OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Rapid GUI Programming with Python and Qt

The Definitive Guide to PyQt Programming

Mark Summerfield

Foreword by Phil Thompson, Creator of PyQt

Rapid GUI Programming with Python and Qt

Actions are added to menus and toolbars using addAction(). To reduce typing we have created a tiny helper method which can be used to add actions to a menu or to a toolbar, and which can also add separators. Here is its code:

```
def addActions(self, target, actions):
    for action in actions:
        if action is None:
            target.addSeparator()
        else:
            target.addAction(action)
```

The target is a menu or toolbar, and actions is a list or tuple of actions or Nones. We could have used the built-in QWidget.addActions() method, but in that case we would have to create separator actions (shown later) rather than use Nones.

The last option on the Edit menu, Mirror, has a small triangle on its right. This signifies that it has a submenu.

```
mirrorMenu = editMenu.addMenu(QIcon(":/editmirror.png"),
                              "&Mirror")
self.addActions(mirrorMenu, (editUnMirrorAction,
        editMirrorHorizontalAction, editMirrorVerticalAction))
```

Submenus are populated in exactly the same way as any other menu, but they are added to their parent menu using QMenu.addMenu() rather than to the main window's menu bar using QMainWindow.menuBar().addMenu(). Having created the mirror menu, we add actions to it using our addActions() helper method, just as we did before.

Most menus are created and then populated with actions in the same way as the Edit menu, but the File menu is different.

```
self.fileMenu = self.menuBar().addMenu("&File")
self.fileMenuActions = (fileNewAction, fileOpenAction,
        fileSaveAction, fileSaveAsAction, None,
        filePrintAction, fileQuitAction)
self.connect(self.fileMenu, SIGNAL("aboutToShow()"),
             self.updateFileMenu)
```

We want the File menu to show recently used files. For this reason, we do not populate the File menu here, but instead generate it dynamically whenever the user invokes it. This is why we made the File menu an instance variable, and also why we have an instance variable holding the File menu's actions. The connection ensures that whenever the File menu is invoked our updateFileMenu() slot will be called. We will review this slot later on.

The Help menu is created conventionally, in the same way as the Edit menu, so we won't show it.

Figure 6.5 *The File toolbar*

With the menus in place, we can now turn to the toolbars.

```
fileToolbar = self.addToolBar("File")
fileToolbar.setObjectName("FileToolBar")
self.addActions(fileToolbar, (fileNewAction, fileOpenAction,
                              fileSaveAsAction))
```

Creating a toolbar is similar to creating a menu: We call `addToolBar()` to create a `QToolBar` object and populate it using `addActions()`. We can use our `addActions()` method for both menus and toolbars because their APIs are very similar, with both providing `addAction()` and `addSeparator()` methods. We set an object name so that PyQt can save and restore the toolbar's position—there can be any number of toolbars and PyQt uses the object name to distinguish between them, just as it does for dock widgets. The resulting toolbar is shown in Figure 6.5.

The edit toolbar and the checkable actions ("edit invert", "edit swap red and blue", and the mirror actions) are all created in the same way. But as Figure 6.6 shows, the edit toolbar has a spinbox in addition to its toolbar buttons. In view of this, we will show the code for this toolbar in full, showing it in two parts for ease of explanation.

```
editToolbar = self.addToolBar("Edit")
editToolbar.setObjectName("EditToolBar")
self.addActions(editToolbar, (editInvertAction,
        editSwapRedAndBlueAction, editUnMirrorAction,
        editMirrorVerticalAction,
        editMirrorHorizontalAction))
```

Creating a toolbar and adding actions to it is the same for all toolbars.

