4

- A Pop-Up Alert in 25 Lines
- An Expression Evaluator in 30 Lines
- A Currency Converter in 70 Lines
- Signals and Slots

Introduction to GUI Programming

In this chapter we begin with brief reviews of three tiny yet useful GUI applications written in PyQt. We will take the opportunity to highlight some of the issues involved in GUI programming, but we will defer most of the details to later chapters. Once we have a feel for PyQt GUI programming, we will discuss PyQt's "signals and slots" mechanism—this is a high-level communication mechanism for responding to user interaction that allows us to ignore irrelevant detail.

Although PyQt is used commercially to build applications that vary in size from hundreds of lines of code to more than 100 000 lines of code, the applications we will build in this chapter are all less than 100 lines, and they show just how much can be done with very little code.

In this chapter we will design our user interfaces purely by writing code, but in Chapter 7, we will learn how to create user interfaces using Qt's visual design tool, *Qt Designer*.

Python console applications and Python module files always have a `.py` extension, but for Python GUI applications we use a `.pyw` extension. Both `.py` and `.pyw` are fine on Linux, but on Windows, `.pyw` ensures that Windows uses the `pythonw.exe` interpreter instead of `python.exe`, and this in turn ensures that when we execute a Python GUI application, no unnecessary console window will appear.* On Mac OS X, it is essential to use the `.pyw` extension.

The PyQt documentation is provided as a set of HTML files, independent of the Python documentation. The most commonly referred to documents are those covering the PyQt API. These files have been converted from the original C++/Qt documentation files, and their index page is called `classes.html`; Win-

*If you use Windows and an error message box titled, "pythonw.exe - Unable To Locate Component" pops up, it almost certainly means that you have not set your path correctly. See Appendix A, page 564, for how to fix this.

dows users will find a link to this page in their Start button's PyQt menu. It is well worth looking at this page to get an overview of what classes are available, and of course to dip in and read about those classes that seem interesting.

The first application we will look at is an unusual hybrid: a GUI application that must be launched from a console because it requires command-line arguments. We have included it because it makes it easier to explain how the PyQt event loop works (and what that is), without having to go into any other GUI details. The second and third examples are both very short but standard GUI applications. They both show the basics of how we can create and lay out widgets ("controls" in Windows-speak)—labels, buttons, comboboxes, and other on-screen elements that users can view and, in most cases, interact with. They also show how we can respond to user interactions—for example, how to call a particular function or method when the user performs a particular action.

In the last section we will cover how to handle user interactions in more depth, and in the next chapter we will cover layouts and dialogs much more thoroughly. Use this chapter to get a feel for how things work, without worrying about the details: The chapters that follow will fill in the gaps and will familiarize you with standard PyQt programming practices.

A Pop-Up Alert in 25 Lines

Our first GUI application is a bit odd. First, it must be run from the console, and second it has no "decorations"—no title bar, no system menu, no X close button. Figure 4.1 shows the whole thing.

Figure 4.1 *The Alert program*

To get the output displayed, we could enter a command line like this:

```
C:\>cd c:\pyqt\chap04
C:\pyqt\chap04>alert.pyw 12:15 Wake Up
```

When run, the program executes invisibly in the background, simply marking time until the specified time is reached. At that point, it pops up a window with the message text. About a minute after showing the window, the application will automatically terminate.

The specified time must use the 24-hour clock. For testing purposes we can use a time that has just gone; for example, by using 12:15 when it is really 12:30, the window will pop up immediately (well, within less than a second).

Now that we know what it does and how to run it, we will review the implementation. The file is a few lines longer than 25 lines because we have not counted

comment lines and blank lines in the total—but there are only 25 lines of executable code. We will begin with the imports.

```
import sys
import time
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

We import the `sys` module because we want to access the command-line arguments it holds in the `sys.argv` list. The `time` module is imported because we need its `sleep()` function, and we need the PyQt modules for the GUI and for the `QTime` class.

```
app = QApplication(sys.argv)
```

We begin by creating a `QApplication` object. Every PyQt GUI application must have a `QApplication` object. This object provides access to global-like information such as the application's directory, the screen size (and which screen the application is on, in a multihead system), and so on. This object also provides the *event loop*, discussed shortly.

When we create a `QApplication` object we pass it the command-line arguments; this is because PyQt recognizes some command-line arguments of its own, such as `-geometry` and `-style`, so we ought to give it the chance to read them. If `QApplication` recognizes any of the arguments, it acts on them, and removes them from the list it was given. The list of arguments that `QApplication` recognizes is given in the `QApplication`'s initializer's documentation.

```
try:
    due = QTime.currentTime()
    message = "Alert!"
    if len(sys.argv) < 2:
        raise ValueError
    hours, mins = sys.argv[1].split(":")
    due = QTime(int(hours), int(mins))
    if not due.isValid():
        raise ValueError
    if len(sys.argv) > 2:
        message = " ".join(sys.argv[2:])
except ValueError:
    message = "Usage: alert.pyw HH:MM [optional message]" # 24hr clock
```

At the very least, the application requires a time, so we set the `due` variable to the time right now. We also provide a default message. If the user has not given at least one command-line argument (a time), we raise a `ValueError` exception. This will result in the time being now and the message being the "usage" error message.

If the first argument does not contain a colon, a ValueError will be raised when we attempt to unpack two items from the split() call. If the hours or minutes are not a valid number, a ValueError will be raised by int(), and if the hours or minutes are out of range, due will be an invalid QTime, and we raise a ValueError ourselves. Although Python provides its own date and time classes, the PyQt date and time classes are often more convenient (and in some respects more powerful), so we tend to prefer them.

If the time is valid, we set the message to be the space-separated concatenation of the other command-line arguments if there are any; otherwise, we leave it as the default "Alert!" that we set at the beginning. (When a program is executed on the command line, it is given a list of arguments, the first being the invoking name, and the rest being each sequence of nonwhitespace characters, that is, each "word", entered on the command line. The words may be changed by the shell—for example, by applying wildcard expansion. Python puts the words it is actually given in the sys.argv list.)

Now we know when the message must be shown and what the message is.

```
while QTime.currentTime() < due:
    time.sleep(20) # 20 seconds
```

We loop continuously, comparing the current time with the target time. The loop will terminate if the current time is later than the target time. We could have simply put a pass statement inside the loop, but if we did that Python would loop as quickly as possible, gobbling up processor cycles for no good reason. The time.sleep() command tells Python to suspend processing for the specified number of seconds, 20 in this case. This gives other programs more opportunity to run and makes sense since we don't want to actually do anything while we wait for the due time to arrive.

Apart from creating the QApplication object, what we have done so far is standard console programming.

