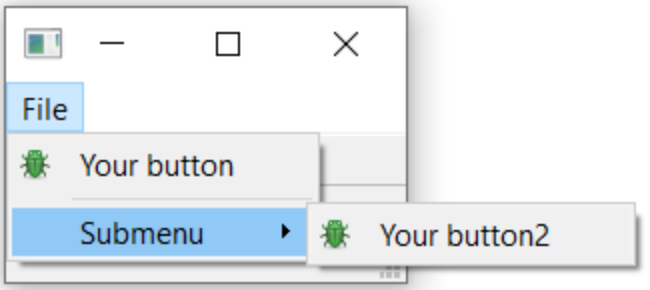


Figure 46. Submenu nested in the File menu.

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.

Listing 44. /toolbars_and_menus_end.py

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

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
```

Organising menus & toolbars

If your users can't find your application's actions, they can't use your app to it's full potential. Making actions discoverable is key to creating a user-friendly application. It is a common mistake to try and address this by adding actions *everywhere* and end up overwhelming and confusing your users.

Put common and necessary actions first, making sure they are easy to find and recall. Think of the **File › Save** in most editing applications. Quickly accessible at the top of the File menu and bound with a simple keyboard shortcut `Ctrl` + `S`. If **Save file...** was accessible through **File › Common operations › File operations › Active document › Save** or the shortcut `Ctrl` + `Alt` + `J` users would have a harder time finding it, a harder time using it, and be less **likely** to save their documents.

Arrange actions into logical groups. It is easier to find something among a small number of alternatives, than in a long list. It's even easier to find if it is among similar things.

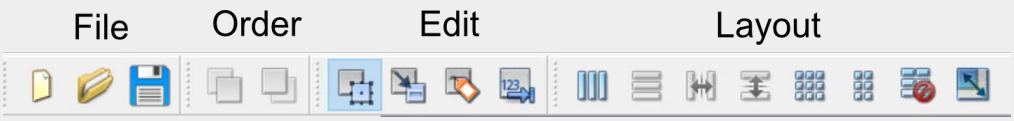


Figure 47. Grouped toolbars in Qt Designer.

Avoid replicating actions in multiple menus, as this introduces an ambiguity of "do these do the same thing?" even if they have an identical label. Lastly, don't be tempted to simplify menus by hiding/removing entries dynamically. This leads to confusion as users hunt for something that doesn't exist "...it was here a minute ago". Different states should be indicated by disabling menu items or separate windows and dialogs.

14. Getting started with Qt Designer

In this chapter we'll take a quick tour through using *Qt Designer* to design a UI and exporting that UI for use in your PyQt5 application. We'll only scratch the surface of what you can do with *Qt Designer* here, but once you've got the basics down, feel free to experiment in more detail.

Open up *Qt Designer* 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.

Qt Designer

Qt Designer starts up with the *New Form* dialog. Here you can choose the type of interface you're building—this decides the base widget you will build your interface on. If you are starting an application then *Main Window* is usually the right choice. However, you can also create `.ui` files for dialog boxes and custom compound widgets.

Form is the technical name given to a UI layout, since many UIs resemble a paper form with various input boxes.

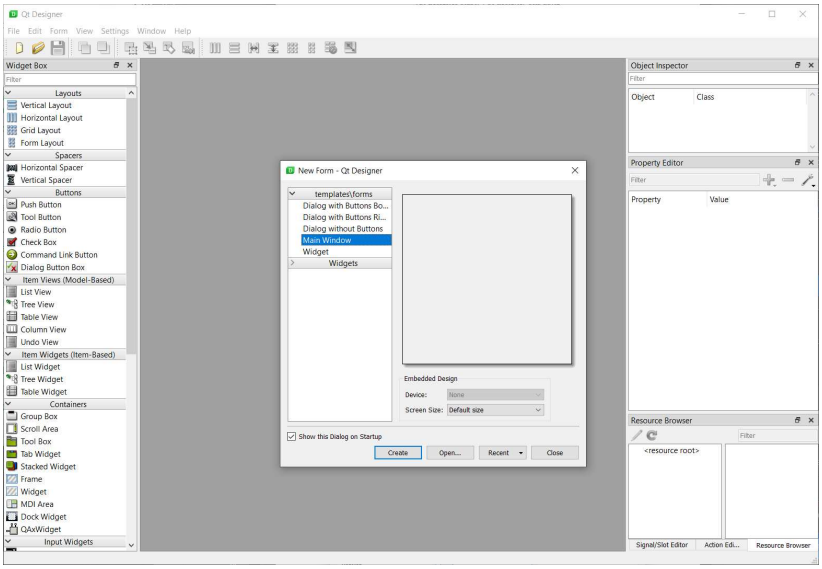


Figure 69. The Qt Designer interface

Aesthetics

If you're not a designer, it can be hard to create *beautiful* interfaces, or even know what they are. Thankfully there are simple rules you can follow to create interfaces that, if not *beautiful* at least won't be *ugly*. The key concepts are—alignment, groups and space.

Alignment is about reducing visual noise. Think of the corners of widgets as *alignment points* and aim to minimize the number of *unique* alignment points in the UI. In practice, this means making sure the edges of elements in the interface *line up* with one another.

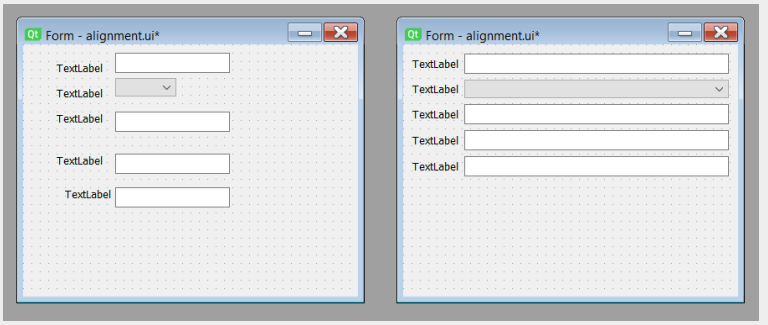


Figure 84. The effect of alignment of interface clarity.

If you have differently sized inputs, align the edge you read *from*. English is a left-to-right language, so if your app is in English align the left.

Groups of related widgets gain context making them easier to *understand*. Structure your interface so related things are found together.

Space is key to creating visually distinct regions in your interface—without space between groups, there are no groups! Keep spacing consistent and meaningful.

As before, once the palette is constructed it must be applied to take effect. Here we apply it to the application as a whole by calling `app.setPalette()`. All widgets will adopt the theme once applied. You can use this skeleton to construct your own applications using this theme.

In the code examples with this book you can also find `themes/palette_dark_widgets.py` which reproduces the widgets demo, using this palette. The result on each platform is shown below.

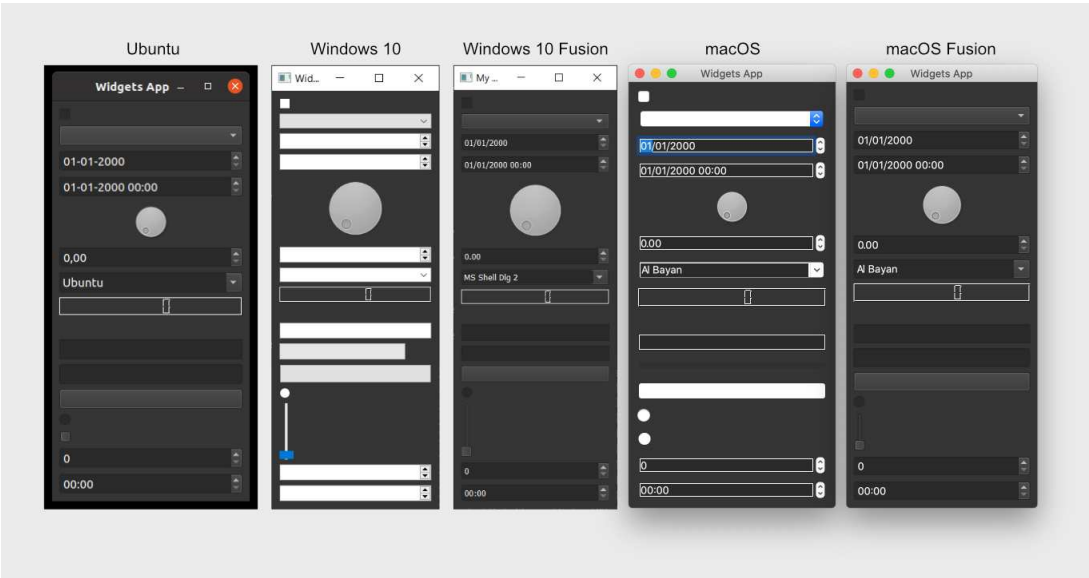


Figure 95. Custom dark palette on different platforms and themes

You'll notice that when using the default Windows and macOS themes some widgets do not have their colors applied correctly. This is because these themes make use of platform-native controls to give a true native feel. If you want to use a dark or heavily customized theme on Windows 10, it is recommended to use the *Fusion* style on these platforms.

Dark Mode

Dark mode is becoming popular as people spend more and more time on screens. Darker themed OS and applications help to minimize eye strain and reduce sleep distribution if working in the evening.

Windows, macOS and Linux all provide support for dark mode themes, and

