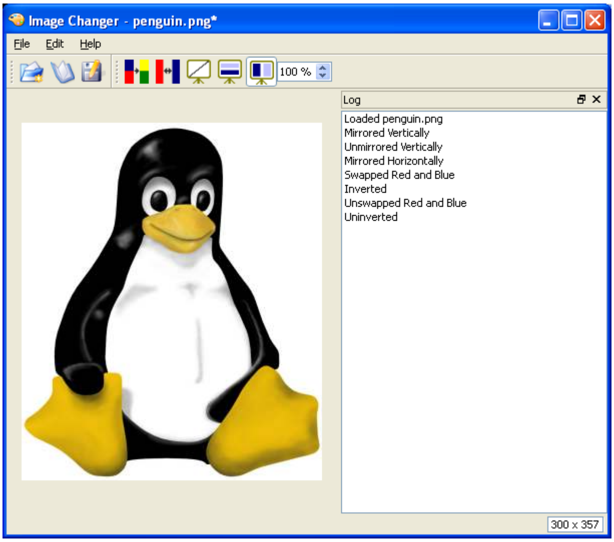
6

Main Windows

Most applications are main-window-style applications, that is, they have a menu bar, toolbars, a status bar, a central area, and possibly dock windows, to provide the user with a rich yet navigable and comprehensible user interface. In this chapter, we will see how to create a main-window-style application which demonstrates how to create and use all of these features.



```
Loaded penguin.png
Mirrored Vertically
Unmirrored Vertically
Mirrored Horizontally
Swapped Red and Blue
Inverted
Unswapped Red and Blue
Uninverted
```

Figure 6.1 *The Image Changer application*

We will use the Image Changer application shown in Figure 6.1 to demonstrate how to create a main-window-style application. Like most such applications it has menus, toolbars, and a status bar; it also has a dock window. In addition to seeing how to create all these user interface elements, we will cover how to relate user interactions with them, to methods that perform the relevant actions.

This chapter also explains how to handle the creation of new files and the opening of existing files, including keeping the user interface synchronized with the application's state. Also covered is how to give the user the opportunity to save unsaved changes, and how to manage a recently used files list. We will also show how to save and restore user preferences, including the sizes and positions of the main window and of the toolbars and dock windows.

Most applications have a data structure for holding their data, and use one or more widgets through which users can view and edit the data. The Image Changer application holds its data in a single `QImage` object, and uses a `QLabel` widget as its data viewer. In Chapter 8, we will see a main-window-style application that is used to present and edit lots of data items, and in Chapter 9, we will see how to create main window applications that can handle multiple documents.

Before looking at how to create the application, we will discuss some of the state that a user interface must maintain. Quite often, some menu options and toolbar buttons are "checkable", that is, they can be in one of two states. For example, in a word processor, a toolbar button for toggling italic text could be "on" (pushed down) or "off". If there is also an italic menu option, we must make sure that the menu option and the toolbar button are kept in sync. Fortunately, PyQt makes it easy to automate such synchronization.

Some options may be interdependent. For example, we can have text left-aligned, centered, or right-aligned, but only one of these can be "on" at any one time. So if the user switched on centered alignment, the left and right alignment toolbar buttons and menu options must be switched off. Again, PyQt makes it straightforward to synchronize such interdependent options. In this chapter, we will cover options that are noncheckable, such as "file open", and both independent and interdependent checkable options.

Although some menu and toolbar options can have an immediate effect on the application's data, others are used to invoke dialogs through which users can specify precisely what they want done. Since we have given so much coverage to dialogs in the preceding two chapters, here we will concentrate on how they are used rather than on how they are created. In this chapter we will see how to invoke custom dialogs, and also how to use many of PyQt's built-in dialogs, including dialogs for choosing a filename, the print dialog, and dialogs for asking the user for an item of data, such as a string or a number.

Creating a Main Window

For most main-window-style applications, the creation of the main window follows a similar pattern. We begin by creating and initializing some data structures, then we create a "central widget" which will occupy the main window's central area, and then we create and set up any dock windows. Next, we create "actions" and insert them into menus and toolbars. It is quite common to also read in the application's settings, and for applications that restore the

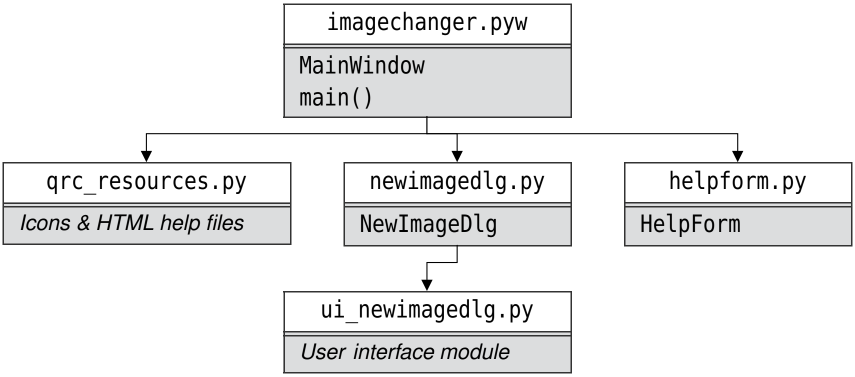


Figure 6.2 *The Image Changer's modules, classes, and functions*

user's workspace, to load the files that the application had open when it was last terminated.

The files that make up the Image Changer application are shown in Figure 6.2. The application's main window class is in the file `chap06/imagechanger.pyw`. The initializer is quite long, so we will look at it in pieces. But first we will look at the imports that precede the class definition.

```
import os
import platform
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import helpform
import newimagedlg
import qrc_resources

__version__ = "1.0.0"
```

In this book, the practice is to import Python's standard modules, then third-party modules (such as PyQt), and then our own modules. We will discuss the items we use from the `os` and `platform` modules when we use them in the code. The `sys` module is used to provide `sys.argv` as usual. The `helpform` and `newimagedlg` modules provide the `HelpForm` and `NewImageDlg` classes. We will discuss the `qrc_resources` module later on.

It is common for applications to have a version string, and conventional to call it `__version__`; we will use it in the application's about box.

Now we can look at the beginning of the `MainWindow` class.

```
class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.image = QImage()
        self.dirty = False
        self.filename = None
        self.mirroredvertically = False
        self.mirroredhorizontally = False
```

The initializer begins conventionally with the `super()` call. Next, we create a null `QImage` that we will use to hold the image the user loads or creates. A `QImage` is not a `QObject` subclass, so it does not need a parent; instead, we can leave its deletion to Python's normal garbage collection when the application terminates. We also create some instance variables. We use `dirty` as a Boolean flag to indicate whether the image has unsaved changes. The `filename` is initially set to `None`, which we use to signify that either there is no image, or there is a newly created image that has never been saved.

PyQt provides various mirroring capabilities, but for this example application we have limited ourselves to just three possibilities: having the image mirrored vertically, horizontally, or not at all. We need to keep track of the mirrored state so that we can keep the user interface in sync, as we will see when we discuss the mirroring actions.

```
        self.imageLabel = QLabel()
        self.imageLabel.setMinimumSize(200, 200)
        self.imageLabel.setAlignment(Qt.AlignCenter)
        self.imageLabel.setContextMenuPolicy(Qt.ActionsContextMenu)
        self.setCentralWidget(self.imageLabel)
```

In some applications the central widget is a composite widget (a widget that is composed of other widgets, laid out just like those in a dialog), or an item-based widget (such as a list or table), but here a single `QLabel` is sufficient. A `QLabel` can display plain text, or HTML, or an image in any of the image formats that PyQt supports; later on we will see how to discover what these formats are, since they can vary. We have set a minimum size because initially the label has nothing to show, and would therefore take up no space, which would look peculiar. We have chosen to align our images vertically and horizontally centered.

PyQt offers many ways of creating context menus, but we are going to use the easiest and most common approach. First, we must set the context menu policy for the widget which we want to have a context menu. Then, we must add some actions to the widget—something we will do further on. When the user invokes the context menu, the menu will pop up, displaying the actions that were added.

Unlike dialogs, where we use layouts, in a main-window-style application we only ever have one central widget—although this widget could be composite, so there is no limitation in practice. We only need to call `setCentralWidget()` and we are done. This method both lays out the widget in the main window's central area, and reparents the widget so that the main window takes ownership of it.