```
label = QLabel("<font color=red size=72><b>" + message + "</b></font>")
label.setWindowFlags(Qt.SplashScreen)
label.show()
QTimer.singleShot(60000, app.quit) # 1 minute
app.exec_()
```

We have created a QApplication object, we have a message, and the due time has arrived, so now we can begin to create our application. A GUI application needs widgets, and in this case we need a label to show the message. A QLabel can accept HTML text, so we give it an HTML string that tells it to display bold red text of size 72 points.*

*The supported HTML tags are listed at http://doc.trolltech.com/richtext-html-subset.html.

In PyQt, any widget can be used as a top-level window, even a button or a label. When a widget is used like this, PyQt automatically gives it a title bar. We don't want a title bar for this application, so we set the label's window flags to those used for splash screens since they have no title bar. Once we have set up the label that will be our window, we call `show()` on it. At this point, the label window is not shown! The call to `show()` merely schedules a "paint event", that is, it adds a new event to the `QApplication` object's event queue that is a request to paint the specified widget.

Next, we set up a single-shot timer. Whereas the Python library's `time.sleep()` function takes a number of seconds, the `QTimer.singleShot()` function takes a number of milliseconds. We give the `singleShot()` method two arguments: how long until it should time out (one minute in this case), and a function or method for it to call when it times out.

In PyQt terminology, the function or method we have given is called a "slot", although in the PyQt documentation the terms "callable", "Python slot", and "Qt slot" are used to distinguish slots from Python's `__slots__`, a feature of new-style classes that is described in the Python Language Reference. In this book we will use the PyQt terminology, since we never use `__slots__`.

Signals and slots ☞ 127

So now we have two events scheduled: A paint event that wants to take place immediately, and a timer timeout event that wants to take place in a minute's time.

The call to `app.exec_()` starts off the `QApplication` object's event loop.* The first event it gets is the paint event, so the label window pops up on-screen with the given message. About one minute later the timer timeout event occurs and the `QApplication.quit()` method is called. This method performs a clean termination of the GUI application. It closes any open windows, frees up any resources it has acquired, and exits.

Event loops are used by all GUI applications. In pseudocode, an event loop looks like this:

```
while True:
    event = getNextEvent()
    if event:
        if event == Terminate:
            break
        processEvent(event)
```

When the user interacts with the application, or when certain other things occur, such as a timer timing out or the application's window being uncovered (maybe because another application was closed), an event is generated inside PyQt and added to the event queue. The application's event loop continuously

*PyQt uses `exec_()` rather than `exec()` to avoid conflicting with Python's built-in `exec` statement.

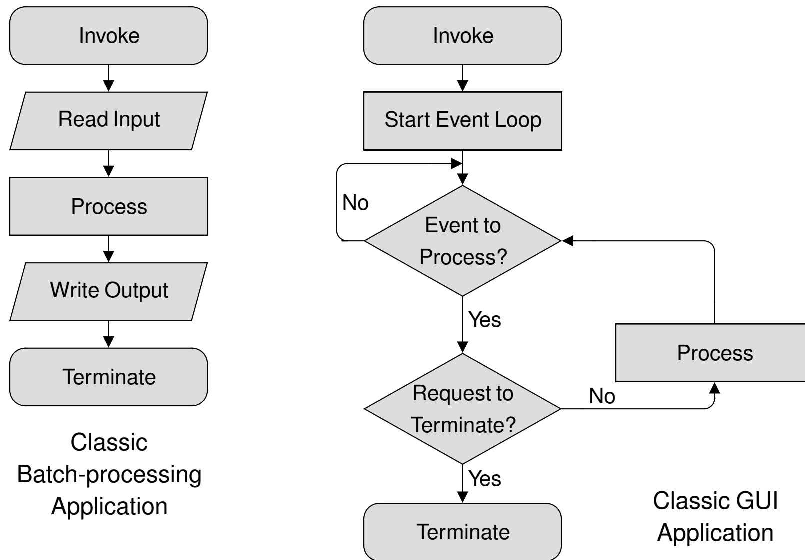


Figure 4.2 *Batch processing applications versus GUI applications*

checks to see whether there is an event to process, and if there is, it processes it (or passes it on to the event's associated function or method for processing).

Although complete, and quite useful if you use consoles, the application uses only a single widget. Also, we have not given it any ability to respond to user interaction. It also works rather like traditional batch-processing programs. It is invoked, performs some processing (waits, then shows a message), and terminates. Most GUI programs work differently. Once invoked, they run their event loop and respond to events. Some events come from the user—for example, key presses and mouse clicks—and some from the system, for example, timers timing out and windows being revealed. They process in response to requests that are the result of events such as button clicks and menu selections, and terminate only when told to do so.

The next application we will look at is much more conventional than the one we've just seen, and is typical of many very small GUI applications generally.

An Expression Evaluator in 30 Lines

This application is a complete dialog-style application written in 30 lines of code (excluding blank and comment lines). "Dialog-style" means an application that has no menu bar, and usually no toolbar or status bar, most commonly with some buttons (as we will see in the next section), and with no central widget. In contrast, "main window-style" applications normally have a menu bar, toolbars, a status bar, and in some cases buttons too; and they have a

central widget (which may contain other widgets, of course). We will look at main window-style applications in Chapter 6.

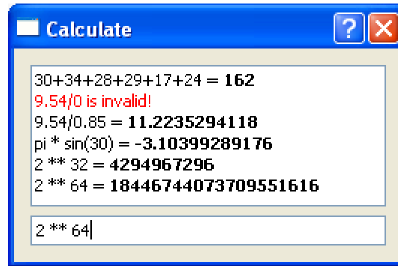


Figure 4.3 *The Calculate application*

This application uses two widgets: A `QTextBrowser` which is a read-only multi-line text box that can display both plain text and HTML; and a `QLineEdit`, which is a single-line text box that displays plain text. All text in PyQt widgets is Unicode, although it can be converted to other encodings when necessary.

The Calculate application (shown in Figure 4.3), can be invoked just like any normal GUI application by clicking (or double-clicking depending on platform and settings) its icon. (It can also be launched from a console, of course.) Once the application is running, the user can simply type mathematical expressions into the line edit and when they press Enter (or Return), the expression and its result are appended to the `QTextBrowser`. Any exceptions that are raised due to invalid expressions or invalid arithmetic (such as division by zero) are caught and turned into error messages that are simply appended to the `QTextBrowser`.

As usual, we will look at the code in sections. This example follows the pattern that we will use for all future GUI applications: A form is represented by a class, behavior in response to user interaction is handled by methods, and the "main" part of the program is tiny.