```
QLineEdit {
    border-width: 7px;
    border-style: dashed;
    border-color: red;
}
```

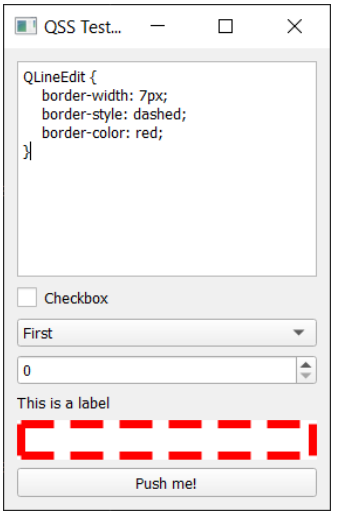


Figure 104. Applying dashed red border to QLineEdit

Next we'll look in some detail at how these QSS rules are styling the widgets, gradually building up to some more complex rule sets.

A full list of styleable widgets is available in the Qt documentation.

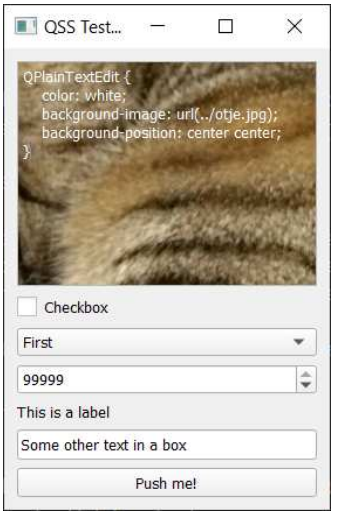


Figure 110. Centered background image.

To align the bottom-right of the image to the bottom-right of the *origin rectangle* of the widget, you would use.