We want to provide the user with a quick means of changing the zoom factor, so we provide a spinbox in the edit toolbar to make this possible. Earlier, we put a separate "edit zoom" action in the Edit menu, to cater to keyboard users.

```
self.zoomSpinBox = QSpinBox()
self.zoomSpinBox.setRange(1, 400)
self.zoomSpinBox.setSuffix(" %")
self.zoomSpinBox.setValue(100)
self.zoomSpinBox.setToolTip("Zoom the image")
self.zoomSpinBox.setStatusTip(self.zoomSpinBox.toolTip())
self.zoomSpinBox.setFocusPolicy(Qt.NoFocus)
self.connect(self.zoomSpinBox,
             SIGNAL("valueChanged(int)"), self.showImage)
editToolbar.addWidget(self.zoomSpinBox)
```

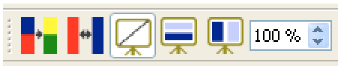

Figure 6.6 *The Edit toolbar*

The pattern for adding widgets to a toolbar is always the same: We create the widget, set it up, connect it to something to handle user interaction, and add it to the toolbar. We have made the spinbox an instance variable because we will need to access it outside the main window's initializer. The `addWidget()` call passes ownership of the spinbox to the toolbar.

We have now fully populated the menus and toolbars with actions. Although every action was added to the menus, some were not added to the toolbars. This is quite conventional; usually only the most frequently used actions are added to toolbars.

Earlier we saw the following line of code:

```
self.imageLabel.setContextMenuPolicy(Qt.ActionsContextMenu)
```

This tells PyQt that if actions are added to the `imageLabel` widget, they are to be used for a context menu, such as the one shown in Figure 6.7.

```
self.addActions(self.imageLabel, (editInvertAction,
        editSwapRedAndBlueAction, editUnMirrorAction,
        editMirrorVerticalAction, editMirrorHorizontalAction))
```

We can reuse our `addActions()` method to add actions to the label widget, providing we don't pass `None`s since `QWidget` does not have an `addSeparator()` method. Setting the policy and adding actions to a widget are all that is necessary to get a context menu for that widget.

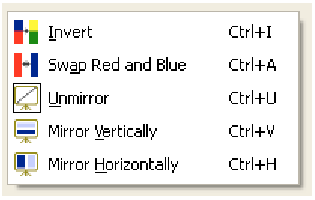

Figure 6.7 *The Image Label's context menu*

The `QWidget` class has an `addAction()` method that is inherited by the `QMenu`, `QMenuBar`, and `QToolBar` classes. This is why we can add actions to any of these classes. Although the `QWidget` class does not have an `addSeparator()` method, one is provided for convenience in the `QMenu`, `QMenuBar`, and `QToolBar` classes. If we want to add a separator to a context menu, we must do so by adding a separator action. For example:

```
    separator = QAction(self)
    separator.setSeparator(True)
    self.addActions(editToolbar, (editInvertAction,
            editSwapRedAndBlueAction, separator, editUnMirrorAction,
            editMirrorVerticalAction, editMirrorHorizontalAction))
```

If we need more sophisticated context menu handling—for example, where the menu's actions vary depending on the application's state, we can reimplement the relevant widget's contextMenuEvent() event-handling method. Event handling is covered in Chapter 10.

contextMenuEvent() ☞ 307

When we create a new image or load an existing image, we want the user interface to revert to its original state. In particular, we want the "edit invert" and "edit swap red and green" actions to be "off", and the mirror action to be "edit unmirrored".

```
            self.resetableActions = ((editInvertAction, False),
                                     (editSwapRedAndBlueAction, False),
                                     (editUnMirrorAction, True))
```

We have created an instance variable holding a tuple of pairs, with each pair holding an action and the checked state it should have when a new image is created or loaded. We will see resetableActions in use when we review the fileNew() and loadFile() slots.

In the Image Changer application, all of the actions are enabled all of the time. This is fine, since we always check for a null image before performing any action, but it has the disadvantage that, for example, "file save" will be enabled if there is no image or if there is an unchanged image, and similarly, the edit actions will be enabled even if there is no image. The solution is to enable or disable actions depending on the application's state, as the sidebar in Chapter 13 shows.

Enabling and Disabling Actions sidebar ☞ 385

Restoring and Saving the Main Window's State

Now that the main window's user interface has been fully set up, we are almost ready to finish the initializer method, but before we do we will restore the application's settings from the previous run (or use default settings if this is the very first time the application has been run).