```
from __future__ import division
import sys
from math import *
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

Truncating division 17 ☞

Since we are doing mathematics and don't want any surprises like truncating division, we make sure we get floating-point division. Normally we import non-PyQt modules using the `import` *`moduleName`* syntax; but since we want all of the `math` module's functions and constants available to our program's users, we simply import them all into the current namespace. As usual, we import `sys` to get the `sys.argv` list, and we import everything from both the `QtCore` and the `QtGui` modules.

```
class Form(QDialog):
```

```
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        self.browser = QTextBrowser()
        self.lineedit = QLineEdit("Type an expression and press Enter")
        self.lineedit.selectAll()
        layout = QVBoxLayout()
        layout.addWidget(self.browser)
        layout.addWidget(self.lineedit)
        self.setLayout(layout)
        self.lineedit.setFocus()
        self.connect(self.lineedit, SIGNAL("returnPressed()"),
                     self.updateUi)
        self.setWindowTitle("Calculate")
```

As we have seen, any widget can be used as a top-level window. But in most cases when we create a top-level window we subclass `QDialog`, or `QMainWindow`, or occasionally, `QWidget`. Both `QDialog` and `QMainWindow`, and indeed all of PyQt's widgets, are derived from `QWidget`, and all are new-style classes. By inheriting `QDialog` we get a blank form, that is, a gray rectangle, and some convenient behaviors and methods. For example, if the user clicks the close `X` button, the dialog will close. By default, when a widget is closed it is merely hidden; we can, of course, change this behavior, as we will see in the next chapter.

We give our `Form` class's `__init__()` method a default parent of `None`, and use `super()` to initialize it. A widget that has no parent becomes a top-level window, which is what we want for our form. We then create the two widgets we need and keep references to them so that we can access them later, outside of `__init__()`. Since we did not give these widgets parents, it would seem that they will become top-level windows—which would not make sense. We will see shortly that they get parents later on in the initializer. We give the `QLineEdit` some initial text to show, and select it all. This will ensure that as soon as the user starts typing, the text we gave will be overwritten.

We want the widgets to appear vertically, one above the other, in the window. This is achieved by creating a `QVBoxLayout` and adding our two widgets to it, and then setting the layout on the form. If you run the application and resize it, you will find that any extra vertical space is given to the `QTextBrowser`, and that both widgets will grow horizontally. This is all handled automatically by the layout manager, and can be fine-tuned by setting layout policies.

One important side effect of using layouts is that PyQt automatically reparents the widgets that are laid out. So although we did not give our widgets a parent of `self` (the `Form` instance), when we call `setLayout()` the layout manager gives ownership of the widgets and of itself to the form, and takes ownership of any nested layouts itself. This means that none of the widgets that are laid out is a top-level window, and all of them have parents, which is what we want. So when the form is deleted, all its child widgets and layouts will be deleted with it, in the correct order.

Object Ownership

All PyQt classes that derive from QObject—and this includes all the widgets, since QWidget is a QObject subclass—can have a "parent". The parent–child relationship is used for two complementary purposes. A widget that has no parent is a top-level window, and a widget that has a parent (always another widget) is contained (displayed) within its parent. The relationship also defines *ownership*, with parents owning their children.

PyQt uses the parent–child ownership model to ensure that if a parent—for example, a top-level window—is deleted, all its children, for example, all the widgets the window contains, are automatically deleted as well. To avoid memory leaks, we should always make sure that any QObject, including all QWidgets, has a parent, the sole exception being top-level windows.

Most PyQt QObject subclasses have constructors that take a parent object as their last (optional) argument. But for widgets we generally do not (and need not) pass this argument. This is because widgets used in dialogs are laid out with layout managers, and when this occurs they are automatically reparented to the widget in which they are laid out, so they end up with the correct parent without requiring us to take any special action.

There are some cases where we must explicitly pass a parent—for example, when constructing QObject subclass objects that are not widgets, or that are widgets but which will not be laid out (such as dock widgets); we will see several examples of such cases in later chapters.

One final point is that it is possible to get situations where a Python variable is referring to an underlying PyQt object that no longer exists. This issue is covered in Chapter 9, in the "aliveness" discussion starting on page 287.

The widgets on a form can be laid out using a variety of techniques. We can use the resize() and move() methods to give them absolute sizes and positions; we can reimplement the resizeEvent() method and calculate their sizes and positions dynamically, or we can use PyQt's layout managers. Using absolute sizes and positions is very inconvenient. For one thing, we have to perform lots of manual calculations, and for another, if we change the layout we have to redo the calculations. Calculating the sizes and positions dynamically is a better approach, but still requires us to write quite a lot of tedious calculating code.

Using layout managers makes things a lot easier. And layout managers are quite smart: They automatically adapt to resize events and to content changes. Anyone used to dialogs in many versions of Windows will appreciate the benefits of having dialogs that can be resized (and that do so sensibly), rather than being forced to use small, nonresizable dialogs which can be very inconvenient when their contents are too large to fit. Layout managers also make life easier for internationalized programs since they adapt to content, so translated labels will not be "chopped off" if the target language is more verbose than the original language.

PyQt provides three layout managers: one for vertical layouts, one for horizontal layouts, and one for grid layouts. Layouts can be nested, so quite sophisticated layouts are possible. And there are other ways of laying out widgets, such as using splitters or tab widgets. All of these approaches are considered in more depth in Chapter 9.

As a courtesy to our users, we want the focus to start in the QLineEdit; we call setFocus() to achieve this. We must do this after setting the layout.

Signals and slots ☞ 127

The connect() call is something we will look at in depth later in this chapter. Suffice it to say that every widget (and some other QObjects) announce state changes by emitting “signals”. These signals (which are nothing to do with Unix signals) are usually ignored. However, we can choose to take notice of any signals we are interested in, and we do this by identifying the QObject that we want to know about, the signal it emits that we are interested in, and the function or method we want called when the signal is emitted.

So in this case, when the user presses Enter (or Return) in the QLineEdit, the returnPressed() signal will be emitted as usual, but because of our connect() call, when this occurs, our updateUi() method will be called. We will see what happens then in a moment.

The last thing we do in __init__() is set the window’s title.

As we will see shortly, the form is created and show() is called on it. Once the event loop begins, the form is shown—and nothing more appears to happen. The application is simply running the event loop, waiting for the user to click the mouse or press a key. Once the user starts interacting, the results of their interaction are processed. So if the user types in an expression, the QLineEdit will take care of displaying what they type, and if they press Enter, our updateUi() method will be called.

```
    def updateUi(self):
        try:
            text = unicode(self.lineedit.text())
            self.browser.append("%s = <b>%s</b>" % (text, eval(text)))
        except:
            self.browser.append(
                    "<font color=red>%s is invalid!</font>" % text)
```

PyQt string policy 28 ☜

When updateUi() is called it retrieves the text from the QLineEdit, immediately converting it to a unicode object. We then use Python’s eval() function to evaluate the string as an expression. If this is successful, we append a string to the QTextBrowser that has the expression text, an equals sign, and then the result in bold. Although we normally convert QStrings to unicode as soon as possible, we can pass QStrings, unicodes, and strs to PyQt methods that expect a QString, and PyQt will automatically perform any necessary conversion. If any exception occurs, we append an error message instead. Using a catch-all except block like this is not good general practice, but for a 30-line program it seems reasonable.