Object Ownership sidebar 119 ☜

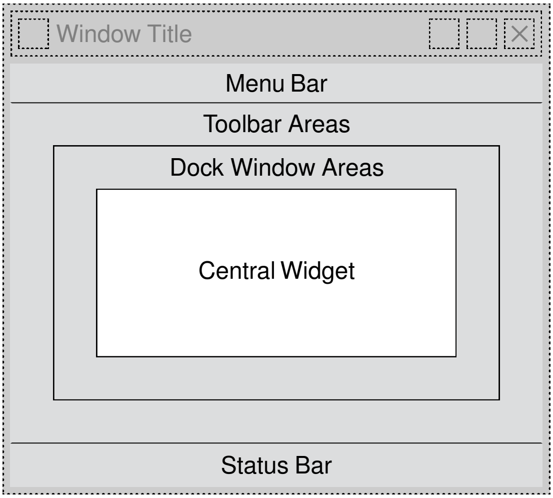


Figure 6.3 *QMainWindow's areas*

Toolbars are suitable for holding toolbar buttons, and some other kinds of widgets such as comboboxes and spinboxes. For larger widgets, for tool palettes, or for any widget that we want the user to be able to drag out of the window to float freely as an independent window in its own right, using a dock window is often the right choice.

Dock windows are windows that can appear in the dock areas shown in Figure 6.3. They have a small caption, and restore and close buttons, and they can be dragged from one dock area to another, or float freely as independent top-level windows in their own right. When they are docked they automatically provide a splitter between themselves and the central area, and this makes them easy to resize.

In PyQt, dock windows are instances of the QDockWidget class. We can add a single widget to a dock widget, just as we can have a single widget in a main window's central area, and in the same way this is no limitation, since the widget added can be a composite.

```
logDockWidget = QDockWidget("Log", self)
logDockWidget.setObjectName("LogDockWidget")
logDockWidget.setAllowedAreas(Qt.LeftDockWidgetArea|
                              Qt.RightDockWidgetArea)
self.listWidget = QListWidget()
logDockWidget.setWidget(self.listWidget)
self.addDockWidget(Qt.RightDockWidgetArea, logDockWidget)
```

Dock widgets are not put into a layout, so when we create them, in addition to providing their window caption, we must give them a parent. By setting a parent, we ensure that the dock widget does not go out of scope and get garbage-

Object Ownership sidebar 119 ☜

collected by Python at the wrong time. Instead, the dock widget will be deleted when its parent, the top-level window (the main window), is deleted.

Every PyQt object can be given an object name, although up to now we have never done so. Object names can sometimes be useful in debugging, but we have set one here because we want PyQt to save and restore the dock widget's size and position, and since there could be any number of dock widgets, PyQt uses the object name to distinguish between them.

By default, dock widgets can be dragged into any dock area and are movable, floatable, and closable. Since our dock widget is going to be used to store a list—a widget that is usually tall and narrow—it only makes sense for it to be in the left or right dock areas (or to float), so we use `setAllowedAreas()` to restrict the areas. Dock widgets also have a `setFeatures()` method which is used to control whether the dock widget can be moved, floated, or closed, but we do not need to use it here because the defaults are fine.

Once the dock widget has been set up, we create the widget it will hold, in this case a list widget. Then we add the widget to the dock widget, and the dock widget to the main window. We did not have to give the list widget a parent because when it is added to the dock widget the dock widget takes ownership of it.

```
        self.printer = None
```

We want users to be able to print out their images. To do this we need to create a `QPrinter` object. We could create the printer whenever we need it and leave it to be garbage-collected afterward. But we prefer to keep an instance variable, initially set to `None`. The first time the user asks to print we will create a `QPrinter` and assign it to our `printer` variable. This has two benefits. First, we create the printer object only when it is needed, and second, because we keep a reference to it, it stays around—and keeps all its previous state such as the user's choice of printer, paper size, and so on.

```
        self.sizeLabel = QLabel()
        self.sizeLabel.setFrameStyle(QFrame.StyledPanel|QFrame.Sunken)
        status = self.statusBar()
        status.setSizeGripEnabled(False)
        status.addPermanentWidget(self.sizeLabel)
        status.showMessage("Ready", 5000)
```

For the application's status bar, we want the usual message area on the left, and a status indicator showing the width and height of the current image. We do this by creating a `QLabel` widget and adding it to the status bar. We also switch off the status bar's size grip since that seems inappropriate when we have an indicator label that shows the image's dimensions. The status bar itself is created for us the first time we call the `QMainWindow`'s `statusBar()` method. If we call the status bar's `showMessage()` method with a string, the string will be displayed in the status bar, and will remain on display until either another `showMessage()`

call supplants it or until `clearMessage()` is called. We have used the two-argument form, where the second argument is the number of milliseconds (5 000, i.e., 5 seconds), that the message should be shown for; after this time the status bar will clear itself.

So far we have seen how to create the main window's central widget, create a dock widget, and set up the status bar. Now we are almost ready to create the menus and toolbars, but first we must understand what PyQt actions are, and then take a brief detour to learn about resources.

Actions and Key Sequences

Qt's designers recognized that user interfaces often provide several different ways for the user to achieve the same thing. For example, creating a new file in many applications can be done via the File→New menu option, or by clicking the New File toolbar button, , or by using the Ctrl+N keyboard shortcut. In general, we do not care how the user asked to perform the action, we only care what action they asked to be done. PyQt encapsulates user actions using the `QAction` class. So, for example, to create a "file new" action we could write code like this:

```
fileNewAction = QAction(QIcon("images/filenew.png"), "&New", self)
fileNewAction.setShortcut(QKeySequence.New)
helpText = "Create a new image"
fileNewAction.setToolTip(helpText)
fileNewAction.setStatusTip(helpText)
self.connect(fileNewAction, SIGNAL("triggered()"), self.fileNew)
```

This assumes that we have a suitable icon and a `fileNew()` method. The ampersand in the menu item's text means that the menu item will appear as New (except on Mac OS X or unless the windowing system is set to suppress underlines), and that keyboard users will be able to invoke it by pressing Alt+F,N, assuming that the File menu's text is `"&File"` so that it appears as File. Alternatively, the user could use the shortcut that was created by `setShortcut()`, and simply press Ctrl+N instead.

Many key sequences are standardized, some even across different windowing systems. For example, Windows, KDE, and GNOME all use Ctrl+N for "new" and Ctrl+S for "save". Mac OS X is similar, with Command+N and Command+S for these actions. The `QKeySequence` class in PyQt 4.2 provides constants for the standardized key sequences, such as `QKeySequence.New`. This is especially useful when the standardized key sequences differ across windowing systems, or where more than one key sequence is associated with an action. For example, if we set a shortcut to `QKeySequence.Paste`, PyQt will trigger a "paste" action in response to Ctrl+V or Shift+Ins on Windows; Ctrl+V, Shift+Ins, or F18 on KDE and GNOME; and Command+V on Mac OS X.

Qt 4.2

For key sequences that are not standardized (or if we want backward compatibility with earlier PyQt releases), we can provide the shortcut as a string;

Table 6.1 *Selected QAction Methods*

Syntax	Description
`a.data()`	Returns `QAction` `a`'s user data as a `QVariant`
`a.setData(v)`	Sets `QAction` `a`'s user data to `QVariant` `v`
`a.isChecked()`	Returns `True` if `QAction` `a` is checked
`a.setChecked(b)`	Checks or unchecks `QAction` `a` depending on `bool` `b`
`a.isEnabled()`	Returns `True` if `QAction` `a` is enabled
`a.setEnabled(b)`	Enables or disables `QAction` `a` depending on `bool` `b`
`a.setSeparator(b)`	Sets `QAction` `a` to be a normal action or a separator depending on `bool` `b`
`a.setShortcut(k)`	Sets `QAction` `a`'s keyboard shortcut to `QKeySequence` `k`
`a.setStatusTip(s)`	Sets `QAction` `a`'s status tip text to string `s`
`a.setText(s)`	Sets `QAction` `a`'s text to string `s`
`a.setToolTip(s)`	Sets `QAction` `a`'s tooltip text to string `s`
`a.setWhatsThis(s)`	Sets `QAction` `a`'s What's This? text to string `s`
`a.toggled(b)`	This signal is emitted when `QAction` `a`'s checked status changes; `bool` `b` is `True` if the action is checked
`a.triggered(b)`	This signal is emitted when `QAction` `a` is invoked; the optional `bool` `b` is `True` if `QAction` `a` is checked

for example, `setShortcut("Ctrl+Q")`. This book uses the standardized key sequences that are available, and otherwise falls back to using strings.