```
QPlainTextEdit {
    color: white;
    background-image: url(../otje.jpg);
    background-position: bottom right;
}
```

The *origin rectangle* can be modified using the `background-origin` property. This accepts one of the values `margin`, `border`, `padding` or `content` which defines that specific box as the reference of background position alignment.

To understand what this means we'll need to take a look at the widget box model.

The widget Box Model

The term *box model* describes the relationships between the *boxes* (rectangles) which surround each widget and the effect these boxes have on the size or layout of widgets in relationship to one another. Each Qt widget is surrounded by four concentric boxes—from inside out, these are content, padding, border and margin.

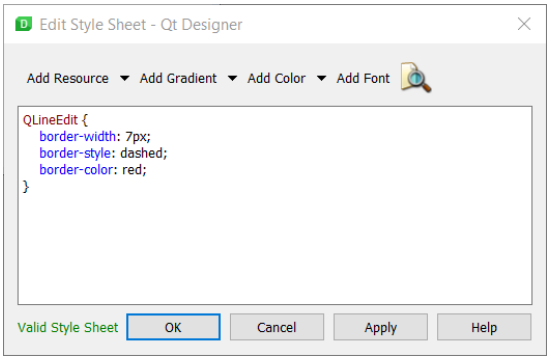


Figure 135. The QSS editor in Qt Designer.

As well as entering rules as text, the QSS editor in Qt Designer gives you access to a resource lookup tool, color selection widget and a gradient designer. This tool (shown below) provides a number of built-in gradients you can add to your rules, but you can also define your own custom gradients if you prefer.

Figure 136. The QSS gradient designer in Qt Designer.

Gradients are defined using QSS rules so you can copy and paste them elsewhere (including into your code) to re-use them if you like.

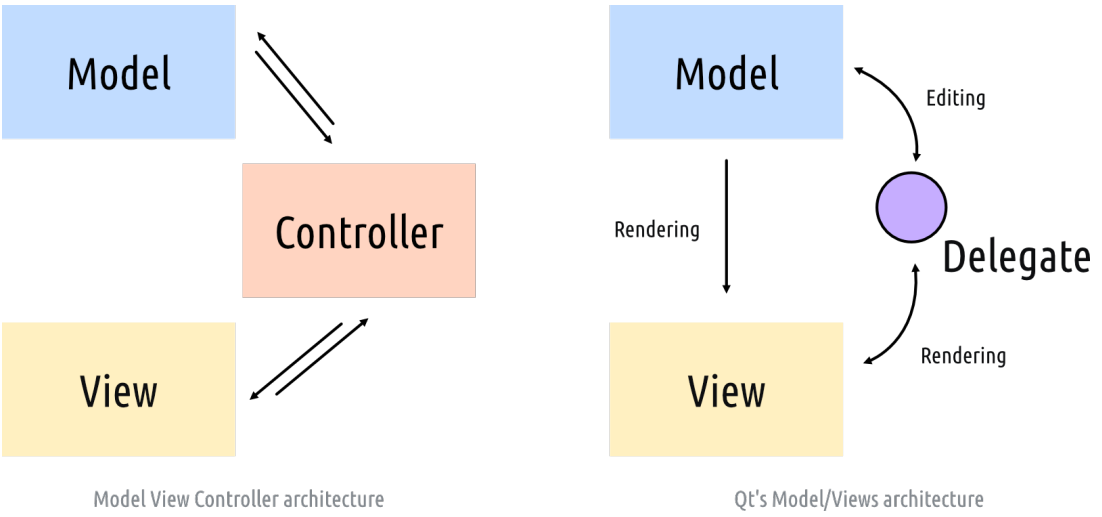


Figure 138. Comparing the MVC model and the Qt Model/View architecture.

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 ViewController. The Model holds the data (or a reference to it) and presents this data through a standardized 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 Qt itself, or 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.

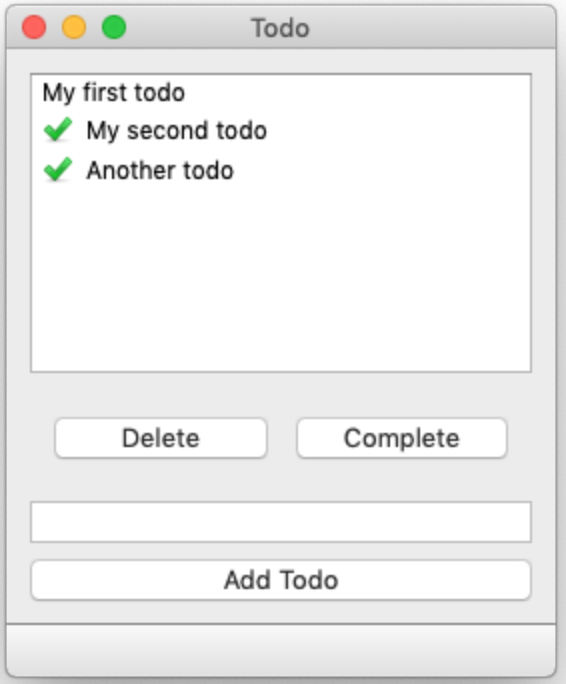


Figure 142. Todos complete

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 any 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 data(self, index, role):
    if role == Qt.DisplayRole:
        # Get the raw value
        value = self._data[index.row()][index.column()]

        # Perform per-type checks and render accordingly.
        if isinstance(value, datetime):
            # Render time to YYY-MM-DD.
            return value.strftime("%Y-%m-%d")

        if isinstance(value, float):
            # Render float to 2 dp
            return "%.2f" % value

        if isinstance(value, str):
            # Render strings with quotes
            return '"%s"' % value

        # Default (anything not captured above: e.g. int)
        return value
```

Use this together with the modified sample data below to see it in action.

```
data = [
    [4, 9, 2],
    [1, -1, 'hello'],
    [3.023, 5, -5],
    [3, 3, datetime(2017,10,1)],
    [7.555, 8, 9],
]
```

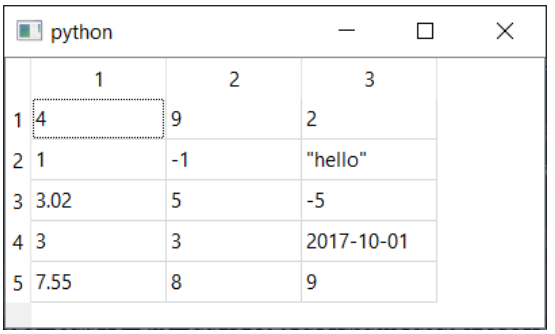


Figure 144. Custom data formatting

So far we've only looked at how we can customize how the data itself is formatted. However, the model interface gives you far more control over the display of table cells including colors and icons. In the next part we'll look at how to use the model to customize `QTableView` appearance.

Styles & Colors with Roles

Using colors and icons to highlight cells in data tables can help make data easier to find and understand, or help users to select or mark data of interest. Qt allows for complete control of all of these from the model, by responding to the relevant *role* on the `data` method.

The types expected to be returned in response to the various `role` types are shown below.