By using `eval()` we avoid all the work of parsing and error checking that we would have to do ourselves if we were using a compiled language.

```
app = QApplication(sys.argv)
form = Form()
form.show()
app.exec_()
```

Now that we have defined our `Form` class, at the end of the `calculate.pyw` file, we create the `QApplication` object, instantiate an instance of our form, schedule it to be painted, and start off the event loop.

And that is the complete application. But it isn't quite the end of the story. We have not said how the user can terminate the application. Because our form derives from `QDialog`, it inherits some behavior. For example, if the user clicks the close button X, or if they press the Esc key, the form will close. When a form closes, it is hidden. When the form is hidden PyQt will detect that the application has no visible windows and that no further interaction is possible. It will therefore delete the form and perform a clean termination of the application.

In some cases, we want an application to continue even if it is not visible—for example, a server. For these cases, we can call `QApplication.setQuitOnLastWindowClosed(False)`. It is also possible, although rarely necessary, to be notified when the last window is closed.

On Mac OS X, and some X Windows window managers, like twm, an application like this will *not* have a close button, and on the Mac, choosing Quit on the menu bar will not work. In such cases, pressing Esc will terminate the application, and in addition on the Mac, Command+. will also work. In view of this, for applications that are likely to be used on the Mac or with twm or similar, it is best to provide a Quit button. Adding buttons to dialogs is covered in this chapter's last section.

We are now ready to look at the last small, complete example that we will present in this chapter. It has more custom behavior, has a more complex layout, and does more sophisticated processing, but its fundamental structure is very similar to the Calculate application, and indeed to that of many other PyQt dialogs.

A Currency Converter in 70 Lines

One small utility that is often useful is a currency converter. But since exchange rates frequently change, we cannot simply create a static dictionary of conversion rates as we did for the units of length in the `Length` class we created in the previous chapter. Fortunately, the Bank of Canada provides exchange rates in a file that is accessible over the Internet, and which uses an easy-to-parse format. The rates are sometimes a few days old, but they are good

enough for estimating the cash required for trips or how much a foreign contract is likely to pay. The application is shown in Figure 4.4.

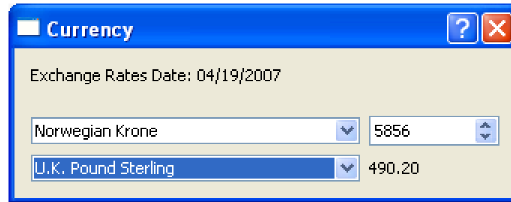


Figure 4.4 *The Currency application*

The application must first download and parse the exchange rates. Then it must create a user interface which the user can manipulate to specify the currencies and the amount that they are interested in.

As usual, we will begin with the imports:

```
import sys
import urllib2
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

Both Python and PyQt provide classes for networking. In Chapter 18, we will use PyQt's classes, but here we will use Python's `urllib2` module because it provides a very useful convenience function that makes it easy to grab a file over the Internet.

```
class Form(QDialog):

    def __init__(self, parent=None):
        super(Form, self).__init__(parent)

        date = self.getdata()
        rates = sorted(self.rates.keys())

        dateLabel = QLabel(date)
        self.fromComboBox = QComboBox()
        self.fromComboBox.addItems(rates)
        self.fromSpinBox = QDoubleSpinBox()
        self.fromSpinBox.setRange(0.01, 10000000.00)
        self.fromSpinBox.setValue(1.00)
        self.toComboBox = QComboBox()
        self.toComboBox.addItems(rates)
        self.toLabel = QLabel("1.00")
```

After initializing our form using `super()`, we call our `getdata()` method. As we will soon see, this method gets the exchange rates, populates the `self.rates` dictionary, and returns a string holding the date the rates were in force. The dictionary's keys are currency names, and the values are the conversion factors.

We take a sorted copy of the dictionary's keys so that we can present the user with sorted lists of currencies in the comboboxes. The `date` and `rates` variables, and the `dateLabel`, are referred to only inside `__init__()`, so we do not keep references to them in the class instance. On the other hand, we do need to access the comboboxes and the `toLabel` (which displays the amount of the target currency), so we make these instance variables by using `self`.

Instance variables 77 ☜

We add the same sorted list of currencies to both comboboxes, and we create a `QDoubleSpinBox`, a spinbox that handles floating-point values. We provide a minimum and maximum value for the spinbox, and also an initial value. It is good practice to always set a spinbox's range before setting its value, since if we set the value first and this happens to be outside the default range, the value will be reduced or increased to fit the default range.

Since both comboboxes will initially show the same currency and the initial value to convert is 1.00, the result shown in the `toLabel` must also be 1.00, so we set this explicitly.

```
grid = QGridLayout()
grid.addWidget(dateLabel, 0, 0)
grid.addWidget(self.fromComboBox, 1, 0)
grid.addWidget(self.fromSpinBox, 1, 1)
grid.addWidget(self.toComboBox, 2, 0)
grid.addWidget(self.toLabel, 2, 1)
self.setLayout(grid)
```

A grid layout seems to be the simplest solution to laying out the widgets. When we add a widget to a grid we give the row and column position it should occupy, both of which are 0-based. The layout is shown schematically in Figure 4.5. Much more can be done with grid layouts. For example, we can have spanning rows and columns; all of this is covered later, in Chapter 9.

`dateLabel`	(0, 0)		
`self.fromComboBox`	(1, 0)	`self.fromSpinBox`	(1, 1)
`self.toComboBox`	(2, 0)	`self.toLabel`	(2, 1)

Figure 4.5 *The Currency application's grid layout*

If we look at the screenshot, or run the application, it is clear that column 0 of the grid layout is much wider than column 1. But there is nothing in the code that specifies this, so why does it happen? Layouts are smart enough to adapt to their environment, both to the space available and to the contents and size policies of the widgets they are managing. In this case, the comboboxes are stretched horizontally to be wide enough to show the widest currency text in full, and the spinbox is stretched horizontally to be wide enough to show its maximum value. Since comboboxes are the widest items in column 0, they effectively set that column's minimum width; and similarly for the spinbox

in column 1. If we run the application and try to make the window narrower, nothing will happen because it is already at its minimum width. But we can make the window wider and both columns will stretch to occupy the extra space. It is, of course, possible to bias the layout so that it gives more horizontal space to, say, column 0, when extra space is available.

None of the widgets is initially stretched vertically because that is not necessary for any of them. But if we increase the window's height, all of the extra space will go to the `dateLabel` because that is the only widget on the form that likes to grow in every direction and has no other widgets to constrain it.

Now that we have created, populated, and laid out the widgets, it is time to set up the form's behavior.