Object Ownership sidebar 119 ☜

Notice that we give the `QAction` a parent of `self` (the form in which the action is applicable). It is important that every `QObject` subclass (except top-level windows) has a parent; for widgets this is usually achieved by laying them out, but for a pure data object like a `QAction`, we must provide the parent explicitly.

Once we have created the action, we can add it to a menu and to a toolbar like this:

```
fileMenu.addAction(fileNewAction)
fileToolbar.addAction(fileNewAction)
```

Now whenever the user invokes the "file new" action (by whatever means), the `fileNew()` method will be called.

Resource Files

Unfortunately, there is a small problem with the code we have written. It assumes that the application's working directory is the directory where it is located. This is the normal case under Windows where the `.pyw` (or a shortcut to it) is clicked (or double-clicked). But if the program is executed from the command line from a different directory—for example, ./chap06/imagechanger.pyw—none of the icons will appear. This is because we gave the icon's path as images, that is, a path relative to the application's working directory, so when invoked from elsewhere, the icons were looked for in the ./images directory (which might not even exist), when in fact they are in the ./chap06/images directory.

We might be tempted to try to solve the problem using Python's os.getcwd() function; but this returns the directory where we invoked the application, which as we have noted, may not be the directory where the application actually resides. Nor does PyQt's QApplication.applicationDirPath() method help, since this returns the path to the Python executable, not to our application itself. One solution is to use os.path.dirname(__file__) to provide a prefix for the icon filenames, since the __file__ variable holds the full name and path of the current .py or .pyw file.

Another solution is to put all our icons (and help files, and any other small resources) into a single .py module and access them all from there. This not only solves the path problem (because Python knows how to look for a module to be imported), but also means that instead of having dozens of icons, help files, and similar, some of which could easily become lost, we have a single module containing them all.

To produce a resource module we must do two things. First, we must create a .qrc file that contains details of the resources we want included, and then we must run pyrcc4 which reads a .qrc file and produces a resource module. The .qrc file is in a simple XML format that is easy to write by hand. Here is an extract from the resources.qrc file used by the Image Changer application:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
<file alias="filenew.png">images/filenew.png</file>
<file alias="fileopen.png">images/fileopen.png</file>
...
<file alias="icon.png">images/icon.png</file>

<file>help/editmenu.html</file>
<file>help/filemenu.html</file>
<file>help/index.html</file>
</qresource>
</RCC>
```

The ellipsis represents many lines that have been omitted to save space because they are all very similar. Each <file> entry must contain a filename, with its relative path if it is in a subdirectory. Now, if we want to use the "file new" action's image, we could write QIcon(":/images/filenew.png"). But thanks to the alias, we can shorten this to QIcon(":/filenew.png"). The leading :/ tells PyQt that the file is a resource. Resource files can be treated just like normal (read-only) files in the filesystem, the only difference being that they have the special path prefix. But before we can use resources, we must make sure we generate the resource module and import it into our application.

Earlier we showed the imports for the Image Changer application, and the last one was import qrc_resources. The qrc_resources.py module was generated by pyrcc4 using the following command line:

```
C:\pyqt\chap06>pyrcc4 -o qrc_resources.py resources.qrc
```

We must run this command whenever we change the resources.qrc file.

As a convenience for readers, two small Python programs are provided with the examples to make using pyrcc4, and some other PyQt command-line programs, much easier. One is mkpyqt.py, itself a command-line program, and the other is Make PyQt, a GUI application written in PyQt4. This means, for example, that instead of running pyrcc4 ourselves, we can simply type this:

mkpyqt.py and Make PyQt sidebar ☞ 207

```
C:\pyqt\chap06>mkpyqt.py
```

Both mkpyqt.py and Make PyQt do the same thing: They run pyuic4 and other PyQt tools, and for each one they automatically use the correct command-line arguments; they are described in the next chapter.

Creating and Using Actions

The code we saw earlier for creating a “file new” action required six lines to create and set up the action. Most main-window-style applications have scores of actions, so typing six lines for each one would soon become very tedious. For this reason, we have created a helper method which allows us to reduce the code for creating actions to just two or three lines. We will look at the helper, and then see how it is used in the main window’s initializer.

```
    def createAction(self, text, slot=None, shortcut=None, icon=None,
                     tip=None, checkable=False, signal="triggered()"):
        action = QAction(text, self)
        if icon is not None:
            action.setIcon(QIcon(":/%s.png" % icon))
        if shortcut is not None:
            action.setShortcut(shortcut)
        if tip is not None:
            action.setToolTip(tip)
            action.setStatusTip(tip)
        if slot is not None:
            self.connect(action, SIGNAL(signal), slot)
        if checkable:
            action.setCheckable(True)
        return action
```

This method does everything that we did by hand for the “file new” action. In addition, it handles cases where there is no icon, as well as “checkable” actions. Icons are optional, although for actions that will be added to a toolbar it is conventional to provide one. An action is checkable if it can have “on” and “off” states like the Bold or Italic actions that word processors normally provide.

Notice that the last argument to the QAction constructor is self; this is the action’s parent (the main window) and it ensures that the action will not be garbage-collected when it goes out of the initializer’s scope. In some cases, we make actions instance variables so that we can access them outside the form’s initializer, something we don’t need to do in this particular example.

Here is how we can create the “file new” action using the createAction() helper method:

```
        fileNewAction = self.createAction("&New...", self.fileNew,
                QKeySequence.New, "filenew", "Create an image file")
```

With the exception of the “file quit” action (and “file save as”, for which we don’t provide a shortcut), the other file actions are created in the same way, so we won’t waste space by showing them.

```
        fileQuitAction = self.createAction("&Quit", self.close,
                "Ctrl+Q", "filequit", "Close the application")
```

The QKeySequence class does not have a standardized shortcut for application termination, so we have chosen one ourselves and specified it as a string. We could have just as easily used a different shortcut—for example, Alt+X or Alt+F4.

The close() slot is inherited from QMainWindow. If the main window is closed by invoking the “file quit” action (which we have just connected to the close() slot), for example, by clicking File→Quit or by pressing Ctrl+Q, the base class’s close() method will be called. But if the user clicks the application’s close button, X, the close() method is *not* called.

The only way we can be sure we are intercepting attempts to close the window is to reimplement the close event handler. Whether the application is closed by the close() method or via the close button, the close event handler is always called. So, by reimplementing this event handler we can give the user the opportunity to save any unsaved changes, and we can save the application’s settings.

In general, we can implement an application’s behavior purely through the high-level signals and slots mechanism, but in this one important case we must use the lower-level event-handling mechanism. However, reimplementing the close event is no different from reimplementing any other method, and it is not difficult, as we will see when we cover it further on. (Event handling is covered in Chapter 10.)

The editing actions are created in a similar way, but we will look at a few of them because of subtle differences.

```
        editZoomAction = self.createAction("&Zoom...", self.editZoom,
                "Alt+Z", "editzoom", "Zoom the image")
```

It is convenient for users to be able to zoom in and out to see an image in more or less detail. We have provided a spinbox in the toolbar to allow mouse users to change the zoom factor (and which we will come to shortly), but we must also support keyboard users, so for them we create an "edit zoom" action which will be added to the Edit menu. When triggered, the method connected to this action will pop up a dialog box where the user can enter a zoom percentage.

There are standardized key sequences for zoom in and for zoom out, but there is not one for zooming generally, so we have chosen to use Alt+Z in this case. (We did not use Ctrl+Z, since that is the standardized key sequence for undo on most platforms.)

```
        editInvertAction = self.createAction("&Invert",
                self.editInvert, "Ctrl+I", "editinvert",
                "Invert the image's colors", True, "toggled(bool)")
```

The "edit invert" action is a toggle action. We could still use the triggered() signal, but then we would need to call isChecked() on the action to find out its state. It is more convenient for us to use the toggled(bool) signal since that not only tells us that the action has been invoked, but also whether it is checked. Actions also have a triggered(bool) signal that is emitted only for user changes, but that is not suitable here because whether the checked status of the invert action is changed by the user or programmatically, we want to act on it.

The "edit swap red and blue" action is similar to the "edit invert" action, so we won't show it.

Like the "edit invert" action and the "edit swap red and blue" action, the mirror actions are also checkable, but unlike the "invert" and "swap red and blue" actions which are independent, we have chosen to make the mirror actions mutually exclusive, allowing only one to be "on" at any one time. To get this behavior we create the mirror actions in the normal way, but add each of them to an "action group". An action group is a class which manages a set of checkable actions and ensures that if one of the actions it manages is set to "on", the others are all set to "off".