Role	Type
`Qt.BackgroundRole`	`QBrush` (also `QColor`)
`Qt.CheckStateRole`	`Qt.CheckState`
`Qt.DecorationRole`	`QIcon`, `QPixmap`, `QColor`
`Qt.DisplayRole`	`QString` (also `int`, `float`, `bool`)
`Qt.FontRole`	`QFont`
`Qt.SizeHintRole`	`QSize`
`Qt.TextAlignmentRole`	`Qt.Alignment`
`Qt.ForegroundRole`	`QBrush` (also `QColor`)

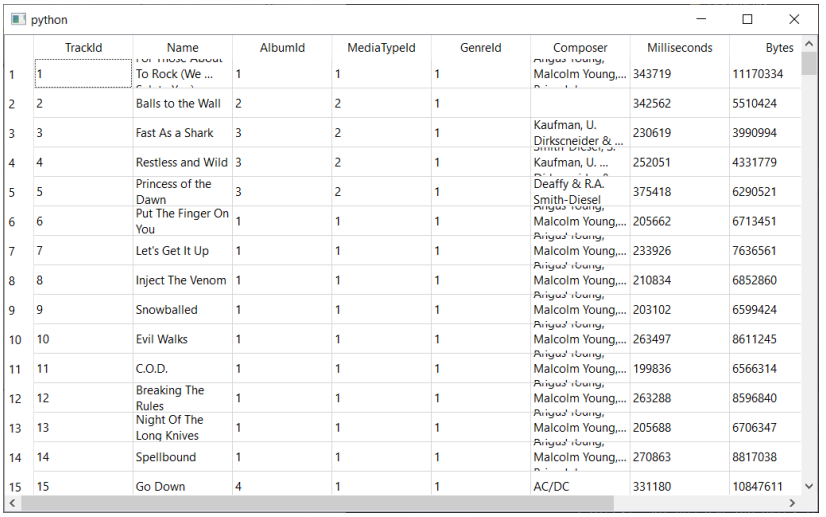


Figure 153. The tracks table displayed in a `QTableView`.

You can resize the columns by dragging the right hand edge. Resize to fit the contents by double-clicking on the right hand edge.

Editing the data

Database data displayed in a `QTableView` is editable by default—just double-click on any cell and you will be able to modify the contents. The changes are persisted back to the database immediately after you finish editing.

Qt provides some control over this editing behavior, which you may want to change depending on the type of app you are building. Qt terms these behaviors *editing strategy* and they can be one of the following -

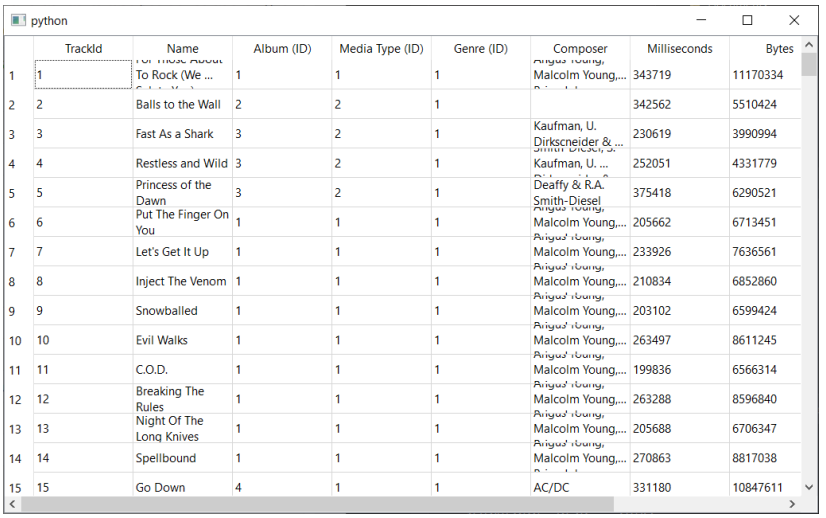
Strategy	Description
`QSqlTableModel.OnFieldChange`	Changes are applied automatically, when the user deselects the edited cell.
`QSqlTableModel.OnRowChange`	Changes are applied automatically, when the user selects a different row.

Column titles

By default the column header titles on the table come from the column names in the database. Often this isn't very user-friendly, so you can replace them with proper titles using `.setHeaderData`, passing in the column index, the direction — horizontal (top) or vertical (left) header — and the label.

Listing 88. database/tableview_tablemodel_titles.py

```
        self.model.setTable("Track")
        self.model.setHeaderData(1, Qt.Horizontal, "Name")
        self.model.setHeaderData(2, Qt.Horizontal, "Album (ID)")
        self.model.setHeaderData(3, Qt.Horizontal, "Media Type (ID)")
        self.model.setHeaderData(4, Qt.Horizontal, "Genre (ID)")
        self.model.setHeaderData(5, Qt.Horizontal, "Composer")
        self.model.select()
```



	TrackId	Name	Album (ID)	Media Type (ID)	Genre (ID)	Composer	Milliseconds	Bytes
1	1	To Rock (We ...	1	1	1	Malcolm Young,...	343719	11170334
2	2	Balls to the Wall	2	2	1		342562	5510424
3	3	Fast As a Shark	3	2	1	Kaufman, U. Dirkscneider & ...	230619	3990994
4	4	Restless and Wild	3	2	1	Kaufman, U. ...	252051	4331779
5	5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Diesel	375418	6290521
6	6	Put The Finger On You	1	1	1	Malcolm Young,...	205662	6713451
7	7	Let's Get It Up	1	1	1	Malcolm Young,...	233926	7636561
8	8	Inject The Venom	1	1	1	Malcolm Young,...	210834	6852860
9	9	Snowballed	1	1	1	Malcolm Young,...	203102	6599424
10	10	Evil Walks	1	1	1	Malcolm Young,...	263497	8611245
11	11	C.O.D.	1	1	1	Malcolm Young,...	199836	6566314
12	12	Breaking The Rules	1	1	1	Malcolm Young,...	263288	8596840
13	13	Night Of The Long Knives	1	1	1	Malcolm Young,...	205688	6706347
14	14	Spellbound	1	1	1	Malcolm Young,...	270863	8817038
15	15	Go Down	4	1	1	AC/DC	331180	10847611

Figure 156. The tracks table with nicer column titles.

As when sorting, it is not always convenient to use the column indexes for this — if the column order changes on the database, the names set in your application will be out of sync.

As before, we can use `.fieldIndex()` to lookup the index for a given name. You can go a step further and define a Python `dict` of column name and title to apply in one go, when setting up the model.