```
        self.connect(self.fromComboBox,
                SIGNAL("currentIndexChanged(int)"), self.updateUi)
        self.connect(self.toComboBox,
                SIGNAL("currentIndexChanged(int)"), self.updateUi)
        self.connect(self.fromSpinBox,
                SIGNAL("valueChanged(double)"), self.updateUi)
        self.setWindowTitle("Currency")
```

If the user changes the current item in one of the comboboxes, the relevant combobox will emit a `currentIndexChanged()` signal with the index position of the new current item. Similarly, if the user changes the value held by the spinbox, a `valueChanged()` signal will be emitted with the new value. We have connected all these signals to just one Python slot: `updateUi()`. This does not have to be the case, as we will see in the next section, but it happens to be a sensible choice for this application.

And at the end of `__init__()` we set the window's title.

```
    def updateUi(self):
        to = unicode(self.toComboBox.currentText())
        from_ = unicode(self.fromComboBox.currentText())
        amount = (self.rates[from_] / self.rates[to]) * \
                 self.fromSpinBox.value()
        self.toLabel.setText("%0.2f" % amount)
```

This method is called in response to the `currentIndexChanged()` signal emitted by the comboboxes, and in response to the `valueChanged()` signal emitted by the spinbox. All the signals involved also pass a parameter. As we will see in the next section, we can ignore signal parameters, as we do here.

No matter which signal was involved, we go through the same process. We extract the "to" and "from" currencies, calculate the "to" amount, and set the `toLabel`'s text accordingly. We have given the "from" text's variable the name `from_` because `from` is a Python keyword and therefore not available to us. We had to escape a newline when calculating the amount to make the line narrow

enough to fit on the page; and in any case, we prefer to limit line lengths to make it easier to read two files side by side on the screen.

```
    def getdata(self): # Idea taken from the Python Cookbook
        self.rates = {}
        try:
            date = "Unknown"
            fh = urllib2.urlopen("http://www.bankofcanada.ca"
                                 "/en/markets/csv/exchange_eng.csv")
            for line in fh:
                if not line or line.startswith(("#", "Closing ")):
                    continue
                fields = line.split(",")
                if line.startswith("Date "):
                    date = fields[-1]
                else:
                    try:
                        value = float(fields[-1])
                        self.rates[unicode(fields[0])] = value
                    except ValueError:
                        pass
            return "Exchange Rates Date: " + date
        except Exception, e:
            return "Failed to download:\n%s" % e
```

This method is where we get the data that drives the application. We begin by creating a new instance attribute, `self.rates`. Unlike C++, Java, and similar languages, Python allows us to create instance attributes as and when we need them—for example, in the constructor, in the initializer, or in any other method. We can even add attributes to specific instances on the fly.

Since a lot can go wrong with network connections—for example, the network might be down, the host might be down, the URL may have changed, and so on, we need to make the application more robust than in the previous two examples. Another possible problem is that we may get an invalid floating-point value such as the "NA" (Not Available) that the currency data sometimes contains. We have an inner `try` ... `except` block that catches invalid numbers. So if we fail to convert a currency value to a floating-point number, we simply skip that particular currency and continue.

We handle every other possibility by wrapping almost the entire method in an outer `try` ... `except` block. (This is too general for most applications, but it seems acceptable for a tiny 70-line application.) If a problem occurs, we catch the exception raised and return it as a string to the caller, `__init__()`. The string that is returned by `getdata()` is shown in the `dateLabel`, so normally this label will show the date applicable to the exchange rates, but in an error situation it will show the error message instead.

Notice that we have split the URL string into two strings over two lines because it is so long—and we did not need to escape the newline. This works because the strings are within parentheses. If that wasn't the case, we would either have to escape the newline or concatenate them using + (and still escape the newline).

We initialize the `date` variable with a string indicating that we don't know what dates the rates were calculated. We then use the `urllib2.urlopen()` function to give us a file handle to the file we are interested in. The file handle can be used to read the entire file in one go using its `read()` method, but in this case we prefer to read line by line using `readlines()`.

Here is the data from the `exchange_eng.csv` file on one particular day. Some columns, and most rows, have been omitted; these are indicated by ellipses.

```
...
#
Date (<m>/<d>/<year>),01/05/2007,...,01/12/2007,01/15/2007
Closing Can/US Exchange Rate,1.1725,...,1.1688,1.1667
U.S. Dollar (Noon),1.1755,...,1.1702,1.1681
Argentina Peso (Floating Rate),0.3797,...,0.3773,0.3767
Australian Dollar,0.9164,...,0.9157,0.9153
...
Vietnamese Dong,0.000073,...,0.000073,0.000073
```

The `exchange_eng.csv` file's format uses several different kinds of lines. Comment lines begin with "#", and there may also be blank lines; we ignore all these. The exchange rates are listed by name, followed by rates, all comma-separated. The rates are those applying on particular dates, with the last one on each line being the most recent. We split each of these lines on commas and take the first item to be the currency name, and the last item to be the exchange rate. There is also a line that begins with "Date"; this lists the dates applying to each column. When we encounter this line we take the last date, since that is the one that corresponds with the exchange rates we are using. There is also a line that begins "Closing"; we ignore it.

For each exchange rate line we insert an item into the `self.rates` dictionary, using the currency's name for the key and the exchange rate as the value. We have assumed that the file's text is either 7-bit ASCII or Unicode; if it isn't one of these we may get an encoding error. If we knew the encoding, we could specify it as a second argument when we call `unicode()`.