```
        mirrorGroup = QActionGroup(self)
```

Object Ownership sidebar 119 ☜

An action group is a QObject subclass that is neither a top-level window nor a widget that is laid out, so we must give it an explicit parent to ensure that it is deleted by PyQt at the right time.

Once we have created the action group, we create the actions in the same way as before, only now we add each one to the action group.

```
editUnMirrorAction = self.createAction("&Unmirror",
        self.editUnMirror, "Ctrl+U", "editunmirror",
        "Unmirror the image", True, "toggled(bool)")
mirrorGroup.addAction(editUnMirrorAction)
```

We have not shown the code for the "edit mirror vertically" or "edit mirror horizontally" actions since it is almost identical to the code shown earlier.

```
editUnMirrorAction.setChecked(True)
```

Checkable actions default to being "off", but when we have a group like this where exactly one must be "on" at a time, we must choose one to be on in the first place. In this case, the "edit unmirror" action is the most sensible to switch on initially. Checking the action will cause it to emit its `toggled()` signal, but at this stage the `QImage` is null, and as we will see, no change is applied to a null image.

We create two more actions, "help about", and "help help", with code very similar to what we have already seen.

Although the actions are all in existence, none of them actually works! This is because they become operational only when they have been added to a menu, a toolbar, or both.

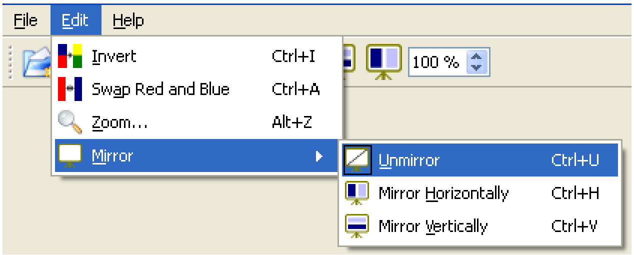


Figure 6.4 *The Edit menu and the Mirror submenu*

Menus in the menu bar are created by accessing the main window's menu bar (which is created the first time `menuBar()` is called, just like the status bar). Here is the code for creating the Edit menu:

```
editMenu = self.menuBar().addMenu("&Edit")
self.addActions(editMenu, (editInvertAction,
        editSwapRedAndBlueAction, editZoomAction))
```

We have created the Edit menu, and then used `addActions()` to add some actions to it. This is sufficient to produce the Edit menu shown in Figure 6.4, apart from the Mirror option, which we will look at in a moment.

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)
```

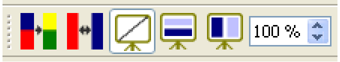


Figure 6.6 *The Edit toolbar*

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.

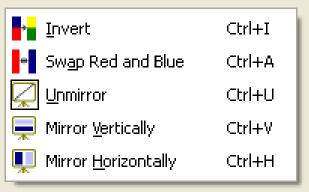


Figure 6.7 *The Image Label's context menu*

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.

Doing Lots of Processing at Start-Up

If we need to do lots of processing at start-up—for example, if we need to load in lots of large files, we always do so in a separate loading method. At the end of the main form's constructor, the loading method is called through a zero-timeout single-shot timer.

What would happen if we didn't use a single-shot timer? Imagine, for example, that the method was `loadInitialFiles()` and that it loaded lots of multimegabyte files. The file loading would be done when the main window was being created, that is, before the `show()` call, and before the event loop (`exec_()`) had been started. This means that the user might experience a long delay between launching the application and actually seeing the application's window appear on-screen. Also, if the file loading might result in message boxes being popped up—for example, to report errors—it makes more sense to have these appear after the main window is shown, and when the event loop is running.

We want the main window to appear as quickly as possible so that the user knows that the launch was successful, and so that they can see any long-running processes, like loading large files, through the main window's user interface. This is achieved by using a single-shot timer as we did in the Image Changer example.

This works because a single-shot timer with a timeout of zero does not execute the slot it is given immediately. Instead, it puts the slot to be called in the event queue and then simply returns. At this point, the end of the main window's initializer is reached and the initialization is complete. The very next statement (in `main()`) is a `show()` call on the main window, and this does nothing except add a show event to the event queue. So, now the event queue has a timer event and a show event. A timer event with a timeout of zero is taken to mean "do this when the event queue has nothing else to do", so when the next statement, `exec_()`, is reached and starts off the event loop, it always chooses to handle the show event first, so the form appears, and then, with no other events left, the single-shot timer's event is processed, and the `loadInitialFiles()` call is made.

The initializer's last line looks rather peculiar. A single-shot timer takes a timeout argument (in milliseconds), and a method to call when the timeout occurs. So, it looks as though the line could have been written like this instead:

```
    self.loadInitialFile()
```

In this application, where we load at most only one initial file, and where that file is very unlikely to be as big even as 1MB, we could use either approach without noticing any difference. Nonetheless, calling the method directly is not the same as using a single-shot timer with a zero timeout, as the Doing Lots of Processing at Start-Up sidebar explains.

We have finished reviewing the code for initializing the main window, so now we can begin looking at the other methods that must be implemented to provide the application's functionality. Although the Image Changer application is just one specific example, to the greatest extent possible we have made the code either generic or easily adaptable so that it could be used as the basis for other main-window-style applications, even ones that are completely different.

In view of the discussions we have just had, it seems appropriate to begin our coverage with the `loadInitialFile()` method.

```
def loadInitialFile(self):
    settings = QSettings()
    fname = unicode(settings.value("LastFile").toString())
    if fname and QFile.exists(fname):
        self.loadFile(fname)
```

This method uses a `QSettings` object to get the last image that the application used. If there was such an image, and it still exists, the program attempts to load it. We will review `loadFile()` when we cover the file actions.

We could just as easily have written `if fname and os.access(fname, os.F_OK):` It makes no noticable difference here, but for multiperson projects, it may be wise to have a policy of preferring PyQt over the standard Python libraries or vice versa in cases like this, just to keep things as simple and clear as possible.

We discussed restoring the application's state a little earlier, so it seems appropriate to cover the close event, since that is where we save the application's state.

```
def closeEvent(self, event):
    if self.okToContinue():
        settings = QSettings()
        filename = QVariant(QString(self.filename)) \
                if self.filename is not None else QVariant()
        settings.setValue("LastFile", filename)
        recentFiles = QVariant(self.recentFiles) \
                if self.recentFiles else QVariant()
        settings.setValue("RecentFiles", recentFiles)
        settings.setValue("MainWindow/Size", QVariant(self.size()))
        settings.setValue("MainWindow/Position",
                QVariant(self.pos()))
        settings.setValue("MainWindow/State",
                QVariant(self.saveState()))
    else:
        event.ignore()
```

If the user attempts to close the application, by whatever means (apart from killing or crashing it), the `closeEvent()` method is called. We begin by calling our own custom `okToContinue()` method; this returns `True` if the user really

Table 6.2 *Selected QMainWindow Methods*

Syntax	Description
`m.addDockWidget(a, d)`	Adds `QDockWidget` `d` into `Qt.QDockWidgetArea` `a` in `QMainWindow` `m`
`m.addToolBar(s)`	Adds and returns a new `QToolBar` called string `s`
`m.menuBar()`	Returns `QMainWindow` `m`'s `QMenuBar` (which is created the first time this method is called)
`m.restoreGeometry(ba)`	Restores `QMainWindow` `m`'s position and size to those encapsulated in `QByteArray` `ba` Qt 4.3
`m.restoreState(ba)`	Restores `QMainWindow` `m`'s dock widgets and toolbars to the state encapsulated in `QByteArray` `ba`
`m.saveGeometry()`	Returns `QMainWindow` `m`'s position and size encapsulated in a `QByteArray` Qt 4.3
`m.saveState()`	Returns the state of `QMainWindow` `m`'s dock widgets and toolbars, that is, their sizes and positions, encapsulated in a `QByteArray`
`m.setCentralWidget(w)`	Sets `QMainWindow` `m`'s central widget to be `QWidget` `w`
`m.statusBar()`	Returns `QMainWindow` `m`'s `QStatusBar` (which is created the first time this method is called)
`m.setWindowIcon(i)`	Sets `QMainWindow` `m`'s icon to `QIcon` `i`; this method is inherited from `QWidget`
`m.setWindowTitle(s)`	Sets `QMainWindow` `m`'s title to string `s`; this method is inherited from `QWidget`

wants to close, and `False` otherwise. It is inside `okToContinue()` that we give the user the chance to save unsaved changes. If the user does want to close, we create a fresh `QSettings` object, and store the "last file" (i.e., the file the user has open), the recently used files, and the main window's state. The `QSettings` class only reads and writes `QVariant` objects, so we must be careful to provide either null `QVariants` (created with `QVariant()`), or `QVariants` with the correct information in them.