Before we can look at application settings, though, we must make a quick detour and look at the creation of the application object and how the main window itself is created. The very last executable statement in the imagechanger.pyw file is the bare function call:

```
    main()
```

As usual, we have chosen to use a conventional name for the first function we execute. Here is its code:

```
def main():
    app = QApplication(sys.argv)
    app.setOrganizationName("Qtrac Ltd.")
    app.setOrganizationDomain("qtrac.eu")
    app.setApplicationName("Image Changer")
    app.setWindowIcon(QIcon(":/icon.png"))
    form = MainWindow()
    form.show()
    app.exec_()
```

The function's first line is one we have seen many times before. The next three lines are new. Our primary use of them is for loading and saving application settings. If we create a `QSettings` object without passing any arguments, it will use the organization name or domain (depending on platform), and the application name that we have set here. So, by setting these once on the application object, we don't have to remember to pass them whenever we need a `QSettings` instance.

But what do these names mean? They are used by PyQt to save the application's settings in the most appropriate place—for example, in the Windows registry, or in a directory under `$HOME/.config` on Linux, or in `$HOME/Library/Preferences` on Mac OS X. The registry keys or file and directory names are derived from the names we give to the application object.

We can tell that the icon file is loaded from the `qrc_resources` module because its path begins with `:/`.

After we have set up the application object, we create the main window, show it, and start off the event loop, in the same way as we have done in examples in previous chapters.

Now we can return to where we got up to in the `MainWindow.__init__()` method, and see how it restores system settings.

```
        settings = QSettings()
        self.recentFiles = settings.value("RecentFiles").toStringList()
        size = settings.value("MainWindow/Size",
                              QVariant(QSize(600, 500))).toSize()
        self.resize(size)
        position = settings.value("MainWindow/Position",
                                  QVariant(QPoint(0, 0))).toPoint()
        self.move(position)
        self.restoreState(
                settings.value("MainWindow/State").toByteArray())

        self.setWindowTitle("Image Changer")
        self.updateFileMenu()
        QTimer.singleShot(0, self.loadInitialFile)
```

We begin by creating a QSettings object. Since we passed no arguments, the names held by the application object are used to locate the settings information. We begin by retrieving the recently used files list. The QSettings.value() method always returns a QVariant, so we must convert it to the data type we are expecting.

Next, we use the two-argument form of value(), where the second argument is a default value. This means that the very first time the application is run, it has no settings at all, so we will get a QSize() object with a width of 600 pixels and a height of 500 pixels.[*] On subsequent runs, the size returned will be whatever the size of the main window was when the application was terminated—so long as we remember to save the size when the application terminates. Once we have a size, we resize the main window to the given size. After getting the previous (or default) size, we retrieve and set the position in exactly the same way.

There is no flickering, because the resizing and positioning are done in the main window's initializer, before the window is actually shown to the user.

Qt 4.3

Qt 4.2 introduced two new QWidget methods for saving and restoring a top-level window's geometry. Unfortunately, a bug meant that they were not reliable in all situations on X11-based systems, and for this reason we have restored the window's size and position as separate items. Qt 4.3 has fixed the bug, so with Qt 4.3 (e.g., with PyQt 4.3), instead of retrieving the size and position and calling resize() and move(), everything can be done using a single line:

```
self.restoreGeometry(settings.value("Geometry").toByteArray())
```

closeEvent() ☞ 185

This assumes that the geometry was saved when the application was terminated, as we will see when we look at the closeEvent().

The QMainWindow class provides a restoreState() method and a saveState() method; these methods restore from and save to a QByteArray. The data they save and restore are the dock window sizes and positions, and the toolbar positions—but they work only for dock widgets and toolbars that have unique object names.

After setting the window's title, we call updateFileMenu() to create the File menu. Unlike the other menus, the File menu is generated dynamically; this is so that it can show any recently used files. The connection from the File menu's aboutToShow() signal to the updateFileMenu() method means that the File menu is created afresh whenever the user clicks File in the menu bar, or presses Alt+F. But until this method has been called for the first time, the File menu does not exist—which means that the keyboard shortcuts for actions that have not been added to a toolbar, such as Ctrl+Q for "file quit", will not work. In view of this, we explicitly call updateFileMenu() to create an initial File menu and to activate the keyboard shortcuts.

[*]PyQt's documentation rarely gives units of measurement because it is assumed that the units are pixels, except for QPrinter, which uses points.