```
app = QApplication(sys.argv)
form = Form()
form.show()
app.exec_()
```

We have used exactly the same code as the previous example to create the `QApplication` object, instantiate the Currency application's form, and start off the event loop.

As for program termination, just like the previous example, because we have subclassed `QDialog`, if the user clicks the close X button or presses Esc, the window will close and then PyQt will terminate the application. In Chapter 6, we will see how to provide more explicit means of termination, and how to ensure that the user has the opportunity to save any unsaved changes and program settings.

By now it should be clear that using PyQt for GUI programming is straightforward. Although we will see more complex layouts later on, they are not intrinsically difficult, and because the layout managers are smart, in most cases they "just work". Naturally, there is a lot more to be covered—for example, creating main window-style applications, creating dialogs that the user can pop-up for interaction, and so on. But we will begin with something fundamental to PyQt, that so far we have glossed over: the signals and slots communication mechanism, which is the subject of the next section.

Signals and Slots

Every GUI library provides the details of events that take place, such as mouse clicks and key presses. For example, if we have a button with the text Click Me, and the user clicks it, all kinds of information becomes available. The GUI library can tell us the coordinates of the mouse click relative to the button, relative to the button's parent widget, and relative to the screen; it can tell us the state of the Shift, Ctrl, Alt, and NumLock keys at the time of the click; and the precise time of the click and of the release; and so on. Similar information can be provided if the user "clicked" the button without using the mouse. The user may have pressed the Tab key enough times to move the focus to the button and then pressed Spacebar, or maybe they pressed Alt+C. Although the outcome is the same in all these cases, each different means of clicking the button produces different events and different information.

The Qt library was the first to recognize that in almost every case, programmers don't need or even want all the low-level details: They don't care *how* the button was pressed, they just want to know *that* it was pressed so that they can respond appropriately. For this reason Qt, and therefore PyQt, provides two communication mechanisms: a low-level event-handling mechanism which is similar to those provided by all the other GUI libraries, and a high-level mechanism which Trolltech (makers of Qt) have called "signals and slots". We will look at the low-level mechanism in Chapter 10, and again in Chapter 11, but in this section we will focus on the high-level mechanism.

Every `QObject`—including all of PyQt's widgets since they derive from `QWidget`, a `QObject` subclass—supports the signals and slots mechanism. In particular, they are capable of announcing state changes, such as when a checkbox

becomes checked or unchecked, and other important occurrences, for example when a button is clicked (by whatever means). All of PyQt's widgets have a set of predefined signals.

Whenever a signal is emitted, by default PyQt simply throws it away! To take notice of a signal we must connect it to a *slot*. In C++/Qt, slots are methods that must be declared with a special syntax; but in PyQt, they can be any callable we like (e.g., any function or method), and no special syntax is required when defining them.

Most widgets also have predefined slots, so in some cases we can connect a predefined signal to a predefined slot and not have to do anything else to get the behavior we want. PyQt is more versatile than C++/Qt in this regard, because we can connect not just to slots, but also to any callable, and from PyQt 4.2, it is possible to dynamically add "predefined" signals and slots to `QObjects`. Let's see how signals and slots works in practice with the Signals and Slots program shown in Figure 4.6.

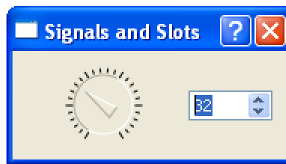


Figure 4.6 *The Signals and Slots program*

Both the `QDial` and `QSpinBox` widgets have `valueChanged()` signals that, when emitted, carry the new value. And they both have `setValue()` slots that take an integer value. We can therefore connect these two widgets to each other so that whichever one the user changes, will cause the other to be changed correspondingly:

```
class Form(QDialog):

    def __init__(self, parent=None):
        super(Form, self).__init__(parent)

        dial = QDial()
        dial.setNotchesVisible(True)
        spinbox = QSpinBox()

        layout = QHBoxLayout()
        layout.addWidget(dial)
        layout.addWidget(spinbox)
        self.setLayout(layout)

        self.connect(dial, SIGNAL("valueChanged(int)"),
                     spinbox.setValue)
        self.connect(spinbox, SIGNAL("valueChanged(int)"),
                     dial.setValue)
```

```
        self.setWindowTitle("Signals and Slots")
```

Since the two widgets are connected in this way, if the user moves the dial—say to value 20—the dial will emit a `valueChanged(20)` signal which will, in turn, cause a call to the spinbox's `setValue()` slot with 20 as the argument. But then, since its value has now been changed, the spinbox will emit a `valueChanged(20)` signal which will in turn cause a call to the dial's `setValue()` slot with 20 as the argument. So it looks like we will get an infinite loop. But what happens is that the `valueChanged()` signal is not emitted if the value is not actually changed. This is because the standard approach to writing value-changing slots is to begin by comparing the new value with the existing one. If the values are the same, we do nothing and return; otherwise, we apply the change and emit a signal to announce the change of state. The connections are depicted in Figure 4.7.

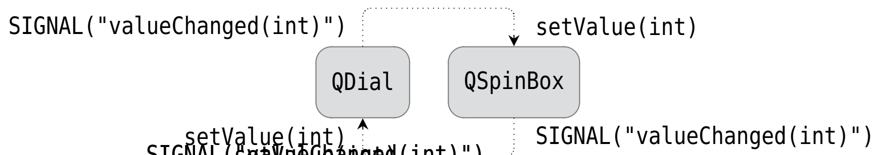


Figure 4.7 *The signals and slots connections*

Now let's look at the general syntax for connections. We assume that the PyQt modules have been imported using the `from` … `import *` syntax, and that *s* and *w* are `QObjects`, normally widgets, with *s* usually being `self`.

```
s.connect(w, SIGNAL("signalSignature"), functionName)
s.connect(w, SIGNAL("signalSignature"), instance.methodName)
s.connect(w, SIGNAL("signalSignature"),
          instance, SLOT("slotSignature"))
```

The *signalSignature* is the name of the signal and a (possibly empty) comma-separated list of parameter type names in parentheses. If the signal is a Qt signal, the type names must be the C++ type names, such as `int` and `QString`. C++ type names can be rather complex, with each type name possibly including one or more of `const`, `*`, and `&`. When we write them as signal (or slot) signatures we can drop any `consts` and `&s`, but must keep any `*s`. For example, almost every Qt signal that passes a `QString` uses a parameter type of `const QString&`, but in PyQt, just using `QString` alone is sufficient. On the other hand, the `QListWidget` has a signal with the signature `itemActivated(QListWidgetItem*)`, and we must use this exactly as written.

PyQt signals are defined when they are actually emitted and can have any number of any type of parameters, as we will see shortly.

The *slotSignature* has the same form as a *signalSignature* except that the name is of a Qt slot. A slot may not have more arguments than the signal that is connected to it, but may have less; the additional parameters are then

discarded. Corresponding signal and slot arguments must have the same types, so for example, we could not connect a `QDial`'s `valueChanged(int)` signal to a `QLineEdit`'s `setText(QString)` slot.

In our dial and spinbox example we used the *`instance.methodName`* syntax as we did with the example applications shown earlier in the chapter. But when the slot is actually a Qt slot rather than a Python method, it is more efficient to use the `SLOT()` syntax:

```
self.connect(dial, SIGNAL("valueChanged(int)"),
             spinbox, SLOT("setValue(int)"))
self.connect(spinbox, SIGNAL("valueChanged(int)"),
             dial, SLOT("setValue(int)"))
```

We have already seen that it is possible to connect multiple signals to the same slot. It is also possible to connect a single signal to multiple slots. Although rare, we can also connect a signal to another signal: In such cases, when the first signal is emitted, it will cause the signal it is connected to, to be emitted.

Connections are made using `QObject.connect()`; they can be broken using `QObject.disconnect()`. In practice, we rarely need to break connections ourselves since, for example, PyQt will automatically disconnect any connections involving an object that has been deleted.

So far we have seen how to connect to signals, and how to write slots—which are ordinary functions or methods. And we know that signals are emitted to signify state changes or other important occurrences. But what if we want to create a component that emits its own signals? This is easily achieved using `QObject.emit()`. For example, here is a complete `QSpinBox` subclass that emits its own custom `atzero` signal, and that also passes a number:

```
class ZeroSpinBox(QSpinBox):

    zeros = 0

    def __init__(self, parent=None):
        super(ZeroSpinBox, self).__init__(parent)
        self.connect(self, SIGNAL("valueChanged(int)"), self.checkzero)


    def checkzero(self):
        if self.value() == 0:
            self.zeros += 1
            self.emit(SIGNAL("atzero"), self.zeros)
```

We connect to the spinbox's own `valueChanged()` signal and have it call our `checkzero()` slot. If the value happens to be 0, the `checkzero()` slot emits the `atzero` signal, along with a count of how many times it has been zero; passing additional data like this is optional. The lack of parentheses for the signal is important: It tells PyQt that this is a "short-circuit" signal.

A signal with no arguments (and therefore no parentheses) is a short-circuit Python signal. When such a signal is emitted, any data can be passed as additional arguments to the `emit()` method, and they are passed as Python objects. This avoids the overhead of converting the arguments to and from C++ data types, and also means that arbitrary Python objects can be passed, even ones which cannot be converted to and from C++ data types. A signal with at least one argument is either a Qt signal or a non-short-circuit Python signal. In these cases, PyQt will check to see whether the signal is a Qt signal, and if it is not will assume that it is a Python signal. In either case, the arguments are converted to C++ data types.

Here is how we connect to the signal in the form's `__init__()` method:

```
        zerospinbox = ZeroSpinBox()
        ...
        self.connect(zerospinbox, SIGNAL("atzero"), self.announce)
```

Again, we must not use parentheses because it is a short-circuit signal. And for completeness, here is the slot it connects to in the form:

```
    def announce(self, zeros):
        print "ZeroSpinBox has been at zero %d times" % zeros
```

If we use the `SIGNAL()` function with an identifier but no parentheses, we are specifying a short-circuit signal as described earlier. We can use this syntax both to emit short-circuit signals, and to connect to them. Both uses are shown in the example.

If we use the `SIGNAL()` function with a *`signalSignature`* (a possibly empty parenthesized list of comma-separated PyQt types), we are specifying either a Python or a Qt signal. (A Python signal is one that is emitted in Python code; a Qt signal is one emitted from an underlying C++ object.) We can use this syntax both to emit Python and Qt signals, and to connect to them. These signals can be connected to any callable, that is, to any function or method, including Qt slots; they can also be connected using the `SLOT()` syntax, with a *`slotSignature`*. PyQt checks to see whether the signal is a Qt signal, and if it is not it assumes it is a Python signal. If we use parentheses, even for Python signals, the arguments must be convertible to C++ data types.

We will now look at another example, a tiny custom non-GUI class that has a signal and a slot and which shows that the mechanism is not limited to GUI classes—any `QObject` subclass can use signals and slots.

```
class TaxRate(QObject):

    def __init__(self):
        super(TaxRate, self).__init__()
        self.__rate = 17.5
```

```
    def rate(self):
        return self.__rate

    def setRate(self, rate):
        if rate != self.__rate:
            self.__rate = rate
            self.emit(SIGNAL("rateChanged"), self.__rate)
```

Both the rate() and the setRate() methods can be connected to, since any Python callable can be used as a slot. If the rate is changed, we update the private __rate value and emit a custom rateChanged signal, giving the new rate as a parameter. We have also used the faster short-circuit syntax. If we wanted to use the standard syntax, the only difference would be that the signal would be written as SIGNAL("rateChanged(float)"). If we connect the rateChanged signal to the setRate() slot, because of the if statement, no infinite loop will occur. Let us look at the class in use. First we will declare a function to be called when the rate changes:

```
def rateChanged(value):
    print "TaxRate changed to %.2f%%" % value
```

And now we will try it out:

```
vat = TaxRate()
vat.connect(vat, SIGNAL("rateChanged"), rateChanged)
vat.setRate(17.5)    # No change will occur (new rate is the same)
vat.setRate(8.5)     # A change will occur (new rate is different)
```

This will cause just one line to be output to the console: "TaxRate changed to 8.50%".

In earlier examples where we connected multiple signals to the same slot, we did not care who emitted the signal. But sometimes we want to connect two or more signals to the same slot, and have the slot behave differently depending on who called it. In this section's last example we will address this issue.

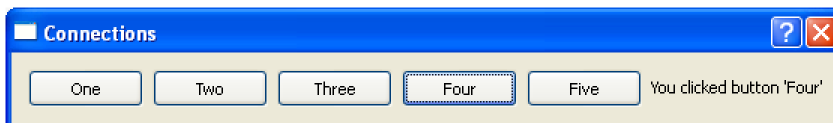


Figure 4.8 *The Connections program*

The Connections program shown in Figure 4.8, has five buttons and a label. When one of the buttons is clicked the signals and slots mechanism is used to update the label's text. Here is how the first button is created in the form's __init__() method:

```
        button1 = QPushButton("One")
```

All the other buttons are created in the same way, differing only in their variable name and the text that is passed to them.

We will start with the simplest connection, which is used by `button1`. Here is the `__init__()` method's `connect()` call:

```
self.connect(button1, SIGNAL("clicked()"), self.one)
```

We have used a dedicated method for this button:

```
def one(self):
    self.label.setText("You clicked button 'One'")
```

Connecting a button's `clicked()` signal to a single method that responds appropriately is probably the most common connection scenario.

Partial function application

63 ☜

But what if most of the processing was the same, with just some parameterization depending on which particular button was pressed? In such cases, it is usually best to connect each button to the same slot. There are two approaches to doing this. One is to use partial function application to wrap a slot with a parameter so that when the slot is invoked it is parameterized with the button that called it. The other is to ask PyQt to tell us which button called the slot. We will show both approaches, starting with partial function application.

Back on page 65 we created a wrapper function which used Python 2.5's `functools.partial()` function or our own simple `partial()` function:

```
import sys

if sys.version_info[:2] < (2, 5):
    def partial(func, arg):
        def callme():
            return func(arg)
        return callme
else:
    from functools import partial
```

Using `partial()` we can now wrap a slot and a button name together. So we might be tempted to do this:

```
self.connect(button2, SIGNAL("clicked()"),
             partial(self.anyButton, "Two")) # WRONG for PyQt 4.0-4.2
```

Unfortunately, this won't work for PyQt versions prior to 4.3. The wrapper function is created in the `connect()` call, but as soon as the `connect()` call completes, the wrapper goes out of scope and is garbage-collected. From PyQt 4.3, wrappers made with `functools.partial()` are treated specially when they are used for connections like this. This means that the function connected to will not be garbage-collected, so the code shown earlier will work correctly.

PyQt 4.3

For PyQt 4.0, 4.1, and 4.2, we can still use `partial()`: We just need to keep a reference to the wrapper—we will not use the reference except for the `connect()` call, but the fact that it is an attribute of the form instance will ensure that the wrapper function will not go out of scope while the form exists, and will therefore work. So the connection is actually made like this:

```
self.button2callback = partial(self.anyButton, "Two")
self.connect(button2, SIGNAL("clicked()"),
             self.button2callback)
```

When `button2` is clicked, the `anyButton()` method will be called with a string parameter containing the text "Two". Here is what this method looks like:

```
def anyButton(self, who):
    self.label.setText("You clicked button '%s'" % who)
```

We could have used this slot for all the buttons using the `partial()` function that we have just shown. And in fact, we could avoid using `partial()` at all and get the same results:

```
self.button3callback = lambda who="Three": self.anyButton(who)
self.connect(button3, SIGNAL("clicked()"),
             self.button3callback)
```

Lambda functions 61 ☜

Here we've created a `lambda` function that is parameterized by the button's name. It works the same as the `partial()` technique, and calls the same `anyButton()` method, only with `lambda` being used to create the wrapper.

Both `button2callback()` and `button3callback()` call `anyButton()`; the only difference between them is that the first passes "Two" as its parameter and the second passes "Three".

PyQt 4.1.1

If we are using PyQt 4.1.1 or later, and we use `lambda` callbacks, we don't have to keep a reference to them ourselves. This is because PyQt treats `lambda` specially when used to create wrappers in a connection. (This is the same special treatment that is expected to be extended to `functools.partial()` in PyQt 4.3.) For this reason we can use `lambda` directly in `connect()` calls. For example:

```
self.connect(button3, SIGNAL("clicked()"),
             lambda who="Three": self.anyButton(who))
```

The wrapping technique works perfectly well, but there is an alternative approach that is slightly more involved, but which may be useful in some cases, particularly when we don't want to wrap our calls. This other technique is used to respond to `button4` and to `button5`. Here are their connections:

```
self.connect(button4, SIGNAL("clicked()"), self.clicked)
self.connect(button5, SIGNAL("clicked()"), self.clicked)
```

Notice that we do not wrap the clicked() method that they are both connected to, so at first sight it looks like there is no way to tell which button called the clicked() method.* However, the implementation makes clear that we can distinguish if we want to:

```
def clicked(self):
    button = self.sender()
    if button is None or not isinstance(button, QPushButton):
        return
    self.label.setText("You clicked button '%s'" % button.text())
```

Inside a slot we can always call sender() to discover which QObject the invoking signal came from. (This could be None if the slot was called using a normal method call.) Although we know that we have connected only buttons to this slot, we still take care to check. We have used isinstance(), but we could have used hasattr(button, "text") instead. If we had connected all the buttons to this slot, it would have worked correctly for them all.

Some programmers don't like using sender() because they feel that it isn't good object-oriented style, so they tend to use partial function application when needs like this arise.

There is actually one other technique that can be used to get the effect of wrapping a function and a parameter. It makes use of the QSignalMapper class, and an example of its use is shown in Chapter 9.

QSignalMapper ☞ 297

It is possible in some situations for a slot to be called as the result of a signal, and the processing performed in the slot, directly or indirectly, causes the signal that originally called the slot to be called again, leading to an infinite cycle. Such cycles are rare in practice. Two factors help reduce the possibility of cycles. First, some signals are emitted only if a real change takes place. For example, if the value of a QSpinBox is changed by the user, or programmatically by a setValue() call, it emits its valueChanged() signal only if the new value is different from the current value. Second, some signals are emitted only as the result of user actions. For example, QLineEdit emits its textEdited() signal only when the text is changed by the user, and not when it is changed in code by a setText() call.

If a signal–slot cycle does seem to have occurred, naturally, the first thing to check is that the code's logic is correct: Are we actually doing the processing we thought we were? If the logic is right, and we still have a cycle, we might be able to break the cycle by changing the signals that we connect to—for example, replacing signals that are emitted as a result of programmatic changes, with those that are emitted only as a result of user interaction. If the problem persists, we could stop signals being emitted at certain places in our code using QObject.blockSignals(), which is inherited by all QWidget classes and is

* It is conventional PyQt programming style to give a slot the same name as the signal that connects to it.

passed a Boolean—`True` to stop the object emitting signals and `False` to resume signalling.

This completes our formal coverage of the signals and slots mechanism. We will see many more examples of signals and slots in practice in almost all the examples shown in the rest of the book. Most other GUI libraries have copied the mechanism in some form or other. This is because the signals and slots mechanism is very useful and powerful, and leaves programmers free to focus on the logic of their applications rather than having to concern themselves with the details of how the user invoked a particular operation.

Summary

In this chapter, we saw that it is possible to create hybrid console–GUI applications. This can actually be taken much further—for example, by including all the GUI code within the scope of an `if` block and executing it only if PyQt is installed. This would allow us to create a GUI application that could fall back to "console mode" if some of our users did not have PyQt.

We also saw that unlike conventional batch-processing programs, GUI applications have an event loop that runs continuously, checking for user events like mouse clicks and key presses, and system events like timers timing out or windows being revealed, and terminating only when requested to do so.

The Calculate application showed us a very simple but structurally typical dialog `__init__()` method. The widgets are created, laid out, and connected, and one or more other methods are used to respond to user interaction. The Currency application used the same approach, only with a more sophisticated interface, and more complex behavior and processing. The Currency application also showed that we can connect multiple signals to a single slot without formality.

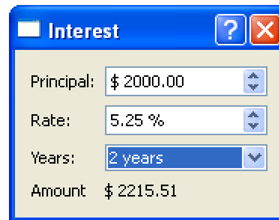
PyQt's signals and slots mechanism allows us to handle user interaction at a much higher level of abstraction than the specific details of mouse clicks and key presses. It lets us focus on what users want to do rather than on how they asked to do it. All the PyQt widgets emit signals to announce state changes and other important occurrences; and most of the time we can ignore the signals. But for those signals that we are interested in, it is easy to use `QObject.connect()` to ensure that the function or method of our choice is called when the signal is emitted so that we can respond to it. Unlike C++/Qt, which must designate certain methods specially as slots, in PyQt we are free to use any callable, that is, any function or method, as a slot.

We also saw how to connect multiple signals to a single slot, and how to use partial function application or the `sender()` method so that the slot can respond appropriately depending on which widget signalled it.

We also learned that we do not have to formally declare our own custom signals: We can simply emit them using `QObject.emit()`, along with any additional parameters we want to pass.

Exercise

Write a dialog-style application that calculates compound interest. The application should be very similar in style and structure to the Currency application, and should look like this:



The amount should be automatically recalculated every time the user changes one of the variable factors, that is, the principle, rate, or years. The years combobox should have the texts "1 year", "2 years", "3 years", and so on, so the number of years will be the combobox's current index + 1. The compound interest formula in Python is `amount = principal * ((1 + (rate / 100.0)) ** years)`. The `QDoubleSpinBox` class has `setPrefix()` and `setSuffix()` methods which can be used for the "$" and "%" symbols. The whole application can be written in around 60 lines.

Hint: The updating can be done by connecting suitable spinbox and combobox signals to an `updateUi()` slot where the calculations are performed and the amount label is updated.

A model answer is provided in the file `chap04/interest.pyw`.