If the user chose not to close, we call `ignore()` on the close event. This will tell PyQt to simply discard the close event and to leave the application running.

Qt 4.3

If we are using Qt 4.3 (e.g., with PyQt 4.3) and have restored the main window's geometry using `QWidget.restoreGeometry()`, we can save the geometry like this:

```
settings.setValue("Geometry", QVariant(self.saveGeometry()))
```

If we take this approach, we do not need to save the main window's size or position separately.

```
    def okToContinue(self):
        if self.dirty:
            reply = QMessageBox.question(self,
                            "Image Changer - Unsaved Changes",
                            "Save unsaved changes?",
                            QMessageBox.Yes|QMessageBox.No|
                            QMessageBox.Cancel)
            if reply == QMessageBox.Cancel:
                return False
            elif reply == QMessageBox.Yes:
                self.fileSave()
        return True
```

This method is used by the `closeEvent()`, and by the “file new” and “file open” actions. If the image is “dirty”, that is, if it has unsaved changes, we pop up a message box and ask the user what they want to do. If they click Yes, we save the image to disk and return `True`. If they click No, we simply return `True`, so the unsaved changes will be lost. If they click Cancel, we return `False`, which means that the unsaved changes are not saved, but the current image will remain current, so it could be saved later.

All the examples in the book use yes/no or yes/no/cancel message boxes to give the user the opportunity to save unsaved changes. An alternative favored by some developers is to use Save and Discard buttons (using the `QMessageBox.Save` and `QMessageBox.Discard` button specifiers), instead.

The recently used files list is part of the application’s state that must not only be saved and restored when the application is terminated and executed, but also kept current at runtime. Earlier we connected the `fileMenu`’s `aboutToShow()` signal to a custom `updateFileMenu()` slot. So, when the user presses Alt+F or clicks the File menu, this slot is called *before* the File menu is shown.

```
    def updateFileMenu(self):
        self.fileMenu.clear()
        self.addActions(self.fileMenu, self.fileMenuActions[:-1])
        current = QString(self.filename) \
                if self.filename is not None else None
        recentFiles = []
        for fname in self.recentFiles:
            if fname != current and QFile.exists(fname):
                recentFiles.append(fname)
        if recentFiles:
            self.fileMenu.addSeparator()
            for i, fname in enumerate(recentFiles):
                action = QAction(QIcon(":/icon.png"), "&%d %s" % (
                        i + 1, QFileInfo(fname).fileName()), self)
                action.setData(QVariant(fname))
                self.connect(action, SIGNAL("triggered()"),
                             self.loadFile)
                self.fileMenu.addAction(action)
```

The Static QMessageBox Methods

The QMessageBox class offers several static convenience methods that pop up a modal dialog with a suitable icon and buttons. They are useful for offering users dialogs that have a single OK button, or Yes and No buttons, and similar.

The most commonly used QMessageBox static methods are critical(), information(), question(), and warning(). The methods take a parent widget (over which they center themselves), window title text, message text (which can be plain text or HTML), and zero or more button specifications. If no buttons are specified, a single OK button is provided.

The buttons can be specified using constants, or we can provide our own text. In Qt 4.0 and Qt 4.1, it was very common to bitwise OR QMessageBox.Default with OK or Yes buttons—this means the button will be pressed if the user presses Enter, and to bitwise OR QMessageBox.Escape with the Cancel or No buttons, which will then be pressed if the user presses Esc. For example:

```
reply = QMessageBox.question(self,
        "Image Changer - Unsaved Changes", "Save unsaved changes?",
        QMessageBox.Yes|QMessageBox.Default,
        QMessageBox.No|QMessageBox.Escape)
```

The methods return the constant of the button that was pressed.

Qt 4.2

From Qt 4.2, the QMessageBox API has been simplified so that instead of specifying buttons and using bitwise ORs, we can just use buttons. For example, for a yes/no/cancel dialog we could write:

```
reply = QMessageBox.question(self,
        "Image Changer - Unsaved Changes", "Save unsaved changes?",
        QMessageBox.Yes|QMessageBox.No|QMessageBox.Cancel)
```

In this case, PyQt will automatically make the Yes (accept) button the default button, activated by the user pressing Enter, and the Cancel (reject) button the escape button, activated by the user pressing Esc. The QMessageBox methods also make sure that the buttons are shown in the correct order for the platform. We use the Qt 4.2 syntax for the examples in this book.

The message box is closed by the user clicking the “accept” button (often Yes or OK) or the “reject” button (often No or Cancel). The user can also, in effect, press the “reject” button by clicking the window’s close button, X, or by pressing Esc.

If we want to create a customized message box—for example, using custom button texts and a custom icon—we can create a QMessageBox instance. We can then use methods such as QMessageBox.addButton() and QMessageBox.setIcon(), and pop up the message box by calling QMessageBox.exec_().

```
            self.fileMenu.addSeparator()
            self.fileMenu.addAction(self.fileMenuActions[-1])
```

We begin by clearing all the File menu's actions. Then we add back the original list of file menu actions, such as "file new" and "file open", but excluding the last one, "file quit". Then we iterate over the recently used files list, creating a local list which only contains files that still exist in the filesystem, and excluding the current file. Although it does not seem to make much sense, many applications include the current file, often showing it first in the list.

Now, if there are any recently used files in our local list we add a separator to the menu and then create an action for each one with text that just contains the filename (without the path), preceded by a numbered accelerator: 1, 2, …, 9. PyQt's QFileInfo class provides information on files similar to some of the functions offered by Python's os module. The QFileInfo.fileName() method is equivalent to os.path.basename(). For each action, we also store an item of "user data"—in this case, the file's full name, including its path. Finally, we connect each recently used filename's action's triggered() signal to the loadFile() slot, and add the action to the menu. (We cover loadFile() in the next section.) At the end, we add another separator, and the File menu's last action, "file quit".

But how is the recently used files list created and maintained? We saw in the form's initializer that we initially populate the recentFiles string list from the application's settings. We have also seen that the list is correspondingly saved in the closeEvent(). New files are added to the list using addRecentFile().

```
        def addRecentFile(self, fname):
            if fname is None:
                return
            if not self.recentFiles.contains(fname):
                self.recentFiles.prepend(QString(fname))
                while self.recentFiles.count() > 9:
                    self.recentFiles.takeLast()
```

This method prepends the given filename, and then pops off any excess files from the end (the ones added longest ago) so that we never have more than nine filenames in our list. We keep the recentFiles variable as a QStringList, which is why we have used QStringList methods rather than Python list methods on it.

The addRecentFile() method itself is called inside the fileNew(), fileSaveAs(), and loadFile() methods; and indirectly from loadInitialFile(), fileOpen(), and updateFileMenu(), all of which either call or connect to loadFile(). So, when we save an image for the first time, or under a new name, or create a new image, or open an existing image, the filename is added to the recently used files list. However, the newly added filename will not appear in the File menu, unless we subsequently create or open another image, since our updateFileMenu() method does not display the current image's filename in the recently used files list.

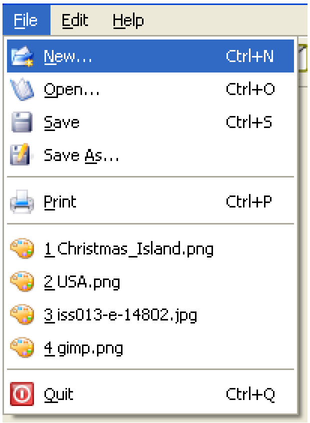


Figure 6.8 *The File menu with some recently used files*

The approach to handling recently used files that we have taken here is just one of many possibilities. An alternative is to create the File menu just once, with a set of actions at the end for recently used files. When the menu is updated, instead of being cleared and re-created, the actions set aside for recently used files are simply hidden or shown, in the latter case having had their filenames updated to reflect the current set of recently used files. From the user's point of view, there is no discernable difference whichever approach we take under the hood, so in either case the File menu will look similar to the one shown in Figure 6.8.