Listing 98. databases/widget_mapper.py

```
        prev_rec = QPushButton("Previous")
        prev_rec.clicked.connect(self.mapper.toPrevious)

        next_rec = QPushButton("Next")
        next_rec.clicked.connect(self.mapper.toNext)

        save_rec = QPushButton("Save Changes")
        save_rec.clicked.connect(self.mapper.submit)
```

Now you can browse between records in the *Tracks* table, make changes to the track data and submit these changes to the database.

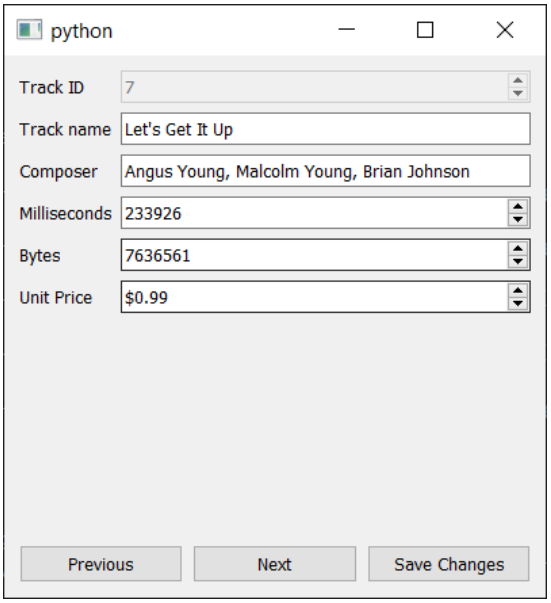


Figure 164. Viewing records, with previous/next controls and save to submit.

Authenticating with `QSqlDatabase`

In the examples so far we've used SQLite database files. But often you'll want to connect to a remote SQL server instead. That requires a few additional parameters, including the hostname (where the database is located) and a username and password if appropriate.

Listing 106. further/routing_2.py

```
    def mousePressEvent(self, e):
        route = {  ①
            Qt.LeftButton: self.left_mousePressEvent,
            Qt.MiddleButton: self.middle_mousePressEvent,
            Qt.RightButton: self.right_mousePressEvent,
        }
        button = e.button()
        fn = route[button]  ②
        return fn(e)  ③

    def left_mousePressEvent(self, e):
        self.label.setText("mousePressEvent LEFT")
        if e.x() < 100:
            self.status.showMessage("Left click on left")
            self.move(self.x() - 10, self.y())
        else:
            self.status.showMessage("Left click on right")
            self.move(self.x() + 10, self.y())

    def middle_mousePressEvent(self, e):
        self.label.setText("mousePressEvent MIDDLE")

    def right_mousePressEvent(self, e):
        if e.x() < 100:
            self.status.showMessage("Right click on left")
            print("Something else here.")
            self.move(10, 10)
        else:
            self.status.showMessage("Right click on right")
            self.move(400, 400)
```

① Define the routing dictionary, keyed by the event type and with handler methods as the values.

② Get the route method from the dictionary.

③ Call the method, passing the event argument.

The value of `e.button()` is used to perform a lookup in our *routing* dictionary. This returns a single method, which we can call to handle that particular

The default behavior in Qt is to close an application once all the active windows have closed. This won't affect this toy example, but will be an issue in application where you do create windows and then close them. Setting `app.setQuitOnLastWindowClosed(False)` stops this and will ensure your application keeps running.

The provided icon shows up in the toolbar (you can see it on the left hand side of the icons grouped on the right of the system tray or menubar).

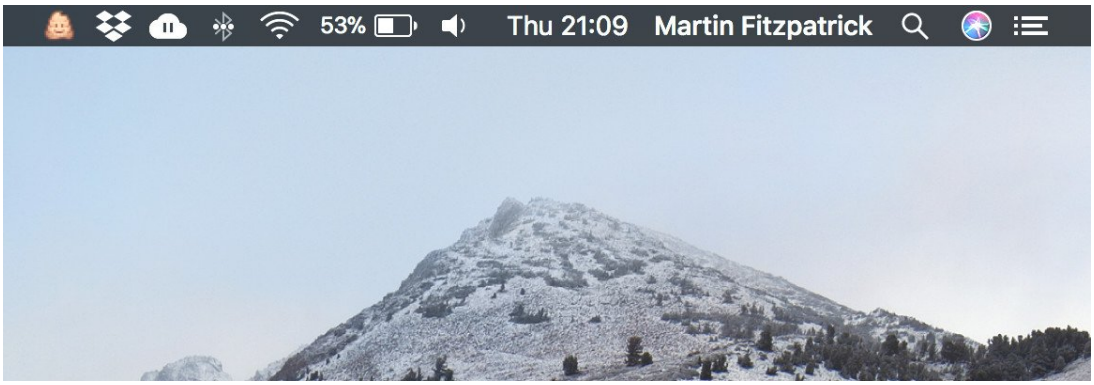


Figure 168. The icon showing on the menubar.

Clicking (or right-clicking on Windows) on the icon shows the added menu.

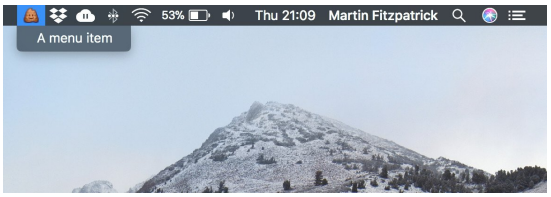


Figure 169. The menubar app menu.

This application doesn't do anything yet, so in the next part we'll expand this example to create a mini color-picker.

Below is a more complete working example using the built in `QColorDialog` from Qt to give a toolbar accessible color picker. The menu lets you choose to get the picked color as HTML-format `#RRGGBB`, `rgb(R,G,B)` or `hsv(H,S,V)`.

```
from PyQt5.QtWidgets import QApplication, QSystemTrayIcon,
```

```
>>> align = Qt.AlignLeft
>>> align == Qt.AlignLeft
True
```

For combined flags we can also check equality with the combination of flags—

```
>>> align = Qt.AlignLeft | Qt.AlignTop
>>> align == Qt.AlignLeft | Qt.AlignTop
True
```

But sometimes, you want to know if a given variable *contains* a specific flag. For example, perhaps we want to know if `align` has the *align left* flag set, regardless of any other alignment state.