Both approaches can be used to implement recently used files in a File menu, adding the list at the end as we have done in the Image Changer application, just before the Quit option. They can also both be used to implement the Open Recent File menu option that has all the recent files as a submenu, as used by OpenOffice.org and some other applications. The benefits of using a separate Open Recent File option is that the File menu is always the same, and full paths can be shown in the submenu—something we avoid when putting recently used files directly in the File menu so that it doesn't become extremely wide (and therefore, ugly).

Handling User Actions

In the preceding section, we created the appearance of our main-window-style application and provided its behavioral infrastructure by creating a set of actions. We also saw how to save and restore application settings, and how to manage a recently used files list.

Some of an application's behavior is automatically handled by PyQt—for example, window minimizing, maximizing, and resizing—so we do not have to do this ourselves. Some other behaviors can be implemented purely through signals and slots connections. In this section we are concerned with the actions that are directly under the control of the user and which can be used to view, edit, and output, their data.

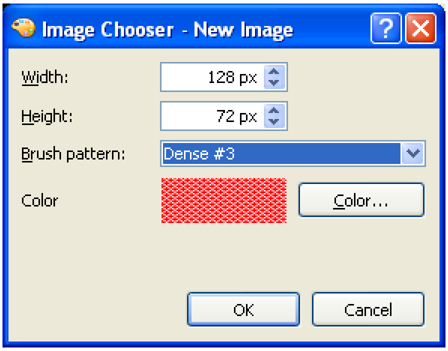
Handling File Actions

The File menu is probably the most widely implemented menu in main-window-style applications, and in most cases it offers, at the least, “new”, “save”, and “quit” (or “exit”) options.

```
    def fileNew(self):
        if not self.okToContinue():
            return
        dialog = newimagedlg.NewImageDlg(self)
        if dialog.exec_():
            self.addRecentFile(self.filename)
            self.image = QImage()
            for action, check in self.resetableActions:
                action.setChecked(check)
            self.image = dialog.image()
            self.filename = None
            self.dirty = True
            self.showImage()
            self.sizeLabel.setText("%d x %d" % (self.image.width(),
                                                self.image.height()))
            self.updateStatus("Created new image")
```

okToContinue() 186 ☞

When the user asks to work on a new file we begin by seeing whether it is “okay to continue”. This gives the user the chance to save or discard any unsaved changes, or to change their mind entirely and cancel the action.



Width: 128 px

Height: 72 px

Brush pattern: Dense #3

Color

Figure 6.9 *The New Image dialog*

If the user continues, we pop up a modal NewImageDlg in which they can specify the size, color, and brush pattern of the image they want to create. This dialog, shown in Figure 6.9, is created and used just like the dialogs we created in the preceding chapter. However, the New Image dialog’s user interface was created using *Qt Designer*, and the user interface file must be converted into a module file, using pyuic4, for the dialog to be usable. This can be done directly by running pyuic4, or by running either mkpyqt.py or Make PyQt, both of which are easier since they work out the correct command-line arguments automatically. We will cover all of these matters in the next chapter.

mkpyqt.py and Make PyQt sidebar ☞ 207

If the user accepts the dialog, we add the current filename (if any) to the recently used files list. Then we set the current image to be a null image, to ensure that any changes to checkable actions have no effect on the image. Next we go through the actions that we want to be reset when a new image is created or loaded, setting each one to our preferred default value. Now we can safely set the image to the one created by the dialog.

We set the filename to be None and the dirty flag to be True to ensure that the user will be prompted to save the image and asked for a filename, if they terminate the application or attempt to create or load another image.

We then call showImage() which displays the image in the imageLabel, scaled according to the zoom factor. Finally, we update the size label in the status bar, and call updateStatus().

```
def updateStatus(self, message):
    self.statusBar().showMessage(message, 5000)
    self.listWidget.addItem(message)
    if self.filename is not None:
        self.setWindowTitle("Image Changer - %s[*]" % \
                            os.path.basename(self.filename))
    elif not self.image.isNull():
        self.setWindowTitle("Image Changer - Unnamed[*]")
    else:
        self.setWindowTitle("Image Changer[*]")
    self.setWindowModified(self.dirty)
```

We begin by showing the message that has been passed, with a timeout of five seconds. We also add the message to the log widget to keep a log of every action that has taken place.

If the user has opened an existing file, or has saved the current file, we will have a filename. We put the filename in the window's title using Python's os.path.basename() function to get the filename without the path. We could just as easily have written QFileInfo(fname).fileName() instead, as we did earlier. If there is no filename and the image variable is not a null image, it means that the user has created a new image, but has not yet saved it; so we use a fake filename of "Unnamed". The last case is where no file has been opened or created.

Regardless of what we set the window title to be, we include the string "[*]" somewhere inside it. This string is never displayed as it is: Instead it is used to indicate whether the file is dirty. On Linux and Windows this means that the filename will be shown unadorned if it has no unsaved changes, and with an asterisk (*) replacing the `"[*]"` string otherwise. On Mac OS X, the close button will be shown with a dot in it if there are unsaved changes. The mechanism depends on the window modified status, so we make sure we set that to the state of the `dirty` flag.

```
    def fileOpen(self):
        if not self.okToContinue():
            return
        dir = os.path.dirname(self.filename) \
                if self.filename is not None else "."
        formats = ["*.%s" % unicode(format).lower() \
                   for format in QImageReader.supportedImageFormats()]
        fname = unicode(QFileDialog.getOpenFileName(self,
                        "Image Changer - Choose Image", dir,
                        "Image files (%s)" % " ".join(formats)))
        if fname:
            self.loadFile(fname)
```

If the user asks to open an existing image, we first make sure that they have had the chance to save or discard any unsaved changes, or to cancel the action entirely.

If the user has decided to continue, as a courtesy, we want to pop up a file open dialog set to a sensible directory. If we already have an image filename, we use its path; otherwise, we use "`.`", the current directory. We have also chosen to pass in a file filter string that limits the image file types the file open dialog can show. Such file types are defined by their extensions, and are passed as a string. The string may specify multiple extensions for a single type, and multiple types. For example, a text editor might pass a string of:

```
"Text files (*.txt)\nHTML files (*.htm *.html)"
```

If there is more than one type, we must separate them with newlines. If a type can handle more than one extension, we must separate the extensions with spaces. The string shown will produce a file type combobox with two items, "Text files" and "HTML files", and will ensure that the only file types shown in the dialog are those that have an extension of `.txt`, `.htm`, or `.html`.

In the case of the Image Changer application, we use the list of image type extensions for the image types that can be read by the version of PyQt that the application is using. At the very least, this is likely to include `.bmp`, `.jpg` (and `.jpeg`, the same as `.jpg`), and `.png`. The list comprehension iterates over the readable image extensions and creates a list of strings of the form "*.bmp", "*.jpg", and so on; these are joined, space-separated, into a single string by the string `join()` method.

List comprehensions 53 ☜

The QFileDialog.getOpenFileName() method returns a QString which either holds a filename (with the full path), or is empty (if the user canceled). If the user chose a filename, we call loadFile() to load it.

Here, and throughout the program, when we have needed the application's name we have simply written it. But since we set the name in the application object in main() to simplify our QSettings usage, we could instead retrieve the name whenever it was required. In this case, the relevant code would then become:

```
fname = unicode(QFileDialog.getOpenFileName(self,
                "%s - Choose Image" % QApplication.applicationName(),
                dir, "Image files (%s)" % " ".join(formats)))
```

It is surprising how frequently the name of the application is used. The file imagechanger.pyw is less than 500 lines, but it uses the application's name a dozen times. Some developers prefer to use the method call to guarantee consistency. We will discuss string handling further in Chapter 17, when we cover internationalization.

If the user opens a file, the loadFile() method is called to actually perform the loading. We will look at this method in two parts.

```
    def loadFile(self, fname=None):
        if fname is None:
            action = self.sender()
            if isinstance(action, QAction):
                fname = unicode(action.data().toString())
                if not self.okToContinue():
                    return
            else:
                return
```

If the method is called from the fileOpen() method or from the loadInitialFile() method, it is passed the filename to open. But if it is called from a recently used file action, no filename is passed. We can use this difference to distinguish the two cases. If a recently used file action was invoked, we retrieve the sending object. This should be a QAction, but we check to be safe, and then extract the action's user data, in which we stored the recently used file's full name including its path. User data is held as a QVariant, so we must convert it to a suitable type. At this point, we check to see whether it is okay to continue. We do not have to make this test in the "file open" case, because there, the check is made before the user is even asked for the name of a file to open. So now, if the method has not returned, we know that we have a filename in fname that we must try to load.

```
        if fname:
            self.filename = None
            image = QImage(fname)
        if image.isNull():
            message = "Failed to read %s" % fname
        else:
            self.addRecentFile(fname)
            self.image = QImage()
            for action, check in self.resetableActions:
                action.setChecked(check)
            self.image = image
            self.filename = fname
            self.showImage()
            self.dirty = False
            self.sizeLabel.setText("%d x %d" % (
                        image.width(), image.height()))
            message = "Loaded %s" % os.path.basename(fname)
        self.updateStatus(message)
```

addRecentFile()
189 ☜

We begin by making the current filename None and then we attempt to read the image into a local variable. PyQt does not use exception handling, so errors must always be discovered indirectly. In this case, a null image means that for some reason we failed to load the image. If the load was successful we add the new filename to the recently used files list, where it will appear only if another file is subsequently opened, or if this one is saved under another name. Next, we set the instance image variable to be a null image: This means that we are free to reset the checkable actions to our preferred defaults without any side effects. This works because when the checkable actions are changed, although the relevant methods will be called due to the signal–slot connections, the methods do nothing if the image is null.

After the preliminaries, we assign the local image to the image instance variable and the local filename to the filename instance variable. Next, we call showImage() to show the image at the current zoom factor, clear the dirty flag, and update the size label. Finally, we call updateStatus() to show the message in the status bar, and to update the log widget.

```
def fileSave(self):
    if self.image.isNull():
        return
    if self.filename is None:
        self.fileSaveAs()
    else:
        if self.image.save(self.filename, None):
            self.updateStatus("Saved as %s" % self.filename)
            self.dirty = False
        else:
            self.updateStatus("Failed to save %s" % self.filename)
```

The fileSave() method, and many others, act on the application's data (a QImage instance), but make no sense if there is no image data. For this reason, many of the methods do nothing and return immediately if there is no image data for them to work on.

If there is image data, and the filename is None, the user must have invoked the "file new" action, and is now saving their image for the first time. For this case, we pass on the work to the fileSaveAs() method.

If we have a filename, we attempt to save the image using QImage.save(). This method returns a Boolean success/failure flag, in response to which we update the status accordingly. (We have deferred coverage of loading and saving custom file formats to Chapter 8, since we are concentrating purely on main window functionality in this chapter.)

```
    def fileSaveAs(self):
        if self.image.isNull():
            return
        fname = self.filename if self.filename is not None else "."
        formats = ["*.%s" % unicode(format).lower() \
                   for format in QImageWriter.supportedImageFormats()]
        fname = unicode(QFileDialog.getSaveFileName(self,
                        "Image Changer - Save Image", fname,
                        "Image files (%s)" % " ".join(formats)))
        if fname:
            if "." not in fname:
                fname += ".png"
            self.addRecentFile(fname)
            self.filename = fname
            self.fileSave()
```

When the "file save as" action is triggered we begin by retrieving the current filename. If the filename is None, we set it to be ".", the current directory. We then use the QFileDialog.getSaveFileName() dialog to prompt the user to give us a filename to save under. If the current filename is not None, we use that as the default name—the file save dialog takes care of giving a warning yes/no dialog if the user chooses the name of a file that already exists. We use the same technique for setting the file filters string as we used for the "file open" action, but this time using the list of image formats that this version of PyQt can write (which may be different from the list of formats it can read).

If the user entered a filename that does not include a dot, that is, it has no extension, we set the extension to be .png. Next, we add the filename to the recently used files list (so that it will appear if a different file is subsequently opened, or if this one is saved under a new name), set the filename instance variable to the name, and pass the work of saving to the fileSave() method that we have just reviewed.

The last file action we must consider is "file print". When this action is invoked the filePrint() method is called. This method paints the image on a printer. Since the method uses techniques that we have not covered yet, we will defer discussion of it until later. The technique it uses is shown in the Printing Images sidebar, and coverage of the `filePrint()` method itself is in Chapter 13 (from page 400), where we also discuss approaches to printing documents in general.

Printing Images sidebar ☞ 363

The only file action we have not reviewed is the "file quit" action. This action is connected to the main window's `close()` method, which in turn causes a close event to be put on the event queue. We provided a reimplementation of the `closeEvent()` handler in which we made sure the user had the chance to save unsaved changes, using a call to `okToContinue()`, and where we saved the application's settings.

closeEvent() 185 ☜

Handling Edit Actions

Most of the functionality of the file actions was provided by the `MainWindow` subclass itself. The only work passed on was the image loading and saving, which the `QImage` instance variable was required to do. This particular division of responsibilities between a main window and the data structure that holds the data is very common. The main window handles the high-level file new, open, save, and recently used files functionality, and the data structure handles loading and saving.

It is also common for most, or even all, of the editing functionality to be provided either by the view widget or by the data structure. In the Image Changer application, all the data manipulation is handled by the data structure (the `image` `QImage`), and the presentation of the data is handled by the data viewer (the `imageLabel` `QLabel`). Again, this is a very common separation of responsibilities.

In this section, we will review most of the edit actions, omitting a couple that are almost identical to ones that are shown. We will be quite brief here, since the functionality is specific to the Image Changer application.

```
def editInvert(self, on):
    if self.image.isNull():
        return
    self.image.invertPixels()
    self.showImage()
    self.dirty = True
    self.updateStatus("Inverted" if on else "Uninverted")
```

If the user invokes the "edit invert" action, it will be checked (or unchecked). In either case, we simply invert the image's pixels using the functionality provided by `QImage`, show the changed image, set the dirty flag, and call `updateStatus()` so that the status bar briefly shows the action that was performed, and an additional item is added to the log.

The `editSwapRedAndBlue()` method (not shown) is the same except that it uses the `QImage.rgbSwapped()` method, and it has different status text.

```
    def editMirrorHorizontal(self, on):
        if self.image.isNull():
            return
        self.image = self.image.mirrored(True, False)
        self.showImage()
        self.mirroredhorizontally = not self.mirroredhorizontally
        self.dirty = True
        self.updateStatus("Mirrored Horizontally" \
                if on else "Unmirrored Horizontally")
```

This method is structurally the same as editInvert() and editSwapRedAndBlue(). The QImage.mirrored() method takes two Boolean flags, the first for horizontal mirroring and the second for vertical mirroring. In the Image Changer application, we have deliberately restricted what mirroring is allowed, so users can only have no mirroring, vertical mirroring, or horizontal mirroring, but not a combination of vertical and horizontal. We also keep an instance variable that keeps track of whether the image is horizontally mirrored.

The editMirrorVertical() method, not shown, is virtually identical.

```
    def editUnMirror(self, on):
        if self.image.isNull():
            return
        if self.mirroredhorizontally:
            self.editMirrorHorizontal(False)
        if self.mirroredvertically:
            self.editMirrorVertical(False)
```

This method switches off whichever mirroring is in force, or does nothing if the image is not mirrored. It does not set the dirty flag or update the status: It leaves that for editMirrorHorizontal() or editMirrorVertical(), if it calls either of them.

The application provides two means by which the user can change the zoom factor. They can interact with the zoom spinbox in the toolbar—its valueChanged() signal is connected to the showImage() slot that we will review shortly—or they can invoke the "edit zoom" action in the Edit menu. If they use the "edit zoom" action, the editZoom() method is called.

```
    def editZoom(self):
        if self.image.isNull():
            return
        percent, ok = QInputDialog.getInteger(self,
                "Image Changer - Zoom", "Percent:",
                self.zoomSpinBox.value(), 1, 400)
        if ok:
            self.zoomSpinBox.setValue(percent)
```

We begin by using one of the QInputDialog class's static methods to obtain a zoom factor. The getInteger() method takes a parent (over which the dialog will center itself), a caption, text describing what data is wanted, an initial value, and, optionally, minimum and maximum values.

The QInputDialog provides some other static convenience methods, including getDouble() to get a floating-point value, getItem() to choose a string from a list, and getText() to get a string. For all of them, the return value is a two-tuple, containing the value and a Boolean flag indicating whether the user entered and accepted a valid value.