How can we check that an element has `Qt.AlignLeft` applied, once it's been combined with another? In this case a `==` comparison will not work, since they are not numerically equal.

```
>> alignment = Qt.AlignLeft | Qt.AlignTop
>> alignment == Qt.AlignLeft  # 33 == 1
False
```

We need a way to compare the `Qt.AlignLeft` flag against the bits of our compound flag. For this we can use a *bitwise AND*.

Bitwise AND (&) checks

In Python, *bitwise AND* operations are performed using the `&` operator.

In the previous step we combined together `Qt.AlignLeft` (1) and `Qt.AlignTop` (32) to produce "Top Left" (33). Now we want to check if the resulting combined flag has the align left flag set. To test we need to use *bitwise AND* which checks bit by bit to see if both input values are 1, returning a 1 in that place if it is true.

Listing 117. bitmap/rect.py

```
    def draw_something(self):
        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

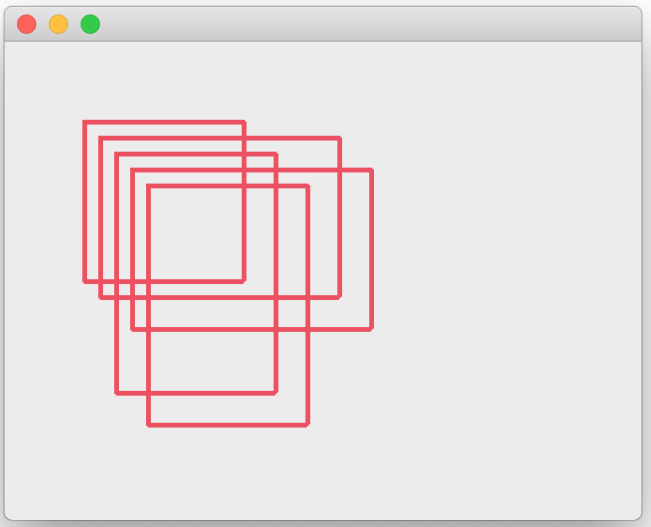


Figure 182. Drawing rectangles.

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.

```
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 expect.

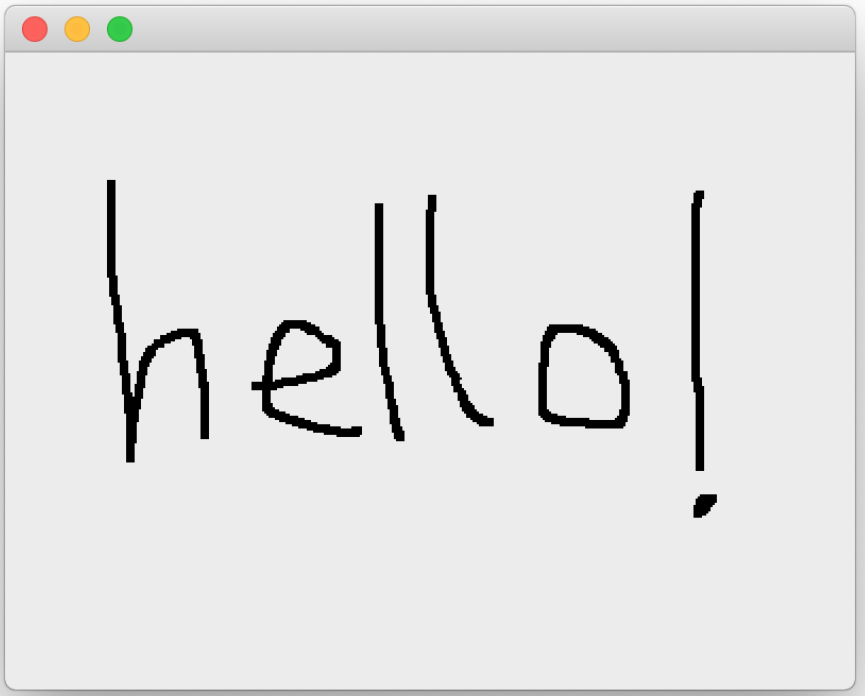


Figure 190. Drawing with the mouse, using a continuous line.

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

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` . When we

30. 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 customizable PowerBar meter with a dial control.

Figure 193. 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

You can now experiment with passing in different values for the `__init__` to `PowerBar`, e.g. increasing the number of bars, or providing a color list. Some examples are shown below.

A good source of hex color palettes is the Bokeh source.

```
PowerBar(10)
PowerBar(3)
PowerBar(["#5e4fa2", "#3288bd", "#66c2a5", "#abdda4", "#e6f598",
"#ffffbf", "#fee08b", "#fdae61", "#f46d43", "#d53e4f", "#9e0142"])
PowerBar(["#a63603", "#e6550d", "#fd8d3c", "#fdae6b", "#fdd0a2",
"#feedde"])
```

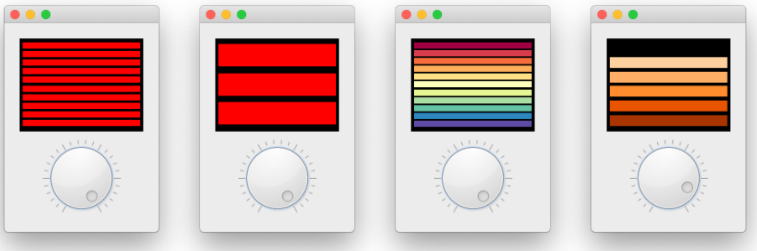


Figure 202. Some PowerBar examples.

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.

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

Familiarity & Skeuomorphism

One of the most powerful tools you can exploit when building user interfaces is *familiarity*. That is, giving your users the sense that your interface is something they have used before.

Good interfaces are often described as being *intuitive*. There is nothing naturally intuitive about moving a mouse pointer around a screen and clicking on square-ish bumps. But, after spending years of our lives doing exactly that, there is something very familiar about it. There is nothing more guaranteed to make an application incredibly user unfriendly than to ignore the value of this experience.

The search for *familiarity* in user interfaces led to the use of *skeuomorphism* in GUI design. Skeuomorphism is the application of non-functional design cues from other objects, where those design elements are functional. That can mean using common interface elements, or replicating some aspects of the manual process you're replacing. In the context of GUIs this often means user interfaces that look like real objects.