If the user clicked OK, we set the zoom spinbox's value to the given integer. If this value is different from the current value, the spinbox will emit a valueChanged() signal. This signal is connected to the showImage() slot, so the slot will be called if the user chose a new zoom percentage value.

```
def showImage(self, percent=None):
    if self.image.isNull():
        return
    if percent is None:
        percent = self.zoomSpinBox.value()
    factor = percent / 100.0
    width = self.image.width() * factor
    height = self.image.height() * factor
    image = self.image.scaled(width, height, Qt.KeepAspectRatio)
    self.imageLabel.setPixmap(QPixmap.fromImage(image))
```

This slot is called when a new image is created or loaded, whenever a transformation is applied, and in response to the zoom spinbox's valueChanged() signal. This signal is emitted whenever the user changes the toolbar zoom spinbox's value, either directly using the mouse, or indirectly through the "edit zoom" action described earlier.

We retrieve the percentage and turn it into a zoom factor that we can use to produce the image's new width and height. We then create a *copy* of the image scaled to the new size and preserving the aspect ratio, and set the imageLabel to display this image. The label requires an image as a QPixmap, so we use the static QPixmap.fromImage() method to convert the QImage to a QPixmap.

Notice that zooming the image in this way has no effect on the original image; it is purely a change in view, not an edit. This is why the dirty flag does not need to be set.

According to PyQt's documentation, QPixmaps are optimized for on-screen display (so they are fast to draw), and QImages are optimized for editing (which is why we have used them to hold the image data).

Handling Help Actions

When we created the main window's actions, we provided each with help text, and set it as their status text and as their tooltip text. This means that when the user navigates the application's menu system, the status text of the currently highlighted menu option will automatically appear in the status bar. Similarly, if the user hovers the mouse over a toolbar button, the corresponding tooltip text will be displayed in a tooltip.

For an application as small and simple as the Image Changer, status tips and tooltips might be entirely adequate. Nonetheless, we have provided an online help system to show how it can be done, although we defer coverage until Chapter 17 (from page 510).

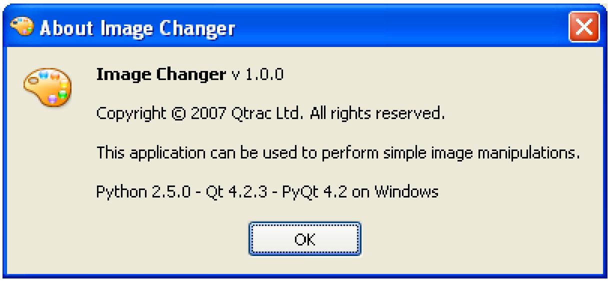


Image Changer v 1.0.0

Copyright © 2007 Qtrac Ltd. All rights reserved.

This application can be used to perform simple image manipulations.

Python 2.5.0 - Qt 4.2.3 - PyQt 4.2 on Windows

Figure 6.10 *The about Image Changer box*

Whether or not we provide online help, it is always a good idea to provide an "about" box. This should at least show the application's version and copyright notice, as Figure 6.10 illustrates.

```
    def helpAbout(self):
        QMessageBox.about(self, "About Image Changer",
                """<b>Image Changer</b> v %s
                <p>Copyright &copy; 2007 Qtrac Ltd.
                All rights reserved.
                <p>This application can be used to perform
                simple image manipulations.
                <p>Python %s - Qt %s - PyQt %s on %s""" % (
                __version__, platform.python_version(),
                QT_VERSION_STR, PYQT_VERSION_STR, platform.system()))
```

The QMessageBox.about() static convenience method pops up a modal OK-style message box with the given caption and text. The text can be HTML, as it is here. The message box will use the application's window icon if there is one.

We display the application's version, and version information about the Python, Qt, and PyQt libraries, as well as the platform the application is running on. The library version information is probably of no direct use to the user, but it may be very helpful to support staff who are being asked for help by the user.

Summary

Main-window-style applications are created by subclassing `QMainWindow`. The window has a single widget (which may be composite and so contain other widgets) as its central widget.

Actions are used to represent the functionality the application provides to its users. These actions are held as `QAction` objects which have text (used in menus), icons (used in both menus and toolbars), tooltips and status tips, and that are connected to slots, which, when invoked, will perform the appropriate action. Usually, all the actions are added to the main window's menus, and the most commonly used ones are added to toolbars. To support keyboard users, we provide keyboard shortcuts for frequently used actions, and menu accelerators to make menu navigation as quick and convenient as possible.

Some actions are checkable, and some groups of checkable actions may be mutually exclusive, that is, one and only one may be checked at any one time. PyQt supports checkable actions by the setting of a single property, and supports mutually exclusive groups of actions through `QActionGroup` objects.

Dock windows are represented by dock widgets and are easy to create and set up. Arbitrary widgets can be added to dock widgets and to toolbars, although in practice we only usually add small or letterbox-shaped widgets to toolbars.

Actions, action groups, and dock windows must all be given a parent explicitly—the main window, for example—to ensure that they are deleted at the right time. This is not necessary for the application's other widgets and `QObjects` because they are all owned either by the main window or by one of the main window's children. The application's non-`QObject` objects can be left to be deleted by Python's garbage collector.

Applications often use resources (small files, such as icons, and data files), and PyQt's resource mechanism makes accessing and using them quite easy. They do require an extra build step, though, either using PyQt's `pyrcc4` console application, or the `mkpyqt.py` or Make PyQt programs supplied with the book's examples.

Dialogs can be created entirely in code as we did in the preceding chapter, or using *Qt Designer*, as we will see in the next chapter. If we need to incorporate *Qt Designer* user interface files in our application, like resources they require an extra build step, either using PyQt's `pyuic4` console application, or again, using `mkpyqt.py` or Make PyQt.

Once the main window's visual appearance has been created by setting its central widget and by creating menus, toolbars, and perhaps dock windows, we can concern ourselves with loading and saving application settings. Many settings are commonly loaded in the main window's initializer, and settings are normally saved (and the user given the chance to save unsaved changes) in a reimplementation of the `closeEvent()` method.

If we want to restore the user's workspace, loading in the files they had open the last time they ran the application, it is best to use a single-shot timer at the end of the main window's initializer to load the files.

Most applications usually have a dataset and one or more widgets that are used to present and edit the data. Since the focus of the chapter has been on the main window's user interface infrastructure, we opted for the simplest possible data and visualization widget, but in later chapters the emphasis will be the other way around.

It is very common to have the main window take care of high-level file handling and the list of recently used files, and for the object holding the data to be responsible for loading, saving, and editing the data.

At this point in the book, you now know enough Python and PyQt to create both dialog-style and main-window-style GUI applications. In the next chapter, we will show *Qt Designer* in action, an application that can considerably speed up the development and maintenance of dialogs. And in the last chapter of Part II, we will explore some of the key approaches to saving and loading custom file formats, using both the PyQt and the Python libraries. In Parts III and IV, we will explore PyQt both more deeply, looking at event handling and creating custom widgets, for example, and more broadly, learning about PyQt's model/view architecture and other advanced features, including threading.

Exercise

Create the dialog shown in Figure 6.11. It should have the class name `ResizeDlg`, and its initializer should accept an initial width and height. The dialog should provide a method called `result()`, which must return a two-tuple of the width and height the user has chosen. The spinboxes should have a minimum of 4 and a maximum of four times the width (or height) passed in. Both should show their contents right-aligned.

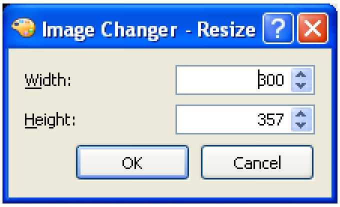


Image Changer - Resize

Width: 300

Height: 357

Figure 6.11 *The Image Changer resize dialog*

Modify the Image Changer application so that it has a new "edit resize" action. The action should appear on the Edit menu (after the "edit zoom" action). An icon called `editresize.png` is in the `images` subdirectory, but will need to be added to the `resources.qrc` file. You will also need to import the resize dialog you have just created.

The resize dialog should be used in an `editResize()` slot that the “edit resize” action should be connected to. The dialog is used like this:

```
form = resizedlg.ResizeDlg(self.image.width(),
                           self.image.height(), self)
if form.exec_():
    width, height = form.result()
```

Unlike the `editZoom()` slot, the image itself should be changed, so the size label, status bar, and dirty status must all be changed if the size is changed. On the other hand, if the “new” size is the same as the original size, no resizing should take place.

The resize dialog can be written in less than 50 lines, and the resize slot in less than 20 lines, with the new action just requiring an extra one or two lines in a couple of places in the main window’s initializer.

A model solution is in the files `chap06/imagechanger_ans.pyw` and `chap06/resizedlg.py`.