Figure 204. RealPhone—One of IBM's RealThings™

Apple was a big proponent of skeuomorphism during the Steve Jobs era. In recent years GUIs have, inspired by the web, moved increasingly to "flat" designs. Yet, modern user-interfaces all still have elements of

② Progress 0-100% as an integer.

③ When the job finishes, we need to cleanup (delete) the workers progress.

④ Delete the progress for the finished worker.

If you run this, you'll see the global progress bar along with an indicator to show how many active workers there are running.

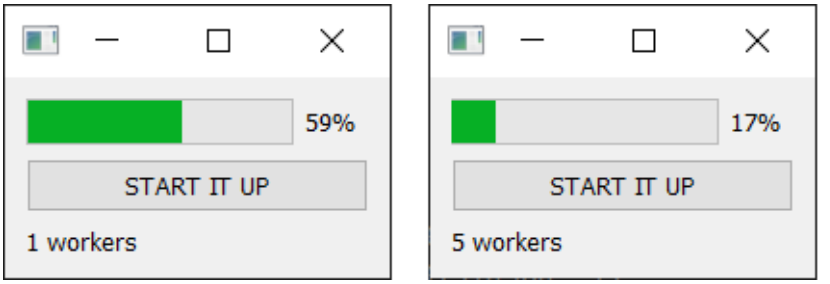


Figure 210. The window showing the global progress state, together with the number of active workers.

Checking the console output for the script you can see the actual status for each of the individual workers.

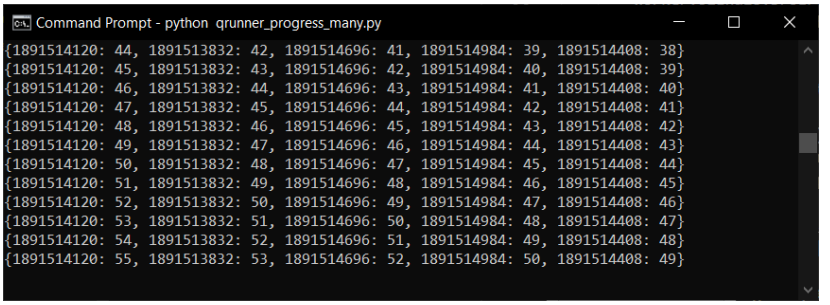


Figure 211. Check the shell output to see the individual worker progress.

Removing the worker immediately means that the progress will jump *backwards* when a job finishes—removing 100 from the average calculation will cause the average to fall. You can postpone the cleanup if you like, for example the following will only remove the entries when *all* progress bars reach 100.

Listing 157. concurrent/qrunner_progress_many_2.py

```
import random
```

Listing 177. concurrent/qrunner_manager_stop.py

```
    def stop_worker(self):
        selected = self.progress.selectedIndexes()
        for idx in selected:
            job_id, _ = self.workers.data(idx, Qt.DisplayRole)
            self.workers.kill(job_id)
```

In addition to this we need to modify the delegate to draw the currently selected item and update the worker and manager to pass through the *kill* signal. Take a look at the full source for this example to see how it all fits together.

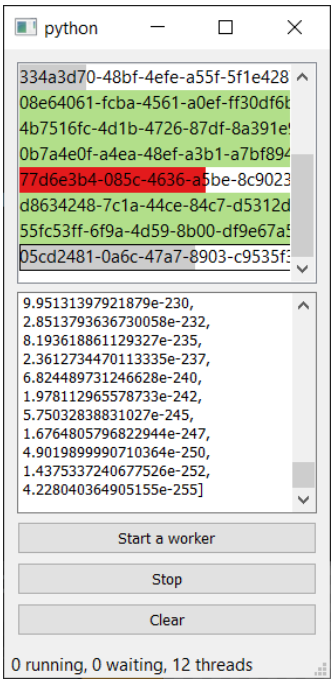


Figure 218. The manager, selecting a job allows you to stop it.

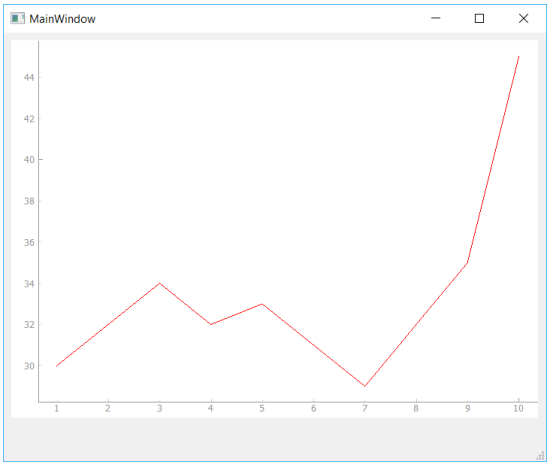


Figure 222. Changing Line Color.

By changing the `QPen` object we can change the appearance of the line, including both line width in pixels and style (dashed, dotted, etc.) using standard Qt line styles. For example, the following example creates a 15px width dashed line in red.

```
pen = pg.mkPen(color=(255, 0, 0), width=15, style=QtCore.Qt.DashLine)
```

The result is shown below, giving a 15px dashed red line.

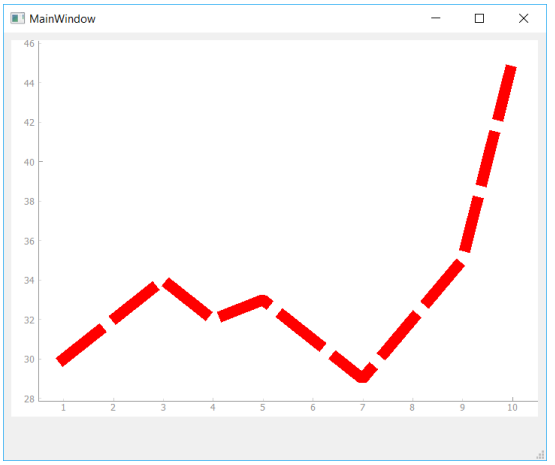


Figure 223. Changing Line Width and Style.

The standard Qt line styles can all be used, including `Qt.SolidLine`, `Qt.DashLine`, `Qt.DotLine`, `Qt.DashDotLine` and `Qt.DashDotDotLine`. Examples of each of these lines are shown in the image below, and you can read more in the Qt

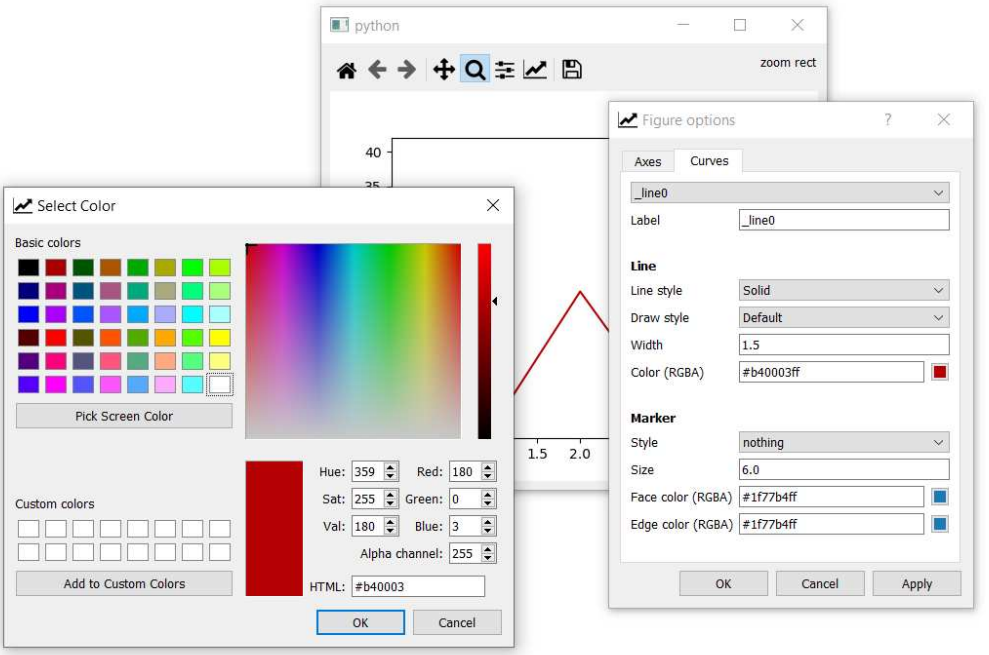


Figure 235. Matplotlib curve options.

For more information on navigating and configuring Matplotlib plots, take a look at the official Matplotlib toolbar documentation.

Updating plots

Quite often in applications you'll want to update the data shown in plots, whether in response to input from the user or updated data from an API. There are two ways to update plots in Matplotlib, either

1. clearing and redrawing the canvas (simpler, but slower) or,
2. by keeping a reference to the plotted line and updating the data.

If performance is important to your app it is recommended you do the latter, but the first is simpler. We start with the simple clear-and-redraw method first below —

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`.

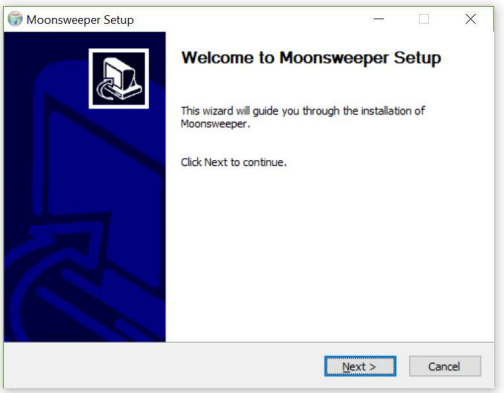


Figure 239. The Windows NSIS installer.

Download the *MoonsweeperSetup* .exe

macOS installer

There are no additional steps to create a macOS installer. Just run the *fbs* command —

```
fbs installer
```

39. 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 tricounter will tell you the number of B'ugs in the vicinity.

Suggested reading

This application makes use of features covered in Signals & Slots, and Events.

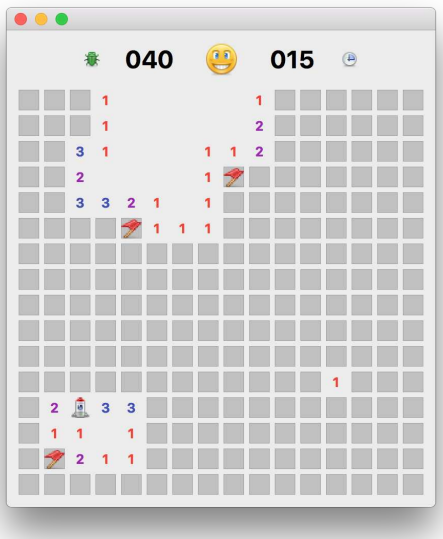


Figure 243. Moonsweeper.

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.

Appendix B: Translating C++ Examples to Python

When writing applications with PyQt5 we are really writing applications with Qt.

PyQt5 acts as a wrapper around the Qt libraries, translating Python method calls to C++, handling type conversions and transparently creating Python objects to represent Qt objects in your applications. The result of all this cleverness is that you can use Qt from Python while writing *mostly* Pythonic code — if we ignore the *camelCase*.

While there is a lot of PyQt5 example code out there, there are far more Qt C++ examples. The core documentation is written for C++. The library is written in C++. This means that sometimes, when you're looking how to do something, the only resource you'll find is a C++ tutorial or some C++ code.

Can you use it? Yes! If you have no experience with C++ (or C-like languages) then the code can look like gibberish. But before you were familiar with Python, Python probably looked a bit like gibberish too. You don't need to be able to *write* C++ to be able to *read* it. Understanding and decoding is easier than writing.

With a little bit of effort you'll be able to take any C++ example code and translate it into fully-functional Python & PyQt5. In this chapter we'll take a snippet of Qt5 code and step-by-step convert it into fully-working Python code.

The example code

We'll start with the following example block of code creating a simple window with a `QPushButton` and a `QLineEdit`. Pressing on the button will clear the line edit. Pretty exciting stuff, but this includes a few key parts of translating Qt examples to PyQt5 — namely, widgets, layouts and